CSCLP 2005
Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming

Mats Carlsson, François Fages, Brahim Hnich, and Francesca Rossi (eds.)

Workshop Proceedings

June 20–22, 2005

Uppsala, Sweden

Preface

Many companies have scheduling, assignment, supply chain and other problems that could be solved with a constraint programming toolkit. Constraint programming also attracted a large community of researchers which is particularly strong in Europe.

This workshop on Constraint Solving and Constraint Logic Programming is jointly organized as the tenth meeting of the European Research Consortium for Informatics and Mathematics (ERCIM) Working Group on Constraints and the third annual workshop of the European Network on Computational Logic (CoLogNET) area on Constraint Logic Programming. These two groups as well as the Swedish Institute of Computer Science (SICS) and Science Foundation Ireland (SFI) jointly sponsored this workshop held June 20–22 at Uppsala University in Sweden. It was hosted by the Department of Information Science (DIS).

This workshop provides a forum for researchers to share their ideas on areas related to the workshop topics. The papers in these proceedings present research into many aspects of Constraint Solving and Constraint Logic Programming. This includes search, constraint logic programming, hybrid approaches, uncertainty and preferences, global constraints, and modeling.

This volume contains 20 papers. We hope the reader will find them useful for advancing their understanding of some issues in constraint programming.

The editors wish to thank all the authors who submitted papers; our invited speakers who kindly accepted our invitation; and all the participants who made the workshop possible. We also thank ERCIM, CologNET, DIS, SICS, and SFI for their support for the workshop and the field of constraint programming in general.

Mats Carlsson, François Fages, Brahim Hnich, and Francesca Rossi
Organising Committee

Table of Contents

Analysis of Heuristic Synergies

Richard J. Wallace

Cork Constraint Computation Centre and Department of Computer Science
University College Cork, Cork, Ireland
email: r.wallace@4c.ucc.ie

Abstract. "Heuristic synergy" refers to improvements in search performance when the decisions made by two or more heuristics are combined. This paper considers combinations based on products and quotients, and then a newer form of combination based on weighted sums of ratings from a set of base heuristics, some of which result in definite improvements in performance. Then, using recent results from an factor analytic study of heuristic performance, which showed two main effects of heuristics, which involve either immediate or future failure, it is shown that heuristic combinations are effective when they are able to balance these two actions. In addition, a factor analysis that included heuristic combinations did not reveal any evidence of new 'factors' in this case. In addition to elucidating the basis for heuristic synergy (or for the lack of it), this work suggests that the task of understanding heuristic search may devolve to an analysis of these two basic actions.

1 Introduction

Combining variable ordering heuristics that are based on different features sometimes results in better performance that can be obtained by either heuristic working in isolation. Perhaps the best-known instance of this is the domain/degree heuristic of [1]. Recently, further examples have been found based on weighted sums of rated selections produced by a set of heuristics [2].

As yet, we do not have a good understanding of the basis for such heuristic synergies. Nor can we predict in general which heuristics will synergize. In fact, until now there has been no proper study of this phenomena, and perhaps not even a proper recognition that it is a phenomenon. The present paper initiates a study of heuristic synergies. A secondary purpose is to test the weighted sum strategy in a setting that is independent from its original machine learning context.

The failure to consider this phenomenon stems in part from our inability to classify heuristic strategies beyond citing features used by a heuristic. However, recent work has begun to shown how to delineate basic strategies, and with this work we can begin to understand how heuristics work in combination. Although the work is still in its early stages, it is already possible to predict which heuristics will synergise in combination and to understand to some extent why this occurs.

The analysis to be presented depends heavily on the factor analysis of heuristic performance, i.e. the efficiency of search when a given heuristic is used to order the variables. This approach is based on inter-problem variation. As will be shown, heuristics can be distinguished by the pattern of variation in search effort across problems. If the action of two heuristics is due to a common strategy, then the pattern of variation should be similar. Using this method, it has been possible to show that such variation can be ascribed to only two factors, which can in turn be interpreted as basic heuristic actions [3].

The next section describes the basic methodology. Section 3 gives results for heuristic combinations involving products and quotients. Section 4 gives results for weighted sums of heuristic ratings. Section 5 gives a brief overview of factor analysis as well as methodological details for the present work. Section 6 gives the basic results of a factor analysis of variable ordering heuristics and their interpretation, and uses these results to explain successes and failures of heuristic combinations. Section 7 gives conclusions.

2 Description of Methods

The basic analyses used a set of well-known variable ordering heuristics that could be combined in various ways. These are listed below together with the abbreviations used in the rest of the paper.

- Minimum domain size (dom, dm). Choose a variable with the smallest current domain size
- Maximum forward degree (fd). Choose a variable with the largest number of neighbors (variables whose nodes are adjacent in the constraint graph) within the set of uninstantiated variables.
- Maximum backward degree (bkd). Choose the variable with largest number of neighbors in the set of instantiated variables.
- Maximum static degree (stdeg, dg). Choose a variable with the largest number of adjacent variables in the constraint graph (variable of highest degree)

All but the last one in the list involve dynamic features of the problem. In all cases, ties were broken according to the lexical order of the variable labels.

The initial tests were done with homogeneous random CSPs because these are easy to generate according to different parameter patterns. Problems were generated according to a probability-of-inclusion model for adding constraints, domain elements and constraint tuples. In all cases generation began with a spanning tree to ensure that the constraint graph was connected. Density as given in this paper was calculated as the proportion of additional edges added to the graph. Typically, there were 100 problems in a set, although some statistics have been gathered for sets of 500 problems with the same parameters, with similar results.

Some further tests were based on other problem classes: geometric problems, which are random problems with small-world characteristics, and quasigroups-with-holes. Geometric problems are generated by choosing n points at random within the unit square, which represent the n variables, and then connecting all pairs of variables whose points lie within a threshold distance. In this case, connectivity was ensured by checking for connected components and if there were more than one, adding an edge between two variables that were separated by the shortest distance. Results to date with these problems indicates that the patterns of performance are similar overall to those found with homogeneous random problem; however, since these results are preliminary, they are not discussed specifically in the present paper.

The tests in this paper were based on the MAC-3 algorithm. The main tests were based on (i) nodes visited during search, (ii) constraint checks. In these experiments, both measures produced similar patterns of performance, so for brevity the results in this paper are restricted to search nodes.

Synergy was evaluated for two kinds of strategy. The first type of strategy was to take products and quotients of the basic heuristics, as is done in the well-known domain/degree heuristics. The second was to combine evaluations of individual heuristics into weighted sums. This strategy was derived from one used in a contemporary learning system [2]; to my knowledge there are no reports of its efficacy outside this context.

Weighted sums were obtained by assigning each individual heuristic a weight, then at each choice point allowing all heuristics to rate the choices (variables) beginning with a score of 10 as the best rating and 9 for the next-best rating, 8 for the next, and so forth down to a minimum of 1. For example, if the current domain sizes were 1, 2, 2, 3, 4, 4, 5, the corresponding ratings for min domain would be 10, 9, 9, 8, 7, 7, 6. The ratings were then combined for each choice by multiplying the rating for each heuristic by its weight and adding the products. Then, the variable with the highest weight was chosen (again, ties were broken lexically, by choosing the variable with the smallest integer label).

3 Heuristic Combinations Based on Quotients and Products

Aside from the domain/degree heuristics, it was not possible to find quotients or products that gave clearly superior performance. Interestingly, for this set of problems domain/static degree did not give better performance than static degree alone, while domain/forward degree was clearly superior to its component heuristics. (Note that for heuristic combinations involving backward degree, when this component was zero, 1 was used instead in both the quotients and products.)

Table 1. Results for Products and Quotients

simple heuristic	nodes	combination	nodes
dom	11,334	min dom/stdeg	2076
fd	2625	min dom/fd	**1621**
bkd	27,391	min dom/bkwd	15081
stdeg	2000	max fd/bkwd	2886
		max stdeg * fd	2417
		max bkwd * fd	2447

Mean nodes per problem. <50,10,0.184,0.32> problems. Bold entries show results that are better than any individual heuristic.

4 Heuristic Combinations Based on Weighted Sums

Several demonstrations of heuristic synergy, as well as a lack thereof, are shown in Table 2. This gives results, in terms of nodes searched, for five individual heuristics and for "combinations" of these heuristics using the technique of weighted sums described in Section 2. For these tests, heuristics were given equal weights.

Note that in these tests the domain/degree quotients were also used as components with respect to the weighted sums, and that this sometimes gave better results than the quotient alone. On the other hand, it was never clearly superior to the best combinations based on simple heuristics.

Table 2. Results for Weighted Sums

heuristic	nodes	combination	nodes	combination	nodes
dom	11,334	dom+dm/dg	2327	dom+dm/dg+stdeg	1654
dm/dg	2076	dom+dm/fd	**1317**	dom+fd+stdeg	**1374**
dm/fd	1621	dom+fd	**1317**	dom+fd+bkwd	**1430**
fd	2625	dom+bkwd	12521	dom+bkwd+stdeg	1822
bkd	27,391	dom+stdeg	**1427**	fd+bkwd+stdeg	1991
stdeg	2000	dm/dg+dm/fd	1800		
		dm/fd+fd	**1304**	dom+dm/dg+fd+bkwd	1657
		dm/fd+stdeg	**1386**	dom+dm/dg+fd+stdeg	**1374**
		fd+stdeg	2344	dom+dm/dg+bkwd+stdeg	1834
		dm/fd+bkwd	1861		
		bkwd+stdeg	1876	dom+dm/dg+fd+bkd+stdeg	**1470**

Mean nodes per problem. <50,10,0.184,0.32> problems. Bold entries show results that are better than any individual heuristic.

The important things to note regarding these results are:

- Some combinations do better, in terms of number of search nodes, than any heuristic used by itself. Some do even better than the best heuristic-quotient tested, which was min domain/forward-degree.
- There is considerable variation in search efficiency with different combinations. In some cases there is no improvement over the best heuristic in the mix; in some there is marked improvement.
- The best results for combinations of two heuristics were as good as the best results for combinations of more than two heuristics.

In addition, weighted sums gave better results than tie-breaking strategies based on the same heuristics. For comparison, here are results on the same set of problems with four tie-breaking strategies:

- min domain, ties broken by max forward degree: 3101
- min domain, ties broken by max static degree: 3155
- forward degree, ties broken by min domain: 2239
- static degree, ties broken by min domain: 1606

Note that, as expected, tie-breaking does reduce the size of the search tree in comparison with the primary heuristic when used alone.

Another significant result is that, for the best combinations of *two* heuristics, equal weights gave better results than unequal weights, and in these cases performance deteriorated as a function of the difference in weights. This is shown in Table 3. In cases in which weight combinations did not 'synergize' or synergized weakly in comparison with the best individual heuristic in the combination, unequal weights sometimes gave some improvement, although the effect was never marked.

Table 3. Two-Heuristic Combinations with Different Weights

wt pattern	dom+fd	dom+stdeg	stdeg+bkwd	stdeg+fd
1:1	1317	1427	1876	2344
1:2	1433	**1420**	2471	2455
2:1	1405	1620	**1852**	**2235**
1:3	1652	1454	3054	2458
3:1	1651	1885	**1812**	**2223**
1:5	2033	1557	3960	2458
5:1	2368	2504	**1816**	**2223**

Notes. Mean nodes per problem. $<$50,10,0.184,0.32$>$ problems. The weight pattern, read from left to right, corresponds to the weights given to each heuristic each combination, again reading from left to right. For ease of comparison, the results for equal weights are repeated in the first row of the table. Bold entries show results that are better than the 1:1 condition.

In contrast to the two-heuristic case, when three or more heuristics were used in combination, the best results were obtained when one of the three was weighted more heavily than the others. This is shown in Tables 4 and 5. An interesting feature of these results is that in each group, the best performance occurs when a particular heuristic is weighted more strongly than the others. For example, in Table 4, for the dom+stdeg+fd combination, all of the best results occurred when dom was weighted more highly than the other two heuristics. Similarly, for the second and third combination, the best results occurred when stdeg was more highly weighted. In Table 5, all of the best results ($<$ 1400 nodes) occurred when fd was highly weighted, and all results that bettered the equal-weight condition involved weighting fd and/or stdeg more highly than the other heuristics. (In this connection, note also that in the one case where weighting one of these heuristics more strongly did not better the equal-weight condition [third line from bottom in Table 5], dom and d/deg were also weighted highly.)

Table 4. Three Heuristics with Different Weights

wt pattern	dom+stdeg+fd	dom+stdeg+dm/dg	dom+stdeg+bkwd
1:1:1	1374	1654	1822
3:1:1	**1372**	1947	2229
1:3:1	1519	**1406**	**1510**
1:1:3	1717	1923	2847
3:3:1	**1348**	**1499**	**1548**
3:1:3	**1331**	2043	2952
1:3:3	1767	**1570**	**1791**

Mean nodes per problem. <50,10,0.184,0.32> problems. Other conventions as in Table 3.

Table 5. Five Heuristics with Different Weights

wt pattern					nodes
dom	stdeg	dm/dg	fd	bkwd	
1	1	1	1	1	1470
3	1	1	1	1	1635
1	3	1	1	1	**1409**
1	1	3	1	1	1649
1	1	1	3	1	**1332**
1	1	1	1	3	1741
3	3	1	1	1	**1448**
1	3	3	1	1	1481
1	1	3	3	1	**1401**
1	1	1	3	3	**1414**
3	1	1	3	1	**1343**
1	3	1	3	1	**1376**
3	3	3	1	1	1562
3	3	1	3	1	**1342**
3	1	3	1	3	1933

<50,10,0.184,0.32> problems. Other conventions as in Table 3.

In this work, the main interest is in explaining heuristic performance, so even if the methods described here are not practical, it is of interest to understand why search efficiency in terms of nodes expanded is made more efficient. The effects shown in Tables 2-5 are rather modest, and in fact the decrease in search nodes is offset by the expense of calculating weights. The latter does have an inherently higher time complexity due to the sorting requirement. However, it is worthwhile to see if the effects observed here will scale up. Some preliminary data on this point has been obtained for large ‘easy’ problems (<200,6,0.054,0.2>). For these problems min domain/forward degree gave a mean of 3886 search nodes, while the combination of five heuristics given in Table 2 gave a mean of 1293; in this case there was also a 50% reduction in average time. Evidently, then, for some problems the effects can scale up, so if efficient means can be found for computing weighted sums (or approximation), then this technique may be of practical importance.

5 Factor Analysis of Heuristic Performance

We turn now to the explanation of why search performance is sometimes improved (and sometimes worsened) by combining heuristics. To this end, a statistical technique called "factor analysis" was employed.

Factor analysis is a statistical technique for determining whether a set of measurements can be accounted for by a smaller number of "factors". Strictly speaking, the notion of factor is solely statistical and refers either to a repackaging of the original patterns of variation across individuals and measurements (variance) or to a set of linear relations that can account for the original statistical results. However, since variation must have causes, this technique if used carefully can yield considerable insight into the causes underlying the measurements obtained. In other words, the factors may be closely related to underlying variables which are sufficient to account for much of the variance.

The strategy here was to see, (i) if patterns of variation in search effort could be ascribed to a relatively few factors in the factor analysis, (ii) whether such factors could, in turn, be ascribed to any definable causal variables, (iii) whether such variables, if discovered, could serve to explain the patterns of synergy or non-synergy for heuristic combinations.

5.1 Resumé of factor analysis

The basic factor analysis model can be written as a set of linear relations of the form (taken from [4]):

$$z_j \;=\; a_{j1}F_1 + a_{j2}F_2 + \ldots + a_{jm}F_m + d_jU_j \quad (j \;=\; 1, 2, \ldots, n)$$

for the jth measure, where the F_i are *common factors*, i.e. factors that are common to some of the original measures, and U_j is a *unique factor* associated with measure j. Usually $m << n$. The correlation coefficients a_{jk} are often referred to as "loadings". The square of the coefficient of U_j is referred to as the uniqueness, because this is the portion of the variance unique to measure j. In this paper, we abuse terminology slightly by using this term in connection with the coefficient d_j; we can do this without confusion because high values of d_j also give high values for the uniqueness.

Factor analysis is based on a matrix of correlations, derived from a sample of n values for each measure. For example, if the measurements were scores on cognitive or personality tests, then the correlation between test i and test j would be based on scores from n individuals. In the present experiments, the individuals are individual problems, and each measurement is an efficiency measure, such as search nodes or constraint checks, for a given heuristic. A factor extraction process is applied, based on whatever method has been chosen. The present work uses the method of maximum likelihood, which starts from a hypothesis of m common factors and determines maximum-likelihood estimates of them using the original correlation matrix [5].

Factor analysis methods such as the maximum likelihood method obtain factors that are uncorrelated with each other. In this case, each a_{jk} above is equal to the correlation coefficient for z_j and F_k [4].

Once obtained, the factors (which constitute a basis for a space of m dimensions) can be rotated according to various criteria. Here the varimax rotation was used; this method tries to eliminate negative loadings while producing maximal loadings on the smallest possible set of measures.

The interpretation of patterns of differences cannot assume that causal factors behave additively, only that patterns of variation can be derived from additive combinations. Factor analysis, therefore, can only identify common sources of variation whose interpretation requires further investigation.

5.2 Methodology

The software used in these analyses was System R, which was downloaded from

http://www.r-project.org.

In this package, the factanal function was used for the factor analysis.

As already noted, maximum likelihood methods require the number of factors as input. Since the number of significant factors was not known beforehand, various numbers of factors were tested, first, to determine at what point factor extraction ceased to account for any significant part of the variance, second, to determine which of these factors gave strong, reliable results. The first kind of test, therefore, can be taken as setting an upper bound on the number of useful factors.

If there are other sources of variation than the ones emphasized here, since they are less important in their effects and less reliable across experiments, they are likely to be related to features of specific problem sets interacting with vagaries of the search process. In addition, the possible existence of further factors does not necessarily diminish the importance of the ones demonstrated here. (In other words, the explanatory process may have to proceed further but it will not need to backtrack if the arguments for the factors described here are cogent.)

6 Factor Patterns for CSP Heuristics and Their Interpretation

6.1 Basic results

Table 6 shows selected results for an analysis based on 12 heuristics (described more fully in [3]). (The heuristics not included in the table were mainly diagnostic pseudo-heuristics: the FFx series of [6] and a promise heuristic, the dual of the one described in [7].) On the left are results from the basic experiment with this set of problems. In this case, the analysis indicated that there were two major factors, but that min domain and max backward degree had idiosyncratic patterns of variation, reflected in their high uniqueness. Further experiments showed that the latter were due to random choices made at the top of the search tree; this occurs because for these heuristics there is no distinction among variables at the start of search. The results of one of these experiments are shown on the right hand side of the table. In this test, for each measurement the first three choices were in lexical order; thereafter, one of the heuristics was used. In this experiment, therefore, the effect of initial random selections was equalized. As a result, all heuristics had moderate to high loadings on two major factors, and the proportion of the total variance accounted by these two factors was 95%.

Table 6. Factor Analysis for CSP heuristics

heuristic	heuristic alone				3 lexical, heuristic			
	nodes	factor 1	factor 2	uniqueness	nodes	factor 1	factor 2	uniqueness
dom	11334	0.146	0.281	**0.900**	19587	**0.804**	0.565	0.034
fd	2625	0.443	**0.873**	0.042	9551	0.602	**0.796**	0.005
bkd	27391	0.107	0.224	**0.938**	37536	**0.708**	0.488	0.261
stdeg	2000	0.486	**0.835**	0.067	7980	0.648	**0.752**	0.015
dm/dg	2076	**0.913**	0.394	0.011	7712	**0.752**	0.652	0.010
dm/fd	1621	**0.909**	0.404	0.010	6473	**0.744**	0.660	0.011
dg*fd	2418	0.436	**0.897**	0.005	8567	0.626	**0.775**	0.008

Notes. $<$50,10,0.184,0.32$>$ problems. Loadings $\geq$ 0.7 are shown in bold.

6.2 Interpretation of the factors

Various lines of evidence suggest the following interpretation of these factors (discussed at greater length in [3]). One factor appears to emphasize immediate failure, so that heuristics such as min domain and backward degree load highly on it (when confounding variables are removed), as well as the FF series of [6] (not shown here). The other factor appears to emphasize future failure, so heuristics such as max forward degree and other look-ahead heuristics load highly on it. These have been tentatively designated as “contention” and “propagation” factors, respectively. It is of interest to note that both act as fail-first strategies, since the same factor pattern was found for problems with the same parameters that had no solutions. It is also important to bear in mind that factor analysis guarantees that the factors are uncorrelated, which is a further argument for considering that there are two independent actions underlying the different heuristics.

6.3 Explanation of heuristic synergies

If we reconsider the results of Table 2 in the light of this analysis, we see that as a rule successful weighted sums were composed of heuristics that load on each of the two major factors. An interesting example is dom+fd, since in this case neither component heuristic was outstanding. There are some apparent exceptions, such as dom+dm/fd; but here the contrast with dom+dm/dg is important because this shows that emphasizing forward degree rather than degree was critical, and again this suggests the action of both *causal* factors.

Evidence that the *balance* between the two factors is important can be found in Tables 3-5. This, in fact, seems to explain both the decline in quality for the two-heuristic combinations when the weights are made unequal (Table 3) and the distinguished heuristic phenomenon that was noted in the data shown in Table 4. In each case, the single heuristic that loaded on a different factor from the other two heuristics was the one that needed to be weighted more strongly. A similar phenomenon seems to be involved in the pattern of results shown in Table 5.

The factor analysis was also useful in showing why certain combinations were especially bad. These combinations involved both dom and bkwd, and these are the heuristics that start with what is essentially a random choice.

From this, it can be concluded that successful heuristic synergies involve both major factors. It is worth noting that domain/degree, which is one of the most efficient heuristics known, is itself a combination of heuristics, which as the results above show are associated with different major factors.

Results of a factor analysis involving some of the heuristic combinations, as well as the building-block heuristics, are shown in Table 7. The main finding is that the factor structure appears to be the same as in the original analysis. That is, two major factors account for most of the variance, with much of the rest due to the uniqueness values for min domain and max backward degree. In some sense, therefore, heuristic combinations are not adding anything new; instead, the improvement they bring appears to be the result of handling the contention and propagation factors (i.e. immediate and future failure) more effectively. It is also of interest that most (but not all) heuristic combinations load most highly on the ‘contention factor’.

Table 7. Factor Analysis for Weighted Sums

heuristic	nodes	factor 1	factor 2	uniqueness
dom	11334	0.150	0.182	**0.829**
d/d	2076	**0.891**	0.363	0.012
d/fd	1621	**0.887**	0.355	0.012
fd	2625	0.418	**0.768**	0.045
bkd	27391		0.151	**0.897**
dg*fd	2417	0.417	**0.797**	0.005
stdeg	2000	0.482	**0.835**	0.005
dm/dg+fd	1369	**0.831**	0.396	0.012
dom+fd	1317	**0.820**	0.437	0.033
dom+stdeg	1427	**0.846**	0.411	0.005
stdeg+fd	2344	0.417	**0.803**	0.005
dom+dm/dg	2327	**0.889**	0.376	0.017
dm/fd+fd	1304	0.497	**0.824**	0.007
dm+stdeg+fd	1374	**0.768**	0.490	0.078
stdeg+fd+bkwd	1991	0.487	**0.834**	0.005
dom+fd+bkwd	1430	**0.845**	0.417	0.005
dom+stdeg+dm/dg	1654	**0.888**	0.339	0.005
dom+stdeg+dm/dg+fd	1374	**0.844**	0.383	0.005
dom+stdeg+dm/dg+bkwd	1834	**0.881**	0.348	0.006
dom+stdeg+fd+bkwd	1424	**0.792**	0.460	0.054
five	1470	**0.849**	0.353	0.005

Note. Blank cell indicates factor loading < 0.1, which is not printed by the System R factanal function.

Some preliminary data has also been collected on promise and fail-first measures (Table 8). This involved max forward degree, min domain and the weighted sum of the two (with equal weights; cf. Table 2). As shown elsewhere, the proper measure of fail-firstness is mistake-tree size [8]. This is the size of an insoluble subtree rooted at the first bad assignment (the “mistake”). Other measures related to fail-firstness are also shown (in rows 1-3 of the table), since they give futher insight into the action of the heuristics. The number of mistakes is also included and is a rough measure of a heuristic’s promise. These data indicate that this heuristic combination shows better adherence to both the fail-first and promise policies than the component heuristics acting alone.

Table 8. Fail-First Measures for Heuristics and Weighted Sum

measure	max forward degree	min domain	dom+fd
fail-depth	5.54	12.44	6.94
fail-length	3.73	9.66	4.89
no. fails	16104	74783	6668
mistake-tree size	139.5	844.9	102.2
no. mistakes	231.2	407.5	150.5

Note. Means for $<$50,10,0.184,0.32$>$ problems.

7 Conclusions

This paper presents a study of the effects of combining heuristics. The most interesting direct result was that weighted sums are in fact quite good at improving search performance in terms of search nodes; this must be because this strategy improves the quality of variable selection.

Some insight was gained into the basis for this improvement, using the recent discovery that there are two basic types of heuristic action: a contention or immediate failure effect and a propagation effect that presumably affects future failure [3]. Evidence, based on factor analyses of heuristic performance, suggests that heuristic combinations work to improve search performance by balancing these two effects. Thus, when the two are not well-balanced, as in some of the conditions listed in Tables 1-5, performance is not improved and can even deteriorate in comparison with the component heuristics.

In addition to showing us where to look for meaningful causal relations, factor analysis has the added strength of ‘bounding our quest’ by giving us some idea of the number of significant variables involved in a domain of study such as heuristic search. In the present work, it has produced evidence that the two effects described here are sufficient to explain nearly all the variance across problems for a given set of heuristics (aside from variance due to blind selection at the top of the search tree). If this conclusion holds up, it narrows the task of understanding heuristic action to elucidating the ways in which heuristics affect these two basic actions and how the balance of these actions serves to improve the overall efficiency of search.

Acknowledgment. This work was supported by Science Foundation Ireland under Grant 00/PI.1/C075.

References

1. Bessière, C., Régin, J.C.: MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Freuder, E.C., ed.: Principles and Practice of Constraint Programming -CP’96. LNCS. No. 1118, Berlin, Springer (1996) 61–75

2. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In van Hentenryck, P., ed.: Principles and Practice of Constraint Programming - CP2002. LNCS. No. 2470, Berlin, Springer (2002) 525–540
3. Wallace, R.J.: Factor analytic studies of heuristics. submitted for publication. (2005)
4. Harman, H.H.: Modern Factor Analysis. 2nd edn. University of Chicago, Chicago and London (1967)
5. Lawley, D.N., Maxwell, A.E.: Factor Analysis as a Statistical Method. 2nd edn. Butterworths, London (1971)
6. Smith, B.M., Grant, S.A.: Trying harder to fail first. In: Proc. Thirteenth European Conference on Artificial Intelligence-ECAI'98, John Wiley & Sons (1998) 249–253
7. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proc. Tenth European Conference on Artificial Intelligence-ECAI'92. (1992) 31–35
8. Beck, J.C., Prosser, P., Wallace, R.J.: Trying again to fail-first. In: Recent Advances in Constraints. Papers from the 2004 ERCIM/CologNet Workshop-CSCLP 2004. LNAI No. 3419, Berlin, Springer (2005) 41–55

A CSP Search Algorithm with Reduced Branching Factor

Igor Razgon and Amnon Meisels

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
{irazgon,am}@cs.bgu.ac.il

Abstract. This paper presents an attempt to construct a "practical" CSP algorithm that assigns a variable with 2 values at every step. Such a strategy has been successfully used for construction of "theoretical" constraint solvers because it decreases twice the base of the exponent of the upper bound of the search algorithm.
We present a solver based on the strategy. The pruning mechanism of the algorithm resembles Forward Checking (FC), therefore we term it 2FC. According to our experimental evaluation, 2FC outperforms FC on graph coloring problems and on non-dense instances of randomly generated CSPs.

1 Introduction

Partition of domains is a strategy that allows to reduce the size of the search space explored by a constraint solver. Consider a method based on the strategy that partitions the domain of every variable into subsets of two values (if a domain has an odd size than one subset is a singleton). The algorithm scans all constraint networks obtained by restriction of the domain of every variable to one of the partition classes. If at least one constraint network being scanned is soluble, the algorithm returns the solution found. Otherwise, it reports failure.

Assume that the considered constraint network has n variables and the maximal domain size is d. Then a simple analysis shows that the algorithm scans $O((d/2)^n)$ constraint networks if d is even and $O(((d+1)/2)^n)$ if d is odd. Taking into account that constraint networks with domains of size at most two can be solved efficiently [7], we get that these upper bounds determine the search space size of the considered algorithm. Clearly, this size is much smaller than $O(d^n)$ of FC, MAC, and others.

The strategy of domain partition has been successfully applied to the design of constraint solvers with exact upper bounds. Sophisticated techniques based on the strategy are presented in [5, 1]. However, the strategy attracted little attention of researchers that investigate "practical" approaches to construction of complete constraint solvers.

This paper introduces an attempt to construct a "practical" complete constraint solver based on domain partition. The main problem with the strategy is that the resulting algorithm is not guaranteed to have a solution when all variables are assigned. Therefore the algorithm could generate many "long" insoluble 2-CNs (constraint networks with domain sizes at most 2) which makes the actual time of its work close to the theoretical upper bound. To overcome this difficulty, we introduce the following modifications.

- Variables are assigned one by one. A variable is assigned with 2 values if its current domain contains at least 2 values. Otherwise, the variable is assigned with one value. To present this situation in a more general form, we say that at every step of the algorithm, a variable v is assigned with a subset S of its current domain.
- Whenever a variable v is assigned with a set of values S, the algorithm removes all values of unassigned variables that are incompatible with all the values of S.
- The algorithm backtracks when the current domain of some variable becomes empty.
- Whenever a variable v is assigned with a set $\{val_1, val_2\}$, a new conflict is added between any pair $(\langle v_1, val'_1\rangle, \langle v_2, val'_2\rangle)$ of values of different unassigned variables v_1 and v_2 such that $\langle v_1, val'_1\rangle$ conflicts with $\langle v, val_1\rangle$ and $\langle v_2, val'_2\rangle$ conflicts with $\langle v, val_2\rangle$.

The proposed algorithm resembles FC with the main difference that a variable can be assigned with 2 values. Therefore we call the algorithm 2FC. The main property of 2FC is that whenever all variables are assigned, the resulting 2-CN is *guaranteed* to have a solution.

In our experimental analysis, we compared 2FC to FC. The main experimental observation is that 2FC strictly outperforms FC on graph k-coloring problem. The rate of runtime improvement changes from a factor of 2 to 10. 2FC also outperforms FC on randomly generated constraint networks with low density.

The experimental result suggest that the proposed approach could be useful in the area of graph coloring. Another possible application of 2FC follows from the fact that it returns a 2-CN which could have many solutions that can be efficiently generated. Therefore the algorithm could be useful for dynamic environments where constraints frequently change: it might allow quick replacing of an inconsistent solution by another solution without performing search.

The rest of the paper is organized as follows. Section 2 provides necessary background. Section 3 presents the 2FC algorithm. Section 4 proves correctness of the 2FC algorithm. Section 5 demonstrates result of experimental evaluation. Section 6 outlines directions of further investigation.

2 Preliminaries

The model we consider in the paper is binary constraint network (CN). A CN $Z = \langle V, D, C\rangle$ is a triple consisting of a set of *variables* V, a set of *domains* D and a set of *constraints* C. Let $V = \{v_1, \dots, v_n\}$. Then $D = \{D(v_1), \dots, D(v_n)\}$, where $D(v_i)$ is the domain of values of v_i, $C = \{C(v_i, v_j) | i \neq j, 1 \leq i, j \leq n\}$, where $C(v_i, v_j) \subseteq D(v_i) \times D(v_j)$ is the set of all *compatible* pairs of values of v_i and v_j. We refer to the parts of Z as V_Z, D_Z, and C_Z. To emphasize that a value val belongs to the domain of a variable v, we refer to this value as $\langle v, val \rangle$. In this paper we consider a special case of constraint network, *2-CN*, that is a CN for which every domain has at most 2 values.

Given a CN Z as above, the task of Constraint Satisfaction Problem (CSP) is to find a set $P = \{\langle v_1, val_1 \rangle, \ldots, \langle v_n, val_n \rangle\}$ such that every val_i belongs to the domain of v_i and all values of P are mutually compatible (consistent), or to report that there is no such a set. The set P is called a *solution* of Z.

A typical CSP search algorithm like Forward Checking (FC) [4] usually creates a solution in an iterative manner. During its work it maintains a consistent set of values of a subset of variables and tries to extend it to a solution. This set of values is called a *partial* solution. Variables whose values are contained in the partial solution are called *assigned*. Other variables are called *unassigned*. The present paper introduces an algorithm that can assign a variable with more than one value. We refer to a set of assigned values maintained by the algorithm as an *extended partial solution*. When all variables are assigned, the algorithm has an *extended solution*.

The following notion is frequently used further in the paper.

Definition 1. *Let Z be a CN and let S be a set of values of Z. A subnetwork Z' of Z induced by S is obtained as follows:*

- *take to Z' only those variables of Z whose values appear in S;*
- *the domain of every variable v of Z' is the intersection of the domain of v in Z with S;*
- *two values are compatible in Z' if and only if they are compatible in Z.*

To illustrate the notion, consider Figure 1. On the left side of the figure there is a CN Z. Ellipses represent variables, black points represent values, conflicting values are connected by arcs. The CN Z' is a subnetwork of Z induced by the set of values encircled by additional circles. Note that Z' does not contain variable V_4 because no value of the domain of V_4 is included in the inducing set of values. Also note that there are conflicts between $\langle V_1, 2 \rangle$ and $\langle V_2, 2 \rangle$ and between $\langle V_2, 3 \rangle$ and $\langle V_3, 3 \rangle$ because these conflicts appear in the original CN.

Finally, we recall the notions of directional arc and path-consistency.

Definition 2. *A value $\langle v_i, val \rangle$ is consistent with a set of values S if it is compatible with at least one value of S.*

Definition 3. *A CN Z is called directionally arc-consistent with respect to an order $v_1, \ldots, v_n$ of its variables if every value $\langle v_i, val \rangle$ is consistent with a domain of a variable v_k whenever $k < i$.*

Definition 4. *A pair of values $\{\langle v_i, val_i \rangle, \langle v_k, val_k \rangle\}$ is consistent with a set of values S if at least one value of S is compatible with both $\langle v_i, val_i \rangle$ and $\langle v_k, val_k \rangle$.*

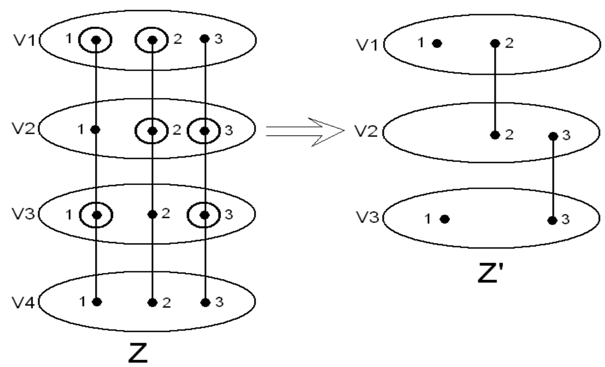


Fig. 1. Illustration of a CN induced by a set of values.

Definition 5. *A CN Z is called directionally path-consistent with respect to the order* $v_1, \ldots, v_n$ *of its variables if every pair of compatible values* $\{\langle v_i, val_i\rangle, \langle v_k, val_k\rangle\}$ *is consistent with the domain of a variable* v_l *whenever* $l < i$ *and* $l < k$*.*

3 The 2FC Algorithm

In this section we introduce a modification of FC that explores much smaller search space than FC. In particular, processing a CN with n variables and maximal domain size d, the algorithm generates a search tree with $O((d/2)^n)$ nodes if d is even and $O(((d+1)/2)^n)$ nodes if d is odd.

The first 3 steps of the modification are the following.

1. Variables are assigned with subsets of their current domains. In particular, let v be the variable being assigned currently. If the current domain of v is of size at least 2 then v is assigned with a subset S of its domain such that $|S| = 2$. Otherwise, if the domain is a singleton, v is assigned with the domain itself.
2. Once a variable v is assigned with a set S, the algorithm removes from the domains of unassigned variables all the values that are incompatible with all the values of S.
3. Unassigning a variable v at the backtrack stage, the algorithm removes from the current domain of v *all* values of the set assigned to v.

These 3 steps naturally generalize FC, providing the ability to assign a variable with one value as well as with two values. However the resulting algorithm has an essential drawback: when all variables are assigned, the CN induced by the assigned values may be insoluble and this is in contrast to the standard FC that has a solution when all variables are assigned. Thus the modified FC has a restricted ability of early recognition of dead-ends.

To illustrate this drawback, consider a CN with variables $\{v_1, v_2, v_3, v_4\}$ and the domain of every variable $\{1, 2, 3\}$. Every 2 variables are connected by the inequality constraint. Assume that the algorithm assigns v_1 with $\{1,2\}$. No value is deleted from the domains of the unassigned variables because there are values incompatible with 1, values incompatible with 2, but none that are incompatible with both of them. In the same way, the algorithm can assign v_2, v_3, v_4 with $\{1,2\}$. The CN induced by the assigned values is clearly insoluble.

The following claim (proved in the next section) suggests a simple way to overcome the drawback.

Lemma 1. *Let Z be a 2-CN with no empty domain which is directionally arc-consistent and directionally path consistent with respect to an order $v_1, \dots, v_n$ of variables of Z. Then Z is soluble.* [1]

Thus, to guarantee that whenever all the variables are assigned, the CN induced by the assigned values is soluble, it is enough to ensure that it is directionally arc-consistent and directionally path-consistent.

Note that directional arc-consistency is already ensured by the modification number 2 described above. To ensure directional path-consistency, it is possible to perform the following operation: *Every time a variable v is assigned with a set S, add a conflict between every pair P of compatible assignments of future variables such that P is inconsistent with S.*

We call the resulting algorithm 2FC. Algorithm 1 introduces its pseudocode.

The algorithm is presented in the form of a recursive procedure that gets a CN Z as input. In the first 6 lines the termination conditions are checked. In particular, if Z has no variables, the procedure returns $\emptyset$ (lines 1-3), if Z has a variable with the empty domain, the procedure returns $FAIL$.

In line 7 a variable u is selected. In line 8 a CN Z' is created by removing from Z the variable u and all its values. Before exploring the domain of u, the algorithm removes from it all values that conflict with domains of some other unassigned variable (line 9).

The loop described in lines 10-27 explores the domain of u. In lines 11-15 a subset S of the current domain of u is selected. The set S contains 2 values unless there is only one value in the current domain of u. After u is assigned by S, the algorithm removes from the domains of variables of Z' all values inconsistent with S (line 16). If the size of S is 2, the algorithm adds conflicts between compatible pairs of values of Z' that are inconsistent with S (lines 18-20). Then the function 2FC is applied recursively to Z' (line 22). If the output of the recursive application is not $FAIL$, the procedure returns the union of the output with S (line 24). (The operation is correct because the output of 2FC is either $FAIL$ or a set of values.) Otherwise, if the recursive application returns $FAIL$, the algorithm removes S from the domain of u (line 26) and starts a new iteration of the loop or finishes it if the domain of u is wiped out. In the latter case the algorithm returns $FAIL$ in line 28.

Note that when 2FC returns a set of values, a solution can from it by the process described in the proof of Lemma 1. (See the next section.)

[1] Note that the suggested sufficient condition of solubility of 2-CNs is weaker than path-consistency whose sufficiency is proved in [7].

Algorithm 1 FUNCTION 2FC(Z)

if $V_Z = \emptyset$ **then**
  Return $\emptyset$
end if
if There is a variable with the empty domain **then**
  Return $FAIL$
end if
Select a variable u
$Z' \leftarrow Z \setminus u$
Remove from $D_z(u)$ all values that are inconsistent with domains of unassigned variables
while $D_z(u) \neq \emptyset$ **do**
  if $|D_z(u)| \geq 2$ **then**
    $S \leftarrow \{val_1, val_2\}$, where val_1 and val_2 are two values of $D_z(u)$
  else
    $S \leftarrow D_z(u)$
  end if
  Remove from the domains of Z' the values that are inconsistent with S
  if $|S| \leq 2$ **then**
    for every pair $\{\langle v_1, val'_1\rangle, \langle v_2, val'_2\rangle\}$ of compatible values of Z' that is inconsistent with S **do**
      $C_{Z'}(v_1, v_2) \leftarrow C_{Z'}(v_1, v_2) \setminus \{\{\langle v_1, val_1\rangle, \langle v_2, val_2\rangle\}\}$
    end for
  end if
  $R \leftarrow 2FC(Z')$
  if $R \neq FAIL$ **then**
    Return $R \cup S$
  end if
  $D_Z(u) = D_Z(u) \setminus S$
end while
Return $FAIL$

Consider an example of application of 2FC. Let Z be the CN described above with variables $\{v_1, \ldots, v_4\}$, each domain equal $\{1, 2, 3\}$ and variables connected by inequality constraint. Assume that v_1 is assigned with $\{1, 2\}$. Then 2FC adds conflicts between every pair of values 1 and 2 of different unassigned variables, that is between $\langle v_2, 1\rangle$ and $\langle v_3, 2\rangle$, between $\langle v_2, 2\rangle$ and $\langle v_3, 1\rangle$ and so on. Assume that in the next iteration, variable v_2 is assigned with $\{1, 2\}$. Then values 1 and 2 are deleted from the current domains of v_3 and v_4. In the next iteration, trying to assign v_3, 2FC backtracks, because the only remaining values $\langle v_3, 3\rangle$ wipes out the current domain of v_4. As a result of backtrack, 2FC unassigns v_2. The only possible next assignment is $\{3\}$. After the assignment, 2FC removes 3 from the domains of v_3 and v_4. In the next iteration, 2FC tries again to assign v_3, but backtracks because both remaining values 1 and 2 wipe out the domain of v_4. This time, after unassigning v_2, the current domain of v_2 is finished; therefore 2FC backtracks again and changes the assignment of v_1 to $\{3\}$. In a few iteration the algorithm finishes with FAIL because of wiping out of the current domain of v_1.

A drawback of 2FC is large overhead spent to addition of conflicts between values of unassigned variables. It is not hard to show that $O(n^2d^2)$ additional consistency checks per iteration must be spent. The overhead can be reduced if we observe that 2FC checks compatibility of values of two variables only if one of these variables is either assigned or selected to be assigned.

Based on the observation we suggest a procedure of adding new conflicts based on the notion of *critical value.* Let u be a variable assigned with a set $\{val_1, val_2\}$. We say that $\langle u, val_1\rangle$ is critical with respect to a value $\langle v, val\rangle$ if $\langle v, val\rangle$ conflicts with $\langle u, val_2\rangle$. Instead of performing lines 18-19 in Algorithm 1, new conflicts can be added in the following "lazy" way. *Whenever a new variable v is selected to be assigned, a conflict is added between every pair of compatible values $\langle v, val\rangle, \langle w, val'\rangle$ that satisfies the following conditions:*

- *w is an unassigned variable other than u;*
- *val' belongs to the current domain of w;*
- *$\langle w, val'\rangle$ conflicts with at least one critical value with respect to $\langle v, val\rangle$.*

One can calculate that the suggested technique of updating of constraints takes $O(n^2d)$ consistency checks per iteration. We use the technique in our implementation of 2FC. We decided not to describe the method directly in the pseudocode because it reduces readability of the code and makes the correctness proof more complicated.

4 Theoretical Analysis.

In this section we prove correctness of 2FC. We start from proving Lemma 1.

Proof of Lemma 1 By induction on n, the number of variables of Z. It is trivial for $n = 1$. For $n > 1$, assign v_n with a value val_n that belongs to its domain. Let Z' be a 2-CN obtained from Z by removing v_n and deleting from the domains of the rest of variables all values that are incompatible with $\langle v_n, val_n\rangle$. Observe the following properties of Z'.

- The domains of all variables of Z' are not empty. Really, an empty domain of some variable v in Z' would mean that $\langle v_n, val_n \rangle$ conflicts with all the values of the domain of v in Z in contradiction to the directional arc-consistency of Z.
- Z' is directionally arc-consistent with respect to the order $v_1, \ldots, v_{n-1}$. Assume by contradiction that a value $\langle v_k, val \rangle$ is inconsistent with the domain of v_i, $i < k$. If the domain of v_i in Z' is the same as in Z, $\langle v_k, val \rangle$ is inconsistent with v_i in Z in contradiction to our assumption about directional arc-consistency of Z. Otherwise, one of the values of the domain of v_i is incompatible with $\langle v_n, val_n \rangle$, the other is incompatible with $\langle v_k, val \rangle$, while $\langle v_n, val_n \rangle$ and $\langle v_k, val \rangle$ are compatible. In this case we get contradiction with out assumption about directional path-consistency of Z.
- Z' is directionally path-consistent. For otherwise, if we have two compatible values $\langle v_k, val_k \rangle$ and $\langle v_l, val_l \rangle$ that wipe out the domain of some variable v_i, $(i < k, l)$, the same situation occurs in Z in contradiction to directional path-consistency of Z.

Thus Z' satisfies all conditions of the lemma and has $n-1$ variables, therefore it is soluble by the induction assumption. Let S be a solution of Z'. Note that all values of Z' are compatible with $\langle v_n, val_n \rangle$. Therefore $S \cup \{\langle v_n, val_n \rangle\}$ is a solution of Z. $\Box$.

The next lemma claims that every extended partial solution generated by 2FC satisfies the conditions of Lemma 1.

Lemma 2. *Every extended partial solution generated by 2FC induces a 2-CN without empty domains, directionally arc-consistent with respect to the chronological order of assignment of variables, and directionally path consistent with respect to the same order.*

Proof. By induction on the length n of the extended partial solution. It is clear that the lemma is valid for $n = 1$. For $n > 1$, let S be the considered extended partial solution and let $v_1, \ldots, v_n$ be the order according to which the variables were assigned. By the induction assumption, the extended partial solution obtained by removing v_n satisfies the conditions of the lemma. Therefore, if S violates these conditions then the values assigned to v_n violate either the directional arc-consistency or the directional path-consistency. Assume that the former holds. That is, a value val of assigned to v_n is inconsistent with all values assigned to v_k $(k < n)$. However, such a situation cannot happen because 2FC would remove val from the current domain of v_n when v_k has been assigned. For the latter, assume that a value val assigned to v_n, together with a compatible value val' assigned to some v_k are inconsistent with the set of values assigned to some v_i $(i < k, n)$. However, such a situation cannot happen as well because 2FC would add a conflict between $\langle v_k, val' \rangle$ and $\langle v_n, val \rangle$ when v_i was selected to be assigned. Thus the lemma holds for S. $\Box$

Now we are ready to claim the correctness of 2FC.

Theorem 1. *The 2FC algorithm is correct.*

Proof. To prove correctness, we have to prove that the algorithm terminates and also that it is sound and complete.

Termination is easy to verify by induction. If the underlying CN has 0 variables, the algorithm clearly terminates. Otherwise, 2FC selects a variable, assigns it with some partition class of its domain, and applies recursively to a CN created by the rest of variables (with updated constraints). By the induction assumption, every recursive application eventually finishes and also the number of partition classes in the first variable is finite so the algorithm terminates.

Soundness (solubility of an extended solution returned by 2FC) directly follows from Lemmas 1 and 2.

It follows from termination and soundness that 2FC always returns $FAIL$ when processes an insoluble CN. It remains to prove completeness, that is to show that 2FC always returns a solution when processes a soluble CN. In essence, all we have to show is that the additional conflicts generated by 2FC do not cause missing of a solution.

We prove completeness by induction on the number n of variables of the CN. Completeness follows immediately for $n = 1$. For $n > 1$, let v be the variable that 2FC selects to be assigned first. If the underlying CN is soluble then 2FC eventually assigns v with a set of values S that belongs to an extended solution. Then 2FC removes from the domains of the rest of variables all values that are inconsistent with S and adds conflicts between pairs of compatible values that wipe S out. Note that neither the removed values can be in the same solution with any value of S nor pairs of values that are made incompatible. Therefore 2FC is applied recursively to a soluble CN where it finds an extended solution by the induction assumption. □

5 Experimental Evaluation

It is not hard to show that 2FC explores $O(\lceil d/2 \rceil^n)$ nodes of the search tree and its running time is the bound multiplied by a polynomial. Clearly, this bound is much smaller than $O(d^n)$ upper bound for FC. However, we are interested to evaluate the practical merits of 2FC. To do this we compare in this section actual running times of 2FC and FC.

We implemented the algorithms in Microsoft Visual C++ 6.0 and tested them on a computer with CPU 2.4GHz and 0.2GB RAM. We used two measures of computation effort: the number of nodes visited and runtime (in seconds). For every tuple of parameters of the tested instances, the computation effort measures were obtained as average of 50 runs.

In our implementation, variables are ordered by the Fail-First heuristic [6] which takes first a variable with the smallest domain. The values of the variable being assigned are ordered according to the min-conflict heuristic, that is, values that conflict with the less number of values in the domains of unassigned variables are assigned first. (FC assigns the values one by one, while 2FC assigns them in pairs.)

We compared these algorithms on graph k-coloring problem and on randomly generated binary CNs.

Given a graph G with n vertices and k-colors, the CN that encodes the k-coloring problem for G has n variables corresponding to the vertices of G. The domain of every variable is $\{1, \ldots, k\}$. Pairs of variables that correspond to adjacent vertices of G are connected by the inequality constraint.

We generated 3 sets of instances: the first with 60 and 6 colors, the second with 45 vertices and 8 colors and the third with 30 vertices and 10 colors. For every set of instances we tried densities from 10% to 90% by steps of 5%.

The results of comparison of 2FC and FC on the first set of instances are shown on Figures 2 and 3. In this set of instances the phase transition region falls to the area of small density. Clearly, 2FC performs better than FC on this set of instances.

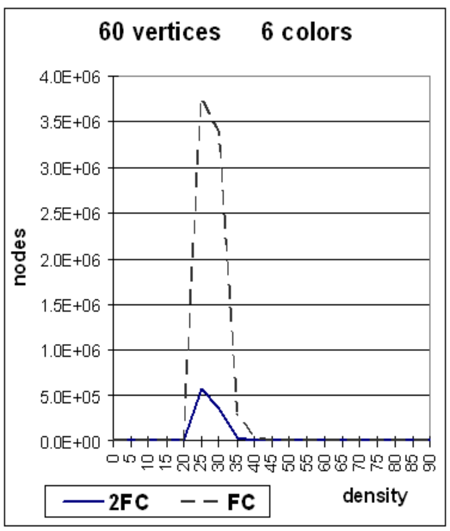


Fig. 2. 2FC vs. FC for graphs with 60 vertices and 6 colors (nodes visited)

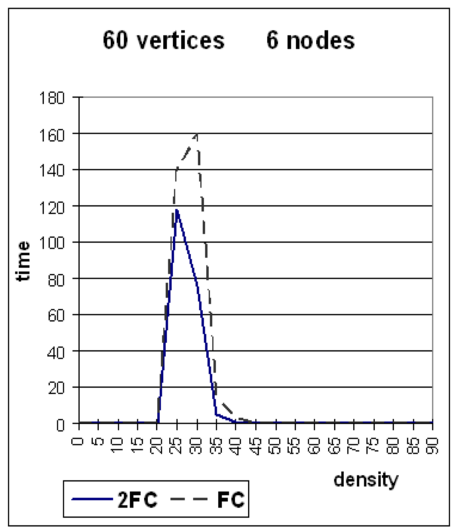


Fig. 3. 2FC vs. FC for graphs with 60 vertices and 6 colors (runtimes)

The results of comparison of 2FC and FC on the second set of instances are shown on Figures 4 and 5. In this experiment the graph that are most hard for coloring have an average density. Note that for denser graphs the rate of improvement of 2FC with respect to FC grows.

The results of comparison of 2FC and FC on the third set of instances are shown on Figures 6 and 7. In this experiments the phase transition region falls to the area of dense graphs. Note that here 2FC exhibits a larger factor of improvement as compared to the previous cases.

Comparing 2FC and FC on randomly generated CNs, we generated them using 4 parameters: the number of variables, the domain size, density, and tight-

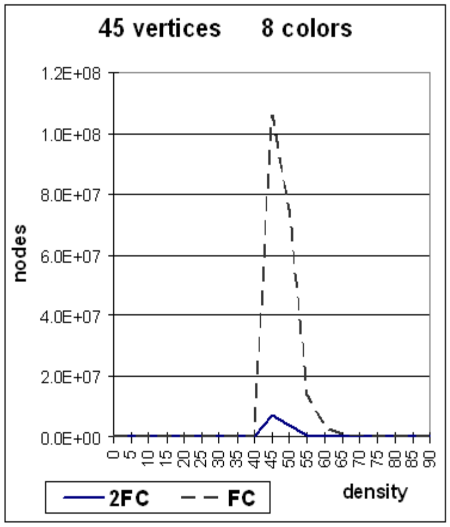


Fig. 4. 2FC vs. FC for graphs with 45 and 8 colors (nodes visited)

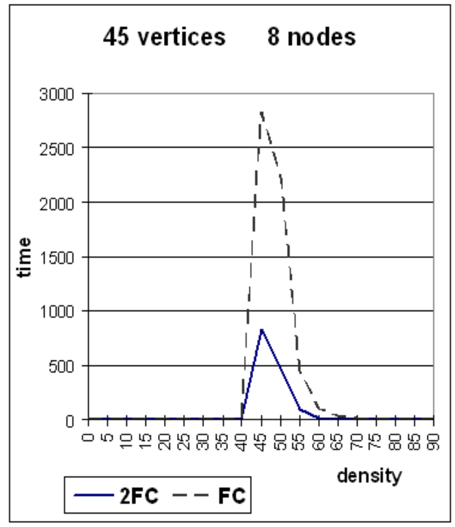


Fig. 5. 2FC vs. FC for graphs with 45 and 8 colors (runtimes)

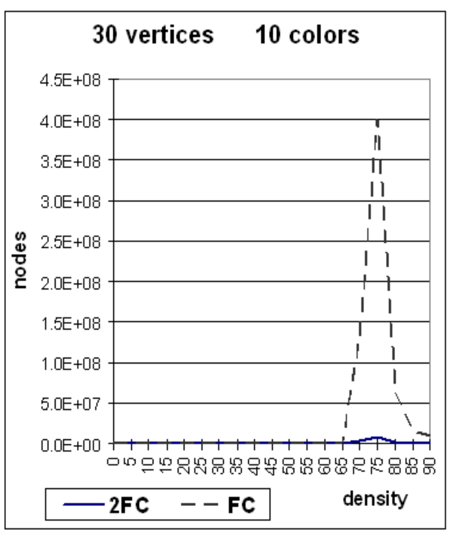


Fig. 6. 2FC vs. FC for graphs with 30 and 10 colors (nodes visited)

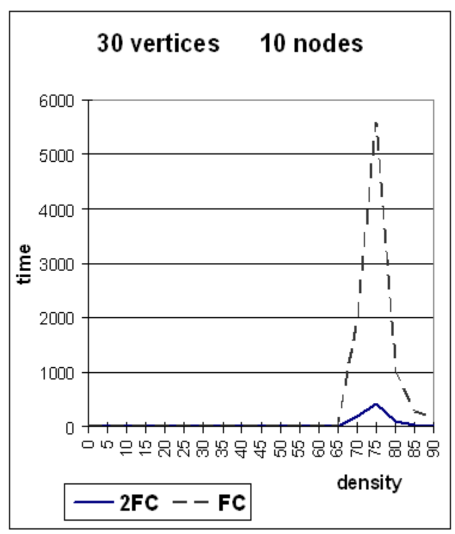


Fig. 7. 2FC vs. FC for graphs with 30 and 10 colors (runtimes)

ness [9]. To generate a set of instances, we fixed the number of variables, the domain size, and the density and varied the tightness from 10% to 90% by steps of 5%.

In the first set of experiments, the generated CNs have 60 variables domains of size 10 and density 10%. Figures 8 and 9 compare the number of nodes visited and the runtimes, respectively. We can see that 2FC outperforms FC on this set of instances.

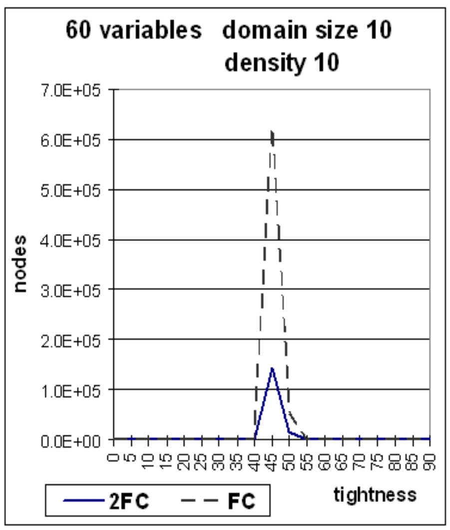


Fig. 8. 2FC vs. FC for CNs with 60 variables, domain size 10, and density 10 (nodes visited)

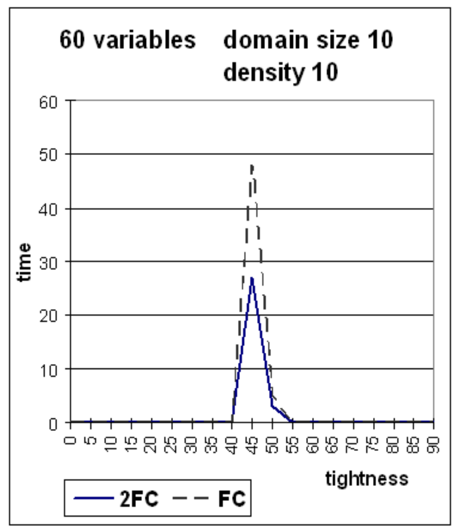


Fig. 9. 2FC vs. FC for CNs with 60 variables, domain size 10, and density 10 (runtimes)

Unfortunately, on denser instances of randomly generated CNs, 2FC works worse than FC. Moreover, 2FC becomes worse and worse compared to FC as the underlying CN gets denser. To see this, consider the following two sets of sets of experiments (Figures 10, 11, 12, and 13). On the set of instances with density 0.2, 2FC continues to perform better in the number of nodes visited while spends more runtime. However, on the instances with density 0.8, it looks worse with respect to the both measures.

Thus, according to our experiments, 2FC performs better than FC on graph coloring problems and non-dense instances of randomly generated CN, while works worse on denser random CNs.

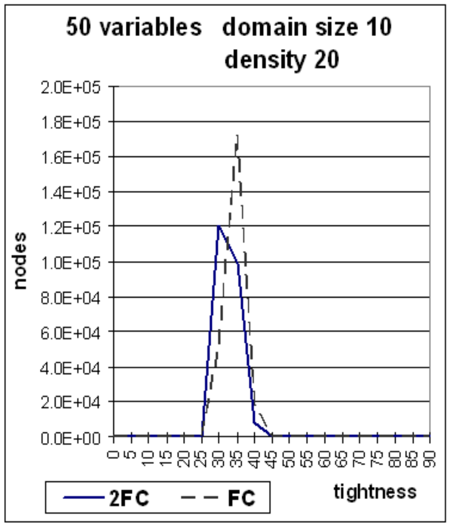


Fig. 10. 2FC vs. FC for CNs with 50 variables, domain size 10, and density 20 (nodes visited)

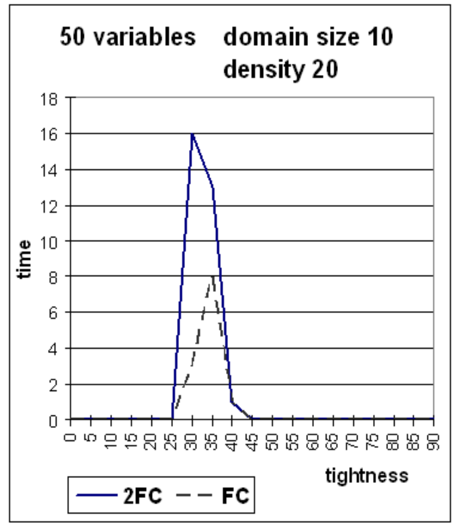


Fig. 11. 2FC vs. FC for CNs with 50 variables, domain size 10, and density 20 (runtimes)

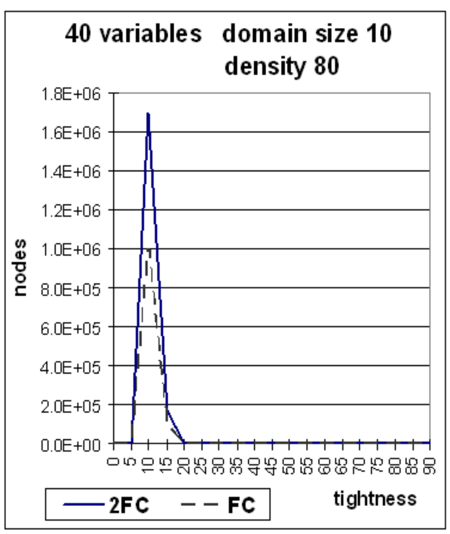


Fig. 12. 2FC vs. FC for CNs with 40 variables, domain size 10, and density 80 (nodes visited)

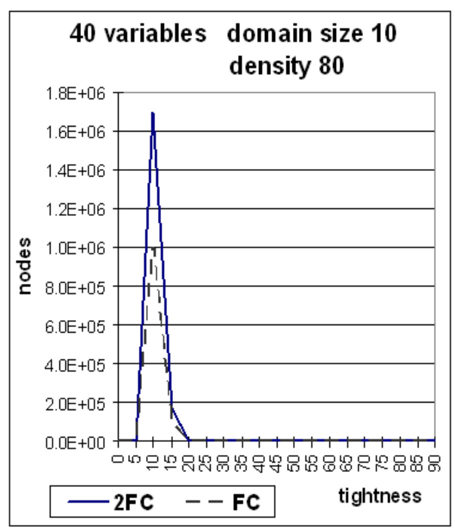


Fig. 13. 2FC vs. FC for CNs with 40 variables, domain size 10, and density 80 (runtimes)

6 Discussion

We introduced the 2FC algorithm which is based on the idea of assigning a variable with two values instead of one. In this section we discuss possible applications of the proposed approach and directions of further development.

According to our experimental results, 2FC performs very well on graph k-coloring problem. This result suggests the possibility of combining the proposed approach (of assigning a vertex with 2 colors) with branch-and-bound algorithms that find chromatic numbers of graphs (like [2]). The proposed approach could also be useful in the area of resource allocation problems because many of such problems, like timetabling [8], have binary constraint networks with inequality constraints.

On the other hand, 2FC is not very successful on random constraint networks. A natural way of improvement of its pruning ability is replacing FC by MAC, that is design of 2MAC. We expect that 2MAC would behave better with respect to MAC than 2FC does with respect to FC. This is because maintaining arc-consistency has a better ability than FC to utilize the conflicts that are added after every new assignment.

An interesting direction of further research is the application of the approach to CNs with non-binary constraints. Note that the application cannot be straightforward because solving a 2-CN with non-binary constraints is NP-complete in general (it can be shown by reduction from SAT). A possible way to recognize dead-ends early is maintaining the current extended partial solution S together with a solution T of the CN induced by S. Every time when S grows by assigning a new variable, T must grow also. Solving a 2-CN with non-binary constraints at every iteration of the algorithm could require too much time, therefore one has to develop heuristic methods of quick solving of such CNs.

2FC could also be useful in dynamic environments where solutions are frequently discarded because of updating of constraints. The set of values returned by 2FC can contain many solutions and they can be efficiently extracted. Therefore, there is a chance that once a single solution is discarded, another solution could be found instantly without applying search.

Finally, there is an intriguing connection of the proposed approach with the technique of bucket elimination [3]. In its simplest form, the principle of bucket elimination states that whenever there is an unassigned variable v conflicting with at most two other unassigned variables, variable v can be eliminated. To preserve consistency, conflicts must be added between the pairs of values that wipe out the current domain of v. Note that the described bucket elimination technique as well as assigning a variable with two values have the following common paradigm: *assign a variable v with a subset s of its domain such that every minimal consistent partial solution on the "future" variables that wipes s out has the size at most 2.* Bucket elimination is an "extremal" realization of the paradigm where a variable is assigned with the whole domain. Assigning a variable with only one value is another case of extremal realization. Then assigning a variable with two values can be considered as some "intermediate" case. Continuing the reasoning, we derive that there may be other "intermediate" realizations of the paradigm. For example, a more flexible version of bucket elimination can be considered, where a variable is assigned with a subset of its values that conflict with at most two unassigned variables.

References

1. O. Angelsmark and P. Jonsson. Improved algorithms for counting solutions in constraint satisfaction problems. In *CP 2003*, pages 81–95, 2003.
2. M. Caramia and P. Dell'Olmo. Constraint propagation in graph coloring. *Journal of Heuristics*, 8:83–107, 2002.
3. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.
4. R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.
5. D. Eppstein. Improved algorithms for 3-coloring, 3-edge coloring and constraint satisfaction. In *SODA-2001*, pages 329–337, 2001.
6. R. M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
7. P. Jeavons, D. Cohen, and M. Cooper. Constraints, consistency, and closure. *Artificial Intelligence*, 101:251–265, 1998.
8. A. Meisels and A. Schaerf. Modelling and solving employee timetabling problems. *Annals of Mathematics and Artificial Intelligence*, 39:41–59, 2003.
9. P. Prosser. Binary constraint satisfaction problems: some are harder than others. In *ECAI-94*, pages 95–99, Amsterdam, 1994.

Partitioning Based Algorithms for some Colouring Problems

Ola Angelsmark*

Department of Computer and Information Science
Linköpings Universitet, S-581 83 Sweden
olaan@ida.liu.se

Abstract. We discuss a number of different versions of the *graph colouring problem,* and give algorithms for solving them. The problems include k-COLOURABILITY, MAX IND k-COL, MAX VAL k-COL, and, finally, MAX k-COL, which is the unweighted case of the MAX k-CUT problem. The algorithms we present are based on the idea of *partitioning* the domain of the problem into disjoint subsets, and then solve a number of instances where the variables have been restricted to assume values from these partitions. Among the results we find the currently fastest polynomial space algorithm for k-COLOURABILITY.

1 Introduction

The graph colouring problem is probably one of the most well-studied graph problem. While it is conceptually easy to understand — colour the vertices of a graph such that if there is an edge between two vertices, then they must have different colours — it turns out to be *NP*-complete as soon as we allow more than two colours [15]. The problem has been studied for a long time, and it was actually the 12th problem in the list of *NP*-complete problems presented by Karp [18]. One of the reasons for studying this problem is, of course, that it regularly appears as a natural problem in such diverse areas as register allocation in compiler construction [8], and frequency assignment in mobile communication [14], just to name two.

The graph colouring problem is very nicely formulated as a constraint satisfaction problem; it is the (very) restricted CSP in which we only allow the constraint *disequality*, i.e. given two vertices of a graph, the only requirement we have is that they have different colours if there is an edge between them.

In this paper we will discuss a number of different versions of the graph colouring problem. Our results are based on an idea which was first formalised in Angelsmark & Jonsson [2]. This method, which is called the *partitioning method*, works by partitioning the domain of the problem in to a number of (disjoint)

* This research is supported in part by the Swedish National Graduate School in Computer Science (CUGS), and in part by the *Swedish Research Council* (VR), under grant 621-2002-4126.

Table 1. Comparison between our k-colouring algorithm and that of Feder & Motwani.

	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
F & M [13]	2.8277^n	3.2125^n	3.5956^n	3.9775^n	4.3581^n
New result	2.3290^n	2.7505^n	2.7505^n	3.1021^n	3.1021^n

subsets, and then solve a number of restricted instances in order to find a solution. In [2] it was used to construct an algorithm for the #CSP problem (i.e. the problem of finding the *number* of solutions to a CSP) and, in particular, it turned out to be very successful when applied to the problem of counting graph colourings. In colouring problems, where we are limited to disequality, any constraint between the variables trivially holds if they are assigned to different partitions, and we need not consider this constraint any further once the partitioning is done (see Angelsmark [1] for an in-depth discussion of this method of algorithm construction.)

The first problem we look at is the *k-colouring problem*, where the aim is to decide if it is possible to colour a given graph using at most k colours. As was mentioned earlier, this problem is *NP*-complete for $k > 2$. Interestingly enough, for $k > 6$, the fastest algorithm for this problem is the general, exponential space algorithm for CHROMATIC NUMBER; the original version by Lawler [20] has a running time of $\mathcal{O}\left((1 + \sqrt[3]{3})^n\right) \in \mathcal{O}\left(2.4423^n\right)$. This was later improved to $\mathcal{O}\left(2.4151^n\right)$ by Eppstein [12], and, recently, to $\mathcal{O}\left(2.4023^n\right)$ by Byskov [5].

The algorithm we present for this problem uses polynomial space, and while it is not faster than the CHROMATIC NUMBER algorithm, it is faster than the currently fastest polynomial space algorithm, which is due to Feder & Motwani [13]. Their algorithm has a running time of $\mathcal{O}\left(\left(\min\left(k/2, 2^{\phi_k}\right)\right)^n\right)$, where ϕ_k is given by the expression

$$\frac{1}{k+1}\sum_{i=0}^{k-1}\left(1 + \frac{i}{\binom{k}{2}}\right)\log_2(k-i).$$

Asymptotically, 2^{ϕ_k} is bounded from above by k/e, where $\mathsf{e} \approx 2.7182$ as usual. In contrast, the algorithm we propose runs $\mathcal{O}\left(\alpha_k^n\right)$, $k > 6$, where n is the number of vertices in the graph and

$$\alpha_k = \begin{cases} i - 2 + \beta_5 \text{ if } 2^i < k \leq 2^i + 2^{i-2} \\ i - 1 + \beta_3 \text{ if } 2^i + 2^{i-2} < k \leq 2^i + 2^{i-1} \\ i - 1 + \beta_4 \text{ if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

for $i \geq 3$. See Table 1 for a comparison. (Throughout the paper, we will omit polynomial factors in time and space complexities.)

MAX IND $(d, 2)$-CSP is, basically, the problem of finding a satisfiable *subinstance* of the original problem, which contains as many variables as possible (we let (d, l)-CSP denote a CSP where the domain has size at most d, and the constraints have arity l.) A subinstance is here a subset of the variables, together with the constraints which only involve these variables. (For example, if we have the variables x, y in the subset, then the constraint $R(x, y)$ would be included, but the constraint $R'(x, y, z)$ would not, since z is not in the subset.) Max Ind $(d, 2)$-CSP is, in some sense, dual to the classical Max CSP in that it does not maximise the number of satisfied *constraints*, but instead tries to maximise the number of *variables* that are assigned values without violating any constraints.

The colouring version of this problem is called the Maximum Induced k-Colourable Subgraph, or Max Ind k-COL for short. Using the partitioning method, we arrive at an algorithm which has a running time of $\mathcal{O}\left(\alpha_k^n\right)$, where

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

with $\beta_2 = 1.4460$, $\beta_3 = 1.7388$, and $i \geq 2$. The values of β_2 and β_3 are actually achieved by applying the Maximum Independent Set algorithm from Robson [21] to the microstructure of the instance (see Angelsmark & Thapper [4].)

Next, we consider the Max Value problem, which (somewhat simplified) is the problem of maximising the sum of the variable values. We first construct a specialised algorithm for the Max Value 3-Colouring problem, which runs in $\mathcal{O}\left(1.6181^n\right)$ time, and, using the partitioning method, we arrive at a running time of $\mathcal{O}\left(\alpha_k^n\right)$, where

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

with $\beta_3 = 1.6180$, and $i \geq 2$, for the general Max Value k-Colouring problem.

For our final problem, we consider the Maximum k-Colourable Subgraph, or Max k-COL, problem. This is also known as the unweighted case of the well-known Max k-CUT problem. The currently fastest algorithm for this problem is the $\mathcal{O}\left(k^{\omega n/3}\right)$ time algorithm presented in Williams [22], which utilises exponential space. Here, $\omega \in \mathbb{R}$ is the exponent in matrix multiplication over a ring, and has been shown to be less than 2.376 [9].

Using the cases $k = 2$ and $k = 3$ from [22], we can again apply the partitioning method and get an algorithm for Max k-COL which has a running time of $\mathcal{O}\left(\alpha_k^n\right)$, where

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

for $i \geq 4$. Unfortunately the underlying algorithms uses exponential space, so this will also be the case for our algorithm. However, since we only use the algorithms for $k = 2$ and $k = 3$, we will only use the space required for these. For large values of k, this is considerably less than the $\mathcal{O}\left(k^{n/3}\right)$ required by the algorithm from [22].

Additionally, this result holds equally for the *counting* problem #Max k-COL, by simply replacing the underlying algorithms with their counting counterparts (also given in [22].)

2 Preliminaries

A (d, l)-*constraint satisfaction problem* ((d, l)-CSP) is a triple (X, D, C) where

- X is a finite set of variables,
- D a finite set of domain values, with $|D| = d$, and
- C is a set of constraints $\{c_1, c_2, \ldots, c_k\}$.

Each constraint $c_i \in C$ is a structure $R(x_{i_1}, \ldots, x_{i_j})$ where $j \leq l, x_{i_1}, \ldots, x_{i_j} \in X$ and $R \subseteq D^j$. A *solution* to a CSP instances is a function $f : X \to D$ s.t. for each constraint $R(x_{i_1}, \ldots, x_{i_j}) \in C$, $(f(x_{i_1}, \ldots, f(x_{i_j}))) \in R$. Given a (d, l)-CSP, the basic computational problem is to decide whether it has a solution or not — to determine if it is *satisfiable*. If the function does not assign values to every variable, but only a subset of them, it will be referred to as a *partial* solution.

The special case $(2, 2)$-CSP is equivalent to 2-Satisfiability, or 2-SAT. An instance of 2-SAT, or a *formula*, consists of the conjunction of a set of clauses, where each clause is the disjunction of (at most) 2 literals. (A literal is either a variable or its negation.) We will be interested in *weighted* instances of 2-SAT, and define them as follows:

Definition 1 (Dahllöf *et al.* [10]). *Let Γ be a 2-SAT formula and let L be the set of all literals for all variables occurring in Γ. Given a weight vector $\mathbf{w}$, and a solution M to Γ, we define the* weight $W(M)$ of M *as*

$$W(M) = \sum_{\{l \in L \ | \ l \textit{ is true in } M\}} \mathbf{w}(l)$$

Dahllöf *et al.* [10] presents an algorithm for counting the number of maximum weighted solutions to 2-SAT instances. This algorithm has a running time of $\mathcal{O}(1.2561^n)$, and it can easily be modified to return one of the solutions. We will denote this modified algorithm *2-SAT*$_w$.

A *graph* G consists of a set $V(G)$ of *vertices*, and a set $E(G)$ of *edges*, where each element of $E(G)$ is an unordered pair of vertices. The *size* of a graph, denoted $|G|$ is the number of vertices. The *neighbourhood* of a vertex $v \in V(G)$ is the set of all vertices adjacent to v, v itself excluded, denoted $N_G(v)$; $N_G(v) = \{w \in V(G) \mid (v, w) \in E(G)\}$. The *degree* $\deg_G(v)$ of a vertex v is the size of its neighbourhood, $|N_G(v)|$. When G is obvious from the context, it can be omitted as a subscript. If we pick a subset S of the vertices of a graph together with the edges between them (but no other edges), then we get the *subgraph of G induced by S*, $G(S)$. $G(S)$ has vertex set S and edge set $\{(v, u) \mid v, u \in S, (v, u) \in E(G)\}$. If the induced subgraph has empty edge set, then it forms an *independent set.*

Definition 2 (Jégou [16]). *Given a binary CSP $\Theta = (X, D, C)$, i.e. a CSP with binary constraints, the* microstructure of Θ *is an undirected graph G defined as follows:*

1. *For each variable $x \in X$, and domain value $a \in D$, there is a vertex $x[a]$ in G.*

2. There is an edge $(x[a], y[b]) \in E(G)$ iff (a, b) violates the constraint between x and y, i.e. if xRy and $(a, b) \notin R$.

We assume there is exactly one constraint between any pair of variables; any variables without explicit constraints are assumed to be constrained by the universal constraint which allows all values.

Definition 2 is actually the complement of the graph defined as the microstructure by Jégou [16].

In the analysis of the algorithm for Max Value k-COL, we encounter a recursion on the form $T(n) = \sum_{i=1}^{k} T(n - r_i) + p(n)$, where $p(n)$ is a polynomial in n, and $r_i \in \mathbb{N}^+$. These equations satisfy $T(n) \in \mathcal{O}\left(\tau(r_1, \ldots, r_k)^n\right)$, where $\tau(r_1, \ldots, r_k)$ is the largest real-valued root to $1 - \sum_{i=1}^{k} x^{-r_i}$ (see Kullmann [19].) Note that this bound does not depend on neither $p(n)$ nor the boundary conditions $T(1) = b_1, \ldots, T(k) = b_k$. We will usually call τ the *work factor.*

3 Partitioning and Colouring Problems

We will now present the underlying ideas behind the algorithms for the colouring problems we present in this paper. We begin by defining what a partitioning is, and briefly discuss the partitioning based method for construction algorithms, before we investigate how it applies to colouring problems, i.e. CSPs were the only constraint is disequality.

Definition 3. *A* partitioning *$P = \{P_1, P_2, \ldots, P_m\}$ of a domain D is a division of D into m disjoint subsets such that $\bigcup P = D$. A k-*partition *is an element of P with k elements. Given a partitioning P, we let $\sigma(P, k)$ denote the number of k-partitions in P. Since the actual elements in the subset $P_i \in P$ is often less interesting than their number, we let the multiset $[|P_1|, \ldots, |P_m|]$ represent P.*

As an example of how the partitioning method could be used to construct an algorithm for solving CSPs, consider the following: An algorithm for solving $(4, 2)$-CSPs has a running time of $\mathcal{O}\left(1^{n_1+n_2} \cdot \alpha^{n_3} \cdot \beta^{n_4}\right)$, where n_i is the number of variables with domain size i. Thus for problems with domains of sizes 1 and 2 it is polynomial, for domain size 3 it runs in $\mathcal{O}\left(\alpha^n\right)$, and for domain size 4, it runs in $\mathcal{O}\left(\beta^n\right)$.

Using this algorithm, we want to solve, say, a $(7, 2)$-CSP. First, we split the domain of each variable into one part with 3 elements and one part with 4 elements. So if the original domain is $\{1, 2, 3, 4, 5, 6, 7\}$, we could, for example, use the partitioning $P_1 = \{1, 2, 3, 4\}$ and $P_2 = \{5, 6, 7\}$. Next, we consider each possible way of restricting the variables to only take values from one of these partitions. With n variables in the original problem, we get k variables restricted to P_1 and $n - k$ restricted to P_2, and thus get a total running time of

$$\mathcal{O}\left(\sum_{k=0}^{n} \alpha^k \cdot \beta^{n-k}\right) = \mathcal{O}\left((\alpha + \beta)^n\right).$$

When we apply this to colouring problems, we can exploit the fact that if two variables are assigned different partitions, then any constraint between them is trivially satisfied.

Let $\Theta = (X, D, C)$ and $\Theta' = (X', D', C')$ be two CSPs with the property that given solutions f to Θ and f' to Θ', they can be combined to get a solution to $\Theta_\cup = (X \cup X', D \cup D', C \cup C')$ (possibly modulo renaming of the variables and domain values.) Conversely, the two subinstances Θ and Θ' correspond to a partitioning of $\Theta_\cup$; the partitioning is $[|D|, |D'|]$, and the variables in X are mapped to D, while those in X' are mapped to D'.

We will let $Col(k, n)$ denote an arbitrary instance of a problem with domain size k and n variables having this property.

Theorem 1 (Angelsmark [1]). *Let $A_1, \ldots, A_m$ be algorithms for the problems $Col(k_1, n)$, $\ldots$, $Col(k_m, n)$, respectively, with running times in $\mathcal{O}(\alpha^n)$. Given a partitioning $P = \{P_1, \ldots, P_p\}$ of the set $\{1, \ldots, k\}$ such that for any partition P_i, $|P_i| \in \{k_1, k_2 \ldots, k_m\}$, i.e. we have algorithms for solving instances with domain size $|P_i|$, there exists a partitioning based algorithm for solving $Col(k, n)$ which has a running time of*

$$\mathcal{O}\left((|P| - 1 + \alpha)^n\right).$$

We note that the running time given by Theorem 1 is largely dependent on *the number of partitions* and less so on the running times of the algorithms for the different partitions. Consequently, in order to minimise the running times, we want to use as few partitions as possible. If we have an algorithm for $Col(k, n)$, then we can of course get an algorithm for $Col(2k, n')$ by using the partitioning $[k, k]$. The idea here is to use a *recursive* partitioning to build the $Col(k, n)$-algorithm bottom up; if we want an algorithm for $Col(4k, n)$, we first create an algorithm for domains of size $2k$ from a $Col(k, n)$ algorithm, together with the partitioning $[k, k]$, provided, of course, that we have an algorithm for $Col(k, n)$. If this is not the case, then we have to construct one by using the partitioning $[\lceil k/2 \rceil, \lfloor k/2 \rfloor]$, etc. Whether this partitioning is *optimal*, however, is still an open question.

In general, given algorithms for instances with domain sizes $k_1, \ldots, k_m$, with running times $\mathcal{O}\left(\beta_{k_i}^n\right)$, $i \in \{1, \ldots, m\}$, if it is faster to use the available algorithm for size k_i than using the partitioning $[\lceil k_{i/2} \rceil, \lfloor k_{i/2} \rfloor]$, i.e. $T([k_i]) < T([\lceil k_i/2 \rceil, \lfloor k_i/2 \rfloor])$, then there exists a partitioning based algorithm for solving for domain size k which has a running time of $\mathcal{O}\left(\alpha_k^n\right)$, where α_k is the solution to the following recurrence:

$$\alpha_k = \begin{cases} \beta_k & \text{if } k \in \{k_1, \ldots, k_m\} \\ 1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise.} \end{cases}$$

Solving this equation is straightforward, albeit tedious, thus we will omit this part of the proofs.

Table 2. The fastest polynomial space algorithms for k-colouring.

k	Time	Reference
3	$\mathcal{O}(1.3289^n)$	Eppstein [11]
4	$\mathcal{O}(1.7504^n)$	Byskov [5]
5	$\mathcal{O}(2.1020^n)$	Byskov & Eppstein [7]
6	$\mathcal{O}(2.3289^n)$	Byskov [5]

4 *k*-Colouring Algorithm

We will now show how the partitioning method applies to the problem of finding a k-colouring of a graph. This being the first of the problems we discuss, it is hoped that the discussion is sufficiently elaborated. Formally, we define the problem as follows:

Definition 4. *Let G be an arbitrary graph with vertex set $V(G)$ and edge set $E(G)$. Given a natural number k, the k-*Colouring *problem consists of finding a function $f : V(G) \to \{1, \ldots, k\}$ which assigns 'colours' to the vertices in such a way that for $v, w \in V(G)$, $f(v) = f(w)$ if $\{v, w\} \notin E(G)$, i.e. adjacent vertices are given different colours.*

Before we can apply Theorem 1, we need to show that the k-colouring problem is a $Col(k, n)$ problem. To see this, we note that if we partition the vertices of a graph G into two disjoint subsets, S_1, S_2, and colour the subgraphs induced by these sets using colours $\{1, \ldots, k_1\}$ for S_1 and $\{k_1 + 1, \ldots, k_2\}$ for S_2, getting solutions f_1 and f_2, respectively, then $f_1 \cup f_2$ must be a valid k_2-colouring of G.

Once we know that Theorem 1 is applicable, it is straightforward to get an algorithm for the problem, as is shown in Algorithm 1. The running time of the algorithm is of course the one given in Theorem 1.

Theorem 2. *Algorithm 1 correctly solves the k-colouring problem.*

Proof. Let P and α be defined as in Theorem 1. Let G be an graph, P a partitioning the domain values, and f an arbitrary total function from $V(G)$ to P — i.e. a function which assigns the vertices to partitions. Lines 3 to 6 work as follows: The vertices restricted to partition P_i induces a subgraph, and we can determine $|P_i|$-colourability of this subgraph in $\mathcal{O}(\alpha^n)$ time. Obviously, if we have two of these induced subgraphs, and we know that we can colour these two using, say, k_1 and k_2 colours, respectively, then we can colour the union of them using $k_1 + k_2$ colours. So, by induction, the variable a will, once all of the induced subgraph have been examined, be **true** iff the graph is k-colourable. Repeating this for all total functions ensures that we will examine all possible restrictions of vertices to partitions. □

Algorithm 1 only determines the existence of a k-colouring, but it is of course straightforward to adapt it to return an explicit colouring.

Algorithm 1 Partitioning based k-COLOURING algorithm.

k-*COL* (G, P)

for each total function $f : V(G) \to P$ **do**
  $a :=$ **true**
  for each $P_i \in P$ **do**
    $G' := G|\{v \in V(G) \mid f(v) = P_i\}$
    $a := a \wedge A_{|P_i|}(G')$
  end for
  if a **then**
    return "yes"
end for
return "no"

In the literature, we find a number of different algorithms for determining k-colourability of a graph. Table 2 contains the currently fastest polynomial space algorithms for $k \leq 6$. For $k > 6$, the most efficient polynomial space algorithm for k-colouring is the $\mathcal{O}\left((k/c_k)^n\right)$ time algorithm by Feder & Motwani [13]. We will not get an improvement over the bounds in Table 2 from the partitioning method, but we do get a way of constructing algorithms for *any* $k > 6$, which is faster than $\mathcal{O}\left((k/c_k)^n\right)$.

As we noted earlier, the *number* of partitions has a large impact on the running time of the algorithm. For example, if we want an algorithm for, say, 8-colouring, it is tempting to use the partitioning $[2, 2, 2, 2]$, since 2-colouring is polynomial. This, however, gives a running time of $\mathcal{O}\left(4^n\right)$, while if we use the partitioning $[4, 4]$, we get a running time of $\mathcal{O}\left(((2-1) + 1.7504)^n\right) = \mathcal{O}\left(2.7504^n\right)$, which is an enormous improvement. Using the partitioning $[3, 3]$ we get a 6-colouring algorithm with the same running time as in [5], $\mathcal{O}\left((2 - 1 + 1.3289)^n\right) = \mathcal{O}\left(2.3289^n\right)$.

Now let β_i^n, $i \in \{3, 4, 5\}$ denote the running times of the 3-, 4- and 5-colouring algorithms in Table 2.

Theorem 3. *If we can solve* 3-, 4-, 5-COLOURING *in* $\mathcal{O}\left(\beta_i^n\right)$ *time, for* $i = 3, 4, 5$*, respectively, then there exists a partitioning based algorithm for solving* k-COLOURING*,* $k > 6$*, which has a running time of* $\mathcal{O}\left(\alpha_k^n\right)$*, where*

$$\alpha_k = \begin{cases} i - 2 + \beta_5 \text{ if } 2^i < k \leq 2^i + 2^{i-2} \\ i - 1 + \beta_3 \text{ if } 2^i + 2^{i-2} < k \leq 2^i + 2^{i-1} \\ i - 1 + \beta_4 \text{ if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

for $i \geq 3$.

Proof. Using the partitioning $[\lfloor k/2 \rfloor, \lceil k/2 \rceil]$ recursively, together with the colouring algorithms above, we get from Theorem 2 that a partitioning based algorithm will have a running time of $\mathcal{O}\left(\alpha_k^n\right)$, where α_k is given by the solution to the following recurrence:

$$\alpha_k = \begin{cases} \beta_k & \text{if } k \in \{3, 4, 5\} \\ 1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise} \end{cases}$$

Solving the equation gives the result. □

So for example, if we wish to determine 14-colourability of a graph, we start with the pre-existing algorithms for 3- and 4- colourability, and let them be parts of a 7-colourability algorithm which uses the partitioning $[3, 4]$. From this we get an algorithm for 14-colourability which works with the partitioning $[7, 7]$. The running time, given by Theorem 3, is then $\mathcal{O}((2+\beta_4)^n) \approx \mathcal{O}(3.7504^n)$

5 Max Ind *k*-Colouring Algorithm

The general Max Ind $(d, 2)$-CSP is defined as follows:

Definition 5 (Jonsson & Liberatore [17]). *Let $\Theta = (X, D, C)$ be an instance of (d, l)-CSP. The* Max Ind *(d, l)-CSP problem consists of finding a maximal subset $X' \subseteq X$ such that $\Theta|X'$ is satisfiable.*

Here, $\Theta|X' = (X', D, C')$ is the *subinstance of Θ induced by X'*, i.e. the CSP we get when we restrict Θ to the variables in X' and the constraints which involve only variables from X', *viz.*

$$C' := \{c \in C \mid c(x_1, x_2, \ldots, x_l) \in C, x_1, \ldots, x_l \in X'\}.$$

When we restrict this problem to colourings, we get the Max Ind k-COL problem, which is the problem of assigning colours to as many vertices as possible without having neighbours of the same colours. Unlike the k-colouring problem, not every vertex is necessarily assigned a colour.

Definition 6. *Given a graph G and a natural number k, the* Max Ind k-COL *problem is to find a subset $S \subseteq V(G)$ such that the induced subgraph $G(S)$ is k-colourable and $|S|$ is maximised.*

The problem is still *NP*-complete even under this restriction (see Jonsson & Liberatore [17]). Theorem 1 is of course applicable to this problem, and it has been shown that Max Ind 2-COL and Max Ind 3-COL can be solved in $\mathcal{O}(1.2025^{2n}) = \mathcal{O}(1.4460^n)$ and $\mathcal{O}(1.2025^{3n}) = \mathcal{O}(1.7388^n)$ time, respectively [4], thus we can combine this with the following theorem to get an algorithm for Max Ind k-COL.

Theorem 4. *Given that we can solve* Max Ind 2-COL *and* Max Ind 3-COL *in time $\mathcal{O}(\beta_2^n)$ and $\mathcal{O}(\beta_3^n)$, respectively, there exists a partitioning based algorithm for solving* Max Ind k-COL *which has a running time of $\mathcal{O}(\alpha_k^n)$, where*

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

for $i \geq 4$.

Proof. We use the partitioning $[\lfloor k/2 \rfloor, \lceil k/2 \rceil]$ recursively, and we get from Theorem 1 that a partitioning based algorithm will have a running time of $\mathcal{O}(\alpha_k^n)$, where α_k is given by the following recurrence:

$$\alpha_k = \begin{cases} \beta_2 & \text{if } k = 2 \\ \beta_3 & \text{if } k = 3 \\ 1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise} \end{cases}$$

Solving the equation gives the result. □

6 Max Value k-Colouring

The MAX VALUE problem for CSPs has been studied in, e.g. Angelsmark *et al.* [3], and is formally defined as follows:

Definition 7 (Angelsmark *et al.* [3]). *Let $\Theta = (X, D, C)$ be an instance of (d, l)-CSP, where $D = \{a_1, a_2, \ldots, a_d\} \subseteq \mathbb{R}$, $X = \{x_1, x_2, \ldots, x_n\}$. Given a real vector $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$, the* MAX VALUE *problem for Θ consists in finding a solution $f : X \to D$ which maximises*

$$\sum_{i=1}^{n} w_i \cdot f(x_i)$$

The colouring version of the MAX VALUE problem, the MAX VALUE k-COL problem, is defined as follows:

Definition 8. *Given a graph G, with $|V(G)| = n$, a real vector of weights $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$ and a natural number k, the* MAX VALUE k-COL *problem consists in finding a function $f : V(G) \to \{1, \ldots, k\}$, with $f(v) \neq f(w)$ if $(v, w) \in E(G)$, such that*

$$\sum_{v \in V(G)} w_v \cdot f(v)$$

is maximised.

For clarity, we let $(\Theta, \mathbf{w})$, where $\Theta = (X, D, C)$, be an instance of MAX VALUE 2-COL. Now let G be the microstructure graph of Θ, and, for $x \in X$, let $\eta(x)$ be the number of constraints x is involved in (in the microstructure, this corresponds to $\deg(x[i]) - 1$.)

Theorem 5. *There exists an algorithm for solving the* MAX VALUE 2-COL *problem which runs in polynomial time.*

Proof. We show that Algorithm 2 correctly solves the MAX VALUE 2-COL problem.

Algorithm 2 Algorithm for MAX VALUE 2-COL

MaxVal 2-COL $(\Theta, \mathbf{w})$

1. Let G be the microstructure of Θ.
2. $m := 0$
3. **if** G is 2-colourable **then**
4. Let $f : V(G) \to \{1, 2\}$ be a 2-colouring of G
5. **for each** connected component c of G **do**
6. $m := m+$

$$\max_{v \in c} \left(\sum_{f(v)=1} \mathbf{w}(v), \sum_{f(v)=2} \mathbf{w}(v) \right)$$

7. **end for**
8. **end if**
9. **return** m

First of all, we note that if the microstructure graph is *not* 2-colourable, then the MAX VALUE 2-COL instance has the trivial solution 0, since there are no colourings, and this is what the algorithm returns.

Next we observe that *if* a 2-colouring exists, then for each of the connected component in G, there are exactly two possible colourings. Consequently, since we can choose the colour with largest weight for each connected component in isolation, when the algorithm reaches line 9, m will contain the weight of the maximum solution. □

In order to successfully apply the partitioning method here, we need to take care of odd-sized colourings, and thus we need an algorithm for the MAX VALUE 3-COL problem.

First, some additional definitions: A variable having three possible domain values we call a 3-variable, and a variable having two possible values will be called a 2-variable. The size of an instance is defined as $m(\Theta) = n_2 + 2n_3$, where n_i denotes the number of i-variables in Θ. Consequently, the size of an instance can be decreased by either fully remove 2-variables or removing one of the possible values for a 3-variable, and turn it into a 2-variable.

Given a variable x with three possible values, $\{d_1, d_2, d_3\}$, ordered in such a way that $w(x, d_1) > w(x, d_2) > w(x, d_3)$, let $\delta(x) := (c_1, c_2, c_3)$ where $c_i = \deg_G(x[d_i])$, G being the microstructure graph. If x is a 2-variable then, similarly, we define $\delta(v) := (c_1, c_2)$. The *maximal* weight of a variable x, i.e. the domain value d for which $w(x, d)$ is maximal, will be denoted $x_{\max}$.

Since the only allowed constraint is disequality, it is never the case that a 3-variable can have two unconstrained values — for example, if $x[d_1]$ had an edge to $y[d_1]$, but $x[d_2]$ and $x[d_3]$ had no edges to y, this would mean that vertices $y[d_2]$ and $y[d_3]$ had already been removed, and thus we could propagate $y[d_1]$, the only possible value for y.

Algorithm 3 Algorithm for MAX VALUE 3-COL.

MaxVal 3-COL $(G, \mathbf{w})$

1. **if** at any time, the domain of a variable becomes empty,
   that branch can be pruned.
2. Apply Lemma 2, keeping track of eliminated variables.
3. **if** applicable return the maximum of the branches described in Lemma 1
4. **else**
5. Let Γ_w be the weighted 2-SAT instance corresponding to G.
6. **return** *2-SAT*$_w(\Gamma_w)$
7. **endif**

Lemma 1. *If there is a variable x with $\delta(x) = (\geq 3, \cdot, \cdot)$, we can reduce the instance with a work factor of $\tau(4, 2)$.*

Proof. If $x_{\max}$ is chosen, then we remove x together with its two external neighbours, thus reducing the size of the instance by (at least) 4. The only reason *not* to choose $x_{\max}$ is, of course, that one of its external neighbours was chosen, thus reducing the size of the instance by 2. □

After applying the reduction in this lemma, it is holds that *no* variable x has $x_{\max}$ with degree greater than 2. This means that either the neighbours of $x_{\max}$ are the other possible values of x, and thus x is unconstrained, or one of the other values has been eliminated, and there are only two possible values for x. We can apply the following lemma to get rid of all of the cases of unconstrained variables, and what we have left is an instance of weighted 2-SAT.

Lemma 2 (Angelsmark & Thapper [4]). *For any instance Θ, we can find an instance Θ' with the same optimal solution as Θ, with size smaller or equal to that of Θ and to which* none *of the following cases apply.*

1. *There is a 2-variable x for which $\delta(x) = (2, \geq 1)$.*
2. *There is a variable x for which the maximal weight is unconstrained.*

Thus, we get the following theorem:

Theorem 6. MAX VALUE 3-COL *can be solved by a deterministic algorithm in time $\mathcal{O}(1.6181^n)$.*

Proof. Algorithm 3 has, apart from the call to *2-SAT*$_w$, a work factor of $\tau(4, 2) \leq 1.2721$. Since we used $m(\Theta) = n_2 + 2n_3$ as the measure of size, the size of an instance is $2n$. Consequently, the algorithm has a running time of

$$\mathcal{O}\left((\max(1.2721, 1.2561))^{2n}\right)$$

i.e. $\mathcal{O}(1.6181^n)$. □

Since we are only considering colourings, we can apply Theorem 1 and get:

Theorem 7. *If we can solve* Max Value 2-COL *in polynomial time, and* Max Value 3-COL *in* $\mathcal{O}(\beta_3^n)$ *time, respectively, then there exists a partitioning based algorithm for solving* Max Value k-COL *which has a running time of* $\mathcal{O}(\alpha_k^n)$*, where*

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

and $i \geq 4$.

Proof. Again, we recursively use the partitioning $[\lfloor k/2 \rfloor, \lceil k/2 \rceil]$, and use Theorem 1 to get an algorithm which will have a running time of $\mathcal{O}(\alpha_k^n)$, where α_k is given by the solution to the following recurrence:

$$\alpha_k = \begin{cases} 1 & \text{if } k = 2 \\ \beta_3 & \text{if } k = 3 \\ 1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise} \end{cases}$$

Solving the equation gives the result. □

7 Max k-COL and #Max k-COL Algorithms

Max CSP is probably one of the most widely studied optimisation problems for CSPs. Williams [22] presents an impressive algorithm for this problem, the first to run in provably less than $\mathcal{O}(d^n)$ time, as well as *counting* version of this problem, i.e. the problem of finding *how many* solutions there are, is usually denoted #Max CSP. Formally, we define the problem as follows:

Definition 9. *Given an instance* $\Theta = (X, D, C)$ *of* $(d, 2)$*-CSP, the* Max $(d, 2)$*-CSP problem is to find an assignment* $f : X \to D$ *which satisfies the maximum number of constraints.*

If we restrict Max CSP to colouring problems, we get the Maximum k-Colourable Subgraph problem, or Max k-COL — also known as the unweighted case of the Max k-CUT problem. Note the difference to the Max Ind k-COL problem; there, we were dealing with an *induced* subgraph.

Definition 10. *Given a graph* G *and a natural number* k*, the* Max k-COL *problem is to find a subset* E' *of* $E(G)$ *such that the graph* $(V(G), E')$ *is* k*-colourable and* $|E'|$ *maximised. The problem of determining the* number *of such subsets is denoted* #Max k-COL*.*

Williams [22] shows that Max k-COL can be solved in $\mathcal{O}\left(k^{\omega n/3}\right)$ time, where $\omega < 2.376$, but we can improve this bound using the partitioning method:

Theorem 8. *Given that we can solve* Max 2-COL *and* Max 3-COL *(*#Max 2-COL *and* #Max 3-COL*) in time* $\mathcal{O}(\beta_2^n)$ *and* $\mathcal{O}(\beta_3^n)$*, respectively, there exists* *a partitioning based algorithm for solving* Max k-COL *(#*Max k-COL*) which has a running time of* $\mathcal{O}(\alpha_k^n)$*, where*

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \textit{if } 2^i < k \le 2^i + 2^{i-1} \\ i + \beta_2 & \textit{if } 2^i + 2^{i-1} < k \le 2^{i+1} \end{cases}$$

for $i \geq 4$*. Furthermore, the space requirement is equal to that of the most demanding of the given algorithms.*

Proof. Again, we use the partitioning $[\lfloor k/2 \rfloor, \lceil k/2 \rceil]$ recursively. From Theorem 1, we know that a partitioning based algorithm will have a running time of $\mathcal{O}(\alpha_k^n)$, where α_k is given by the solution to the following recurrence:

$$\alpha_k = \begin{cases} \beta_2 & \text{if } k = 2 \\ \beta_3 & \text{if } k = 3 \\ 1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise} \end{cases}$$

Solving the equation gives the time complexity stated in the theorem.

As for the space complexity, the algorithms for Max 2-COL and Max 3-COL are applied in sequence, and thus the space requirement remains unchanged. ☐

Combining this theorem with the algorithms for Max 2-COL (#Max 2-COL) and Max 3-COL (#Max 3-COL) with running times of $\mathcal{O}(1.7315^n)$ and $\mathcal{O}(2.3872^n)$, respectively, utilising $\mathcal{O}\left(2^{n/3}\right)$ and $\mathcal{O}\left(3^{n/3}\right)$ space, gives an algorithm for the general Max k-COL (#Max k-COL) problem.

8 Acknowledgements

The author wishes to thank Peter Jonsson and Johan Thapper for interesting and fruitful discussions during the writing of this paper.

References

1. Ola Angelsmark. *Constructing Algorithms for Constraint Satisfaction and Related Problems.* PhD thesis, Department of Computer and Information Science, Linköpings Universitet, Sweden, 2005.
2. Ola Angelsmark and Peter Jonsson. Improved algorithms for counting solutions in constraint satisfaction problems. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming, 9th International Conference (CP-2003), Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 81–95. Springer–Verlag, 2003.
3. Ola Angelsmark, Peter Jonsson, and Johan Thapper. Two methods for constructing new CSP algorithms from old. Unpublished manuscript, 2004.
4. Ola Angelsmark and Johan Thapper. A microstructure based approach to constraint satisfaction optimisation problems. In Ingrid Russell and Zdravko Markov, editors, *Recent Advances in Artificial Intelligience. Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005), 15-17 May, 2005, Clearwater Beach, Florida, USA*, pages 155–160. AAAI Press, 2005.

5. Jesper Makholm Byskov. Enumerating maximal independent sets with applications to graph colouring. *Operations Research Letters*, 32(6):547–556, November 2004.
6. Jesper Makholm Byskov. *Exact Algorithms for Graph Colouring and Exact Satisfiability*. PhD thesis, Basic Research In Computer Science (BRICS), Department of Computer Science, University of Aarhus, Denmark, August 2004.
7. Jesper Makholm Byskov and David Eppstein. An algorithm for enumerating maximal bipartite subgraphs. Unpublished manuscript (see also [6]), 2004.
8. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
9. Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.
10. Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1–3):265–291, February 2005.
11. David Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2001), January 7-9, 2001, Washington, DC, USA*, pages 329–337. ACM/SIAM, 2001.
12. David Eppstein. Small maximal independent sets and faster exact graph coloring. *Journal of Graph Algorithms and Applications*, 7(2):131–140, 2003.
13. Tomás Feder and Rajeev Motwani. Worst-case time bounds for coloring and satisfiability problems. *Journal of Algorithms*, 45(2):192–201, November 2002.
14. Andreas Gamst. Some lower bounds for a class of frequency assignment problems. *IEEE Transactions on Vehicular Technology*, 35(1):8–14, 1986.
15. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
16. Philippe Jégou. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In *Proceedings of the 11th (US) National Conference on Artificial Intelligence (AAAI-93)*, pages 731–736, Washington DC, USA, July 1993. American Association for Artificial Intelligence (AAAI).
17. Peter Jonsson and Paolo Liberatore. On the complexity of finding satisfiable subinstances in constraint satisfaction. Technical Report TR99-038, Electronic Colloquium on Computational Complexity, 1999.
18. Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
19. Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1–2):1–72, 1999.
20. Eugene L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5(3):66–67, August 1976.
21. Mike Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical report, LaBRI, Université Bordeaux I, 2001.
22. Ryan Williams. A new algorithm for optimal constraint satisfaction and its implications. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sanella, editors, *Automata, Languages and Programming: 31st International Colloquium (ICALP-2004), July 12-16, 2004, Turku, Finland. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 1227–1237. Springer–Verlag, 2004.

A Type System for CHR

Emmanuel Coquery[1,2] and François Fages[1]

[1] INRIA Rocquencourt, Projet Contraintes,
BP 105, F-78153 Le Chesnay, France
[2] Conservatoire National des Arts et Métiers,
292 rue Saint Martin, 75141 Paris Cedex 03, France

Abstract. We propose a generic type system for the *Constraint Handling Rules* (CHR), a rewriting rule language for implementing constraint solvers. CHR being a high-level extension of a host language, such as Prolog or Java, this type system is parameterized by the type system of the host language. We show the consistency of the type system for CHR w.r.t. its operational semantics. We also study the case where the host language is a constraint logic programming language, typed with the prescriptive type system we developed in our previous work. In particular, we show the consistency of the resulting type system w.r.t. the extended execution model CLP+CHR. This system is implemented through an extension of our type checker TCLP for constraint logic languages. We report on experimental results about the type-checking of 12 CHR solvers and programs, including TCLP itself.

1 Introduction

The language of *Constraint Handling Rules* (CHR) of T. Frühwirth [1] is a successful rule-based language for implementing constraint solvers in a wide variety of domains. It is an extension of a host language, such as Prolog [2] or Java [3], allowing the introduction of new constraints in a declarative way. CHR is used to handle user-defined constraints while the host language deals with other computations using *native* constraints. CHR is a commited-choice language of guarded rules that rewrite constraints into simpler ones until they are in solved forms. One peculiarity of CHR is that it allows multiple heads in rules.

Typed languages have numerous advantages from the point of view of program development, such as the static detection of programming errors or program composition errors, and the documentation of the code by types. CHR has already been used for the typing of programming languages, either for solving subtyping constraints [4, 5] or for handling overloading in functional languages [6] and constraint logic languages [7, 5]. However, to our knowledge, there is no type system for CHR.

In this article, we propose a type system for CHR inspired by the TCLP type system for constraint logic programs [8]. CHR being an extension of a host language, this system is parameterized by the type system of the host language. We will make three assumptions on the type system of the host language:

- Typing judgments of the form $\Gamma \vdash t : \tau$ are considered, where τ is a type associated to the term t in a typing environment Γ. Moreover *well-typed* constraints in a typing environment Γ are defined by a derivation system for typing judgments.
- The constraint $t_1 = t_2$ is well-typed in the environment Γ if there exists a type τ such as $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$.
- If a conjunct c of native constraints is well-typed in an environment Γ and is equivalent to a conjunct d, then d is also well-typed in Γ.

Using these assumptions, we show the consistency of the type system for CHR w.r.t. its operational semantics. This is expressed by a subject reduction theorem which establishes that if a program is well-typed then all the derived goals from a well-typed goal are well-typed.

We also study the instantiation of this type system with the TCLP type system for constraint logic programs [8]. We show a subject reduction theorem for the CLP+CHR execution model [1] in which it is possible to extend the definition of constraints by clauses. This result is interesting because constraint logic programming is a natural framework for using constraint solvers. A type system for CLP+CHR allows us to type-check a solver together with the program that uses it, as well as complex CHR solvers written as a combination of clauses and rules, where CHR rules use CLP predicates and CLP clauses post CHR constraints.

The rest of the paper is organized as follows. Section 2 recalls the syntax and operational semantics of CHR, including the CLP+CHR execution model. Section 3 presents the type system and section 4 presents its instantiation with the type system for CLP. Section 5 presents some experimental results on the typing of some CHR solvers, using the implementation of the system in TCLP [9]. Finally, we conclude in section 6.

2 Preliminaries on CHR

Here, we recall the syntax and semantics of CHR, as given in [1]. We distinguish the user-defined *CHR constraints* from the *native constraints* of the host language, which represent auxiliary computations that take place during the application of a CHR rule. We assume that native constraints are handled by a predefined solver of the host language. We also assume that native constraints include the equality constraint $=/2$ and the constraint *true*. The terms of the host language are noted $s, t, \ldots$ We note $\mathcal{X}$ the domain of native constraints, and $\mathcal{CT}$ its (possibly incomplete) first-order logic theory.

2.1 Syntax

Definition 1. *A CHR rule is either:*

- *a* simplification *rule of the form:*
  $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$

- *a* propagation *rule of the form:*
  $H_1, \ldots, H_i$ `==>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
- *or a* simpagation *rule of the form:*
  $H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$

with $i > 0$, $j \geq 0$, $k \geq 0$, $l > 0$ and $H_1, \ldots, H_i$ is a nonempty sequence of CHR constraints, the guard $G_1, \ldots, G_j$ being a sequence of native constraints and the body $B_1, \ldots, B_k$ being a sequence of CHR and native constraints.

A CHR program is a finite sequence of CHR rules.

The constraint *true* is used to represent empty sequences. The empty guard can be omitted, together with the | symbol. The notation *name*@R gives a name to a CHR rule R.

Informally, a simplification rule replaces the constraints of the head by the constraints of the body. A propagation rule adds the constraints of the body while keeping the constraints of the head in the store. A simpagation rule is a mix of the two preceding kind of rules: the constraints $H_{l+1}, \ldots, H_i$ are replaced by the body, while the constraints $H_1, \ldots, H_l$ are kept.

For the sake of simplicity, and because the distinction of propagation and simpagation rules are not needed for typing purposes, we will consider that a propagation or a simpagation rule of the form

$$H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i \ \texttt{<=>}\ G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

is just an abbreviation for the simplification rule

$$H_1, \ldots, H_i \ \texttt{<=>}\ G_1, \ldots, G_j \mid H_1, \ldots, H_l, B_1, \ldots, B_k.$$

Example 1. The following CHR program, taken from [1], defines a solver for a general ordering constraint `=<`.

```
reflexivity  @ X=<Y <=> X=Y | true.
antisymetry  @ X=<Y , Y=<X <=> X=Y.
transitivity @ X=<Y , Y=<Z ==> X=<Z.
identity     @ X=<Y \ X=<Y <=> true.
```

The rule `reflexivity` eliminates the `=<` constraints when its two arguments are equal. Rule `antisymmetry` simplifies a double inequality into an equality. The rule `transitivity` adds constraints corresponding to the transitive closure of `=<`. Finally, `identity` eliminates redundant `=<` constraints.

2.2 Operational Semantics

The operational semantics of CHR is expressed by a transition system, noted $\longmapsto$, over states which are triples $\langle F, E, D \rangle$, where F is a goal, that is a multiset of native and CHR constraints, E is a CHR constraint store and D is a native constraint store. A state is thus a conjunction of CHR and native constraints.

In the following definition, the equality is extended to constraints by morphism, that is $c(t_1, \ldots, t_n) = c(t'_1, \ldots, t'_n)$ if $t_1 = t'_1 \wedge \ldots \wedge t_n = t'n$. The conjunction notation $\wedge$ is used to express the matching of a constraint in a multiset, The equality is also extended to conjunctions of constraints: $H_1 \wedge \ldots \wedge H_n = H'_1 \wedge \ldots \wedge H'_n$ if $H_1 = H'_1 \wedge \ldots \wedge H_n = H'_n$.

Definition 2. *Let P be a CHR program. The transition relation $\longmapsto$ is given by the following rules, where the variables appearing in triples stand for conjunctions of constraints and $\bar{x}$ represents the set of variables appearing in the head H.*

Solve
$\langle C \wedge F, E, D\rangle \longmapsto \langle F, E, D'\rangle$
if C is a native constraint and $\mathcal{CT} \models (C \wedge D) \Leftrightarrow D'$.
Introduce
$\langle H \wedge F, E, D\rangle \longmapsto \langle F, H \wedge E, D\rangle$
if H is a CHR constraint.
Simplify
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
if (H `<=>` $G \mid B$) is in P renamed with fresh variables,
and $\mathcal{CT} \models D \Rightarrow \exists\bar{x}(H = H' \wedge G)$.
Propagate
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D\rangle$
if (H `==>` $G \mid B$) is in P renamed with fresh variables,
and $\mathcal{CT} \models D \Rightarrow \exists\bar{x}(H = H' \wedge G)$.

The **Solve** transition corresponds to a transition of the native constraint solver. The **Introduce** transition simply transfers a CHR constraint from the goal to the CHR constraint store The **Simplify** transition correspond to the application of CHR simplification. The **Propagate** transition is indicated for the sake of clarity, although it is treated as an abbreviation for a simplification rule in the rest of the paper. The condition for applying these rules is that the head of the rule can be instantiated such that the guard and the matching condition of the head are implied by the current native constraint store. The body of the rule is then added to the current goal and, when applying a **Simplify** transition, the constraints matching the head are removed from the constraint store.

Definition 3. *An* initial state *consists in a goal F and two empty constraint stores: $\langle F, true, true\rangle$. A* final state *is either of the form $\langle F, E, false\rangle$ (*failure*), or of the form $\langle true, E, D\rangle$ where D is satisfiable (*success*).*

The following example illustrates the execution of a CHR program.

Example 2. Let us consider the solver given in example 1 together with the initial state $\langle$`X=<Y` $\wedge$ `Y=<Z` $\wedge$ `Z=<X`, $true$, $true\rangle$. One possible execution is:

$\langle$`Z=<X`, `X=<Y` $\wedge$ `Y=<Z`, $true\rangle$ (**Introduce** $\times 2$)
$\langle$`X=<Z` $\wedge$ `Z=<X`, `X=<Y` $\wedge$ `Y=<Z`, $true\rangle$ (**Propagate** `transitivity`)
$\langle true$, `X=<Z` $\wedge$ `Z=<X` $\wedge$ `X=<Y` $\wedge$ `Y=<Z`, $true\rangle$ (**Introduce** $\times 2$)
$\langle$`X=Z`, `X=<Y` $\wedge$ `Y=<Z`, $true\rangle$ (**Simplify** `antisymmetry`)
$\langle true$, `X=<Y` $\wedge$ `Y=<Z`, `X=Z`$\rangle$ (**Solve**)
$\langle$`X=Y`, $true$, `X=Z`$\rangle$ (**Simplify** `antisymmetry`)
$\langle true, true$, `X=Y` $\wedge$ `X=Z`$\rangle$ (**Solve**)

One can remark that in this operational semantics, once a propagation rule can be applied, it can be applied infinitely often, which leads to a trivial case of non termination. In the preceding example, one could have applied the `transitivity` rule instead of the `antisymmetry` rule, thus reintroducing the constraint `X=<Z` that was eliminated at the fourth step. In [10], Abdennadher gives refined operational semantics that are more faithful to the actual implementation of CHR. In particular the previous behaviour is avoided by restricting the application of a rule only once on the same constraints. The subject reduction theorems given in the following sections express that given a well-typed program, a transition occurring from a well-typed state leads to a well-typed state. It is worth noting that they thus hold also in these more realistic semantics.

2.3 CLP+CHR

When the host language belongs is a constraint logic programming language $CLP(\mathcal{X})$ [11], it is possible to tightly integrate CHR to the host language. To this end, Frühwirth [1] proposed to extend CHR with the construct *label_with* used to define CHR constraints by CLP clauses. We recall here the syntax and operational semantics of this extension. We note $\mathcal{S}_F$ (resp. $\mathcal{S}_P$) the set of function (resp. predicate) symbols, given with their arity, and $\mathcal{V}$ the set of variables. An *atom* is either a native constraint, a CHR constraint or of the form $p(t_1, \ldots, t_n)$, where p/n is a program predicate symbol. The declaration *label_with* $c(t_1, \ldots, t_n)$ *if* $G_1, \ldots, G_j$ expresses that $G_1, \ldots, G_j$ is a guard for the clauses of the CHR constraint c/n.

Definition 4. *A* labeling declaration *for a CHR constraint H is an expression of the form:*

label_with H *if* $G_1, \ldots, G_j$

where $G_1 \ldots, G_j$ is a conjunction of native constraints.
Clauses are of the form:

H :- $B_1, \ldots, B_n$

where H an atom corresponding either to a predicate or to a CHR constraint but not to a native constraint, and $B_1, \ldots, B_k$ is a sequence of atoms.

Definition 5. *The relation transition between CHR states is extended by the two following rules:*

Unfold
$\langle H' \wedge F, E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
if $(H$ `:-` $B)$ *is in* P *renamed with fresh variables, and* H *is not a CHR constraint.*

Label
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
if $(H$ `:-` $B)$ *and (label_with* H'' *if* G*) are in* P *renamed with fresh variables, and* $\mathcal{X} \models \exists \bar{x}(H' = H'' \wedge G)$

The **Unfold** transition is close to the CSLD resolution rule [11]. The difference is that, under CSLD resolution, the constraints in the body of the resolving clause are added to the native constraint store and the resulting store, i.e. $H = H' \wedge D \wedge C$, must be satisfiable,which is not demanded here. The CLP clauses for CHR constraints can only be used in a **Label** transition, requiring that the guards declared using *label_with* are implied by the current native constraint store.

3 Type System

3.1 Assumptions about the type system of the host language

Since CHR is an extension of a host language. The type system we propose is parameterized by the type system, noted $\vdash_N$, of the host language, We will make the following assumptions on $\vdash_N$.

We suppose that $\vdash_N$ is based on a type algebra, the set of types being noted $\mathcal{T}$. Types are noted using the letter τ. Typing environments, noted Γ, associate types to variables. Given an expression t and a typing environment Γ, $\vdash_N$ is used to deduce typing judgments of the form $\Gamma \vdash_N t : \tau$. Similarly, $\vdash_N$ is used to deduce *well-typed* constraints in a typing environment Γ, a conjunction $C_1 \wedge \ldots \wedge C_n$ of native constraints being well-typed in Γ if for each $i \in \{1, \ldots, n\}$, C_i is well-typed in Γ. We note $\Gamma \vdash_N C\ Atom$, the fact that C is well-typed in the typing environment Γ. We also assume that the equality constraint $s = t$ between s and t is well-typed in Γ if there exists a type τ such that $\Gamma \vdash_N s : \tau$ and $\Gamma \vdash_N t : \tau$.

We assume that the union of type environments over disjoint sets of variables can be formed with an operation noted $\uplus$ such that if $\Gamma \vdash_N t : \tau$ then $\Gamma \uplus \Gamma' \vdash_N t : \tau$ for any typing environment Γ' disjoint from Γ. We also assume that if a conjunction of native constraints C is well-typed in a typing environment Γ and $\mathcal{CT} \models C \Leftrightarrow D$, then there exists a typing environment Γ', such that the conjunction of constraints D is well-typed in $\Gamma \uplus \Gamma'$.

3.2 Type System for CHR

The type system we propose for CHR defines a notion of well-typedness for CHR rules. To each CHR constraint symbol c/n is associated one, and only one, type of the form $\tau_1 \times \ldots \times \tau_n$. This type is assumed to be fixed, for example using some declarations provided by the programmer. One can remark that the limitation to one type for each constraint symbol prevents the use of type schemes, such as those used for parametric polymorphism [12]. For example, it is forbidden to give the type $\forall\alpha.list(\alpha) \times list(\alpha) \times list(\alpha))$ to a CHR constraint `append/3` , because it is equivalent to give it all the types of the form $list(\tau) \times list(\tau) \times list(\tau)$. This restriction is motivated by the example 3 given below, which shows that the allowance of type schemes in CHR would make the type system inconsistent.

(*Native*)	$\dfrac{\Gamma \vdash_N C\ Atom}{\Gamma \vdash C\ Atom}$	if C is a native constraint
(*CHR Atom*)	$\dfrac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ Atom}$	if c/n a CHR constraint and if c/n has type $\tau_1 \times \ldots \times \tau_n$
(*Goal*)	$\dfrac{\Gamma \vdash B_1\ Atom \quad \ldots \quad \Gamma \vdash B_n\ Atom}{\Gamma \vdash B_1, \ldots, B_n\ Goal}$	
(*Simpl CHR*)	$\dfrac{\Gamma \vdash H_1, \ldots, H_i\ Goal \quad \Gamma \vdash G_1, \ldots, G_j\ Goal \quad \Gamma \vdash B_1, \ldots, B_k\ Goal}{\vdash H_1, \ldots, H_i \texttt{ <=> } G_1, \ldots, G_j \mid B_1, \ldots, B_k\ Rule}$	

Table 1. Type system for CHR

The rules of the type system for CHR are given in table 1. A CHR constraint H is *well-typed* in Γ if the judgment $\Gamma \vdash H\ Atom$ can be derived from the typing rule. Terms or expressions appearing as arguments of the constraints are typed using the type system $\vdash_N$ for the host language. The rule (*Simpl CHR*) expresses that a CHR rule is *well-typed* if its head, its guard and its body are well-typed in the same typing environment Γ. A CHR program is well-typed if all its clauses are well-typed.

The following lemma expresses that the well-typedness of goals is preserved by extension of the environment:

Lemma 1. *Let G be a goal and Γ a typing environment such that $\Gamma \vdash G\ Goal$. Let Γ' be a typing environment such that $\Gamma \uplus \Gamma'$ is defined. Then $\Gamma \uplus \Gamma' \vdash G\ Goal$.*

Proof. By induction on the derivation and by using the assumption that if $\Gamma \vdash_N t : \tau$ then $\Gamma \uplus \Gamma' \vdash_N t : \tau$ for any expression t.

The consistency of the type system w.r.t. the operational semantics of CHR is given by the following subject reduction theorem, which expresses that the well-typedness of goals is preserved by transitions:

Theorem 1. *Let P be a well-typed CHR program. Let $\langle F, E, D\rangle$ and $\langle F', E', D'\rangle$ be two states such that $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$. If there exists a typing environment Γ such that $\Gamma \vdash F, E, D\ Goal$, then there exists a typing environment Γ' such that $\Gamma' \vdash F', E', D'\ Goal$. Moreover, if the transition rule contains a guard G then $\Gamma' \vdash G\ Goal$.*

Proof. By case on the transition.

Solve By hypothesis, $\Gamma \vdash D\ Goal$ and $\Gamma \vdash C\ Atom$. Since $\mathcal{CT} \models (C \wedge D) \Leftrightarrow D'$, and by assumption on $\vdash_N$, there exists a typing environment Γ'' such that $\Gamma \uplus \Gamma'' \vdash_N D'\ Goal$. By posing $\Gamma' = \Gamma \uplus \Gamma''$ and by lemma 1, we obtain $\Gamma' \vdash F, E\ Goal$, thus $\Gamma' \vdash F, E, D'\ Goal$.

Introduce This transition only moves a constraint from the goal to the CHR constraint store, thus the resulting state is also well-typed in Γ.

Simplify By hypothesis, the rule (H `<=>` $G \mid B$) is well-typed. Thus, there exists a typing environment Γ'' such that $\Gamma'' \vdash H, G, B\ Goal$. By lemma 1, and by posing $\Gamma' = \Gamma \uplus \Gamma''$, we obtain $\Gamma' \vdash F, E, D\ Goal$ and $\Gamma' \vdash G, B\ Goal$. It remains to prove that $H = H'$ is well-typed in Γ'. H is of the form $c_1(t^1_1, \ldots, t^1_{m_1}), \ldots, c_i(t^i_1, \ldots, t^i_{m_i})$ and H' of the form $c_1(s^1_1, \ldots, s^1_{m_1}), \ldots, c_i(s^i_1, \ldots, s^i_{m_i})$. Let $l \in \{1, \ldots, i\}$ and $p \in \{1, \ldots, m_l\}$. Let $\tau_1 \times \ldots \times \tau_{m_l}$ be the type of c_l. Since $\Gamma' \vdash c_l(t^1_l, \ldots, t^l_{m_l})\ Atom$, we have $\Gamma' \vdash_N t^l_p : \tau_p$. Similarly, $\Gamma' \vdash_N s^l_p : \tau_p$. Thus $\Gamma' \vdash t^l_p = s^l_p\ Atom$. Thus, we obtain $\Gamma' \vdash B, F, E, H = H', D\ Goal$ and $\Gamma' \vdash H = H', G\ Goal$.

The following example illustrates how the use of a type scheme for a CHR constraint symbol may lead to check an ill-typed conjunction of constraints.

Example 3. Let us assume that the constraint `=<` has the type scheme $\forall\alpha.\alpha \times \alpha$. Let us suppose that `"a"` and `"b"` have type *string*, that `1` and `2` have type *int*, and that these two types are incompatible. Let us consider the rule:

```
transitivity @ X=<Y, Y=<Z ==> X=<Z.
```

and the state ⟨`true`, `"a"=<"b"` $\wedge$ `1=<2`, *true*⟩. This state is well-typed. However if one tries to apply the rule `transitivity` with **Propagate**, one needs to check that the constraint store implies `X="a"` $\wedge$ `Y="b"` $\wedge$ `Y=1` $\wedge$ `Z=2`, which is not a well-typed constraint, because `Y` must have both types *string* and *int*.

4 Integration with CLP

In this section we are interested in the particular case where the host language is a constraint logic language, typed using the prescriptive type system TCLP [8]. This system combines parametric polymorphism, subtyping and overloading to obtain the flexibility that is needed for typing CLP programs that are originally untyped. In particular, subtyping is used for typing the simultaneous use of different constraint domains: for instance, the relation $boolean \leq int$ allows one to see booleans as integers, and thus to type check constraints combining boolean variables with integer variables (such as in a sum of boolean variables). Subtyping is also used for the typing of programs using meta-programming techniques: the relation $list(\alpha) \leq term$ allows one to see homogeneous lists as terms and to apply decomposition predicates to them, such as `functor/3`, `arg/3` or `=../2`.

In [8], the type system of TCLP is proved consistent w.r.t. the CSLD execution model [11], which is an abstract model of execution proceeding by constraint accumulation. In particular, the transformations that can be made by the constraint solver are not considered. In the following, we assume that the solver for native constraints only performs simplifications that preserve well-typedness, according to the assumptions of section 3.1. This can be obtain either by using a typed execution model, as proposed in [8], or, in the case of the equality constraint, by using modes to fix the dataflow [13].

First, we present the type algebra used in the system, then we recall the typing rules for CLP, together with a typing rule for the labeling declaration *label_with*. The resulting system is proven consistent w.r.t. the CLP+CHR execution model.

4.1 Type Structure

We consider a partial order $(\mathcal{K}, \leq_{\mathcal{K}})$ of *type constructors*, given with their arity. The set $\mathcal{T}$ of types is the set of finite or infinite types built on $\mathcal{K}$.

Subtyping Relation The use of subtyping for meta-programming purposes requires to consider relations like $list(\alpha) \leq term$. This form of non-structural non-homogeneous subtyping links different constructors of different arities. Such subtyping relations require to express the correspondence between the different arguments of type constructors. For example, by writing $k_1(\alpha, \beta) \leq k_2(\beta)$, we specify that types built with k_1 are subtypes of those built with k_2, provided that the second argument of k_1 is a subtype of the argument of k_2, the first argument of k_1 being forgotten in the subtyping relation. One way to express the correspondence is to use a formalism of labels, as proposed by Pottier [14]. In this formalism, a label is associated to each argument of type constructors, the correspondence being expressed by the fact that two arguments of type type constructors have the same label. The subtyping order $\leq$ is built from the order $\leq_{\mathcal{K}}$ on type constructors and from the labels. A formal description of the type structure is given in [4], where the structures of types and type constructors are quasi-lattices, i.e. partial orders in which two elements have a least upper (resp. greatest lower) bound if and only if they have an upper (resp. lower) bound.

Subtyping constraints Let $\mathcal{W}$ be a set of *type variables*, or *parameters*, noted $\alpha, \beta, \ldots$. We note $\mathcal{T}_{\mathcal{W}}$ the set of types built on $\mathcal{K} \cup \mathcal{W}$.

Definition 6. *A* subtyping constraint *is of the form $\tau_1 \leq \tau_2$, where $\tau_1, \tau_2 \in \mathcal{T}_{\mathcal{W}}$ are finite types. A substitution $\rho : \mathcal{W} \to \mathcal{T}$* satisfies *the constraint $\tau_1 \leq \tau_2$, noted $\rho \models \tau_1 \leq \tau_2$, if $\rho(\tau_1) \leq \rho(\tau_2)$. The subtyping constraint $\tau_1 \leq \tau_2$ is* satisfiable *if there exists a substitution ρ such that $\rho \models \tau_1 \leq \tau_2$.*

In [4], sufficient conditions on $(\mathcal{K}, \leq_{\mathcal{K}})$ are given for the decidability of the satisfiability of subtyping constraints in quasi-lattices, this problem is shown to NP-complete, and a practical algorithm (used in section 5) is given for computing explicit solutions.

4.2 Type system for CLP+CHR

In order to support the overloading of CLP function and predicate symbols, we assume that a set *types*(f/n) of type schemes of the form $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \tau$ is associated to each function symbol f/n (resp. predicate symbol p/n), where $\bar{\alpha}$ is the set of parameters occurring in types $\tau_1, \ldots, \tau_n, \tau$. These sets of types are supposed to be fixed, for example using declarations provided by the programmer. We also assume that the type of the constraint $= /2$ is the type scheme $\forall\alpha.\alpha \times \alpha$. For the sake of simplicity, the quantification $\forall\bar{\alpha}$ will be omitted in type schemes, each occurrence of a type scheme being renamed with fresh parameters. For a CHR constraint c/n of type $\tau_1 \times \ldots \times \tau_n$, *types*$(c/n)$ is defined as the singleton $\{\tau_1 \times \ldots \times \tau_n\}$.

A typing environment is a partial mapping $\Gamma : \mathcal{V} \mapsto \mathcal{T}_{\mathcal{W}}$, also noted $\{X_1 : \tau_1, \ldots, X_n : \tau_n\}$. The operation $\uplus$ on typing environments is defined as disjoint union, that is $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_1(X)$ if $X \in dom(\Gamma_1)$, $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_2(X)$ if $X \in dom(\Gamma_2)$, and $(\Gamma_1 \uplus \Gamma_2)(X)$ is undefined otherwise.

$$(Var)\ \frac{X : \tau \in \Gamma}{\Gamma \vdash X : \tau} \qquad (Sub)\ \frac{\Gamma \vdash t : \tau \quad \tau \leq \tau'}{\Gamma \vdash t : \tau'}$$

$$(Func)\ \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau\rho} \quad \begin{array}{r}\rho \text{ is a type substitution}\\ \tau_1 \times \ldots \tau_n \to \tau \in \mathit{types}(f/n)\end{array}$$

$$(Atom)\ \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash p(t_1, \ldots, t_n)\ Atom} \quad \begin{array}{r}\rho \text{ is a type substitution}\\ \tau_1 \times \ldots \tau_n \in \mathit{types}(p/n)\end{array}$$

$$(Head)\ \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash p(t_1, \ldots, t_n)\ Head_{\tau_1 \times \ldots \times \tau_n}} \quad \begin{array}{r}\rho \text{ is a type renaming}\\ \tau_1 \times \ldots \tau_n \in \mathit{types}(p/n)\end{array}$$

$$(Clause)\ \frac{\forall\sigma \in \mathit{types}(p/n) \quad \begin{array}{c}\Gamma_\sigma \vdash p(t_1, \ldots, t_n)\ Head_\sigma\\ \Gamma_\sigma \vdash B_1\ Atom \quad \ldots \quad \Gamma_\sigma \vdash B_k\ Atom\end{array}}{\vdash p(t_1, \ldots, t_n)\ \texttt{:-}\ B_1, \ldots, B_k\ Clause}$$

$$(Label\ with)\ \frac{\Gamma \vdash H\ Atom \quad \Gamma \vdash G\ Goal}{\vdash label\_with\ H\ if\ G\ Label\_with}$$

Table 2. Type system for CLP and *label_with*

Table 2 gives the typing rules for CLP, together with the typing rule for the declaration *label_with*. The typing rules for CLP resemble the rules of Mycroft and O'Keefe [12] with the addition of subtyping and overloading. A predicate call $p(t_1, \ldots, t_n)$ (resp. a native constraint) is *well-typed* in a typing environment Γ if $\Gamma \vdash p(t_1, \ldots, t_n)\ Atom$ can be derived from the rules. A clause H `:-` B is *well-typed* if $\vdash H\ \texttt{:-}\ B\ Clause$ can be derived from the rules. A labeling declaration *label_with H if G* is well-typed if $\vdash$ *label_with H if G Label_with* can be derived. The (*Sub*) rule gives the semantics of subtyping by expressing that if a term t has type τ, then it has all types that are greater than τ.

The set of rules of tables 1 and 2 define the type system for CLP+CHR. A CLP+CHR program is well-typed if all its CHR rules, all its clauses and all its labeling declarations are well-typed. The rule (*CHR Atom*) can be seen has a particular case of rules (*Atom*) and (*Head*), where the symbol p/n has only one type scheme without any parameter, which corresponds to the restriction that a CHR constraint has only one type.

The distinction between rules (*Atom*) and (*Head*) expresses the principle of *definitional genericity* [15], which establishes that the type of the head of a clause must be equivalent to, up to renaming but not an instance of, the declared type of the predicate. The rule (*Clause*) imposes that a clause must be well-typed for all possible types of the defined predicate, which can be seen as a condition similar to definitional genericity for overloading. These two conditions are useful for the following subject reduction theorem which expresses the consistency of the type system with reference to the operational semantics of CLP+CHR. It is preceded by a lemma which expresses that in a derivation apart from (*Head*) or (*Clause*), the types can be arbitrarily instantiated.

Lemma 2. *For any typing environment Γ, for any judgment R different from Head or Clause and any type substitution ρ, if $\Gamma \vdash R$ then $\Gamma\rho \vdash R\rho$.*

Proof. By induction on the derivation.

Theorem 2. *Let us consider a well-typed CHR+CLP program. Let $\langle F, E, D\rangle$ and $\langle F', E', D'\rangle$ be two states and Γ be a typing environment such that $\Gamma \vdash F, E, D$ Goal. If $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$, then there exists a typing environment Γ' such that $\Gamma' \vdash F', E', D'$ Goal. Moreover, if the transition rules requires a guard G then $\Gamma' \vdash G$ Goal.*

Proof. One can check that the assumptions of section 3.1 are correct for the the system of table 2. Moreover, an atom corresponding to a predicate call and an atom corresponding to a native constraint are typed in the same way. Therefore, by theorem 1, if the transition is one of **Solve**, **Introduce** or **Simplify**, then there exists a typing environment te' such that $\Gamma' \vdash F', E', D'$ *Goal* and $\Gamma' \vdash G$ *Goal* in case of need.

Let us consider the **Unfold** transition. We can assume, without loss of generality, that $H' = p(s)$ and $H = p(t)$. Since $\Gamma \vdash p(s)$ *Atom*, there exists a type scheme $\tau \in types(p)$ and a substitution ρ such that $\Gamma \vdash s : \tau\rho$. Since the program is well-typed, $\vdash H$ `:-` B *Clause*, thus there exists a typing environment Γ_τ such that $\Gamma_\tau \vdash B$ *Goal* and $\Gamma_\tau \vdash p(t)$ $Head_\tau$, that is $\Gamma_\tau \vdash t : \tau\rho_r$ where ρ_r is a renaming of τ. By posing $\rho' = \rho_r^{-1}\rho$, and by lemma 2, we obtain $\Gamma_\tau\rho' \vdash t : \tau\rho$ and $\Gamma_\tau\rho' \vdash B$ *Goal*. By posing $\Gamma' = \Gamma_\tau\rho' \uplus \Gamma$, we obtain $\Gamma' \vdash t = s$ *Atom*. Thus $\Gamma' \vdash B, F, E, s = t, D$ *Goal*.

Let us finally consider the case of a **Label** transition. Similarly to the case of the **Unfold** transition, there exists a typing environment Γ', such that $\Gamma' \vdash$

$B, F, E, t = s, D$. Since H' is a CHR constraint, τ does not contain any parameter, that is ρ'' is the identity substitution. We have $H'' = p(u)$ for some term u. Since $\vdash$ *label_with* H'' *if* G *Label_with*, there exists a typing environment Γ_{lw} such that $\Gamma_{lw} \vdash G\ Goal$ and $\Gamma_{lw} \vdash H''\ Atom$, that is $\Gamma_{lw} \vdash u : \tau$. By posing $\Gamma'' = \Gamma' \uplus \Gamma_{lw}$, we obtain $\Gamma'' \vdash s = u\ Atom$, $\Gamma'' \vdash B, F, E, s = t, D$ and $\Gamma'' \vdash G\ Goal$, and thus $\Gamma'' \vdash s = u, G\ Goal$.

5 Experimental Results

The type system for CLP+CHR has been implemented as an extension of the TCLP software [9], which is a type checker for constraint logic programming. Furthermore a type inference algorithms makes it possible to infer types for variables and for program predicates automatically. In a lattice of types with top element *term* however, the type $term \times \ldots \times term$ is always a possible type for predicates. For this reason, a heuristic type inference algorithm is used, providing a more informative type and often the expected type [8, 5]. This algorithm can also be used to infer the type of CHR constraints that are not declared by the user.

TCLP uses several solvers written in CHR. The main solver is the one for subtyping constraints. We also use a CHR solver to handle overloading of function and predicate symbols during type checking. Some other small CHR solvers are also used for handling typing environments and preliminary computations on the structure of type constructors. Hence, the possibility to type check CHR programs makes it possible that TCLP type checks its own source code.

The following example shows the typical kind of errors detected by TCLP:

Example 4. The following solver handles counters. The constraint `cpt/2` associates the name of the counter to its value, and has type $atom \times int$.[3] The constraint `val/2` also has type $atom \times int$ and constraints `incr/1` and `init/1` have type *atom*.

```
init(C) <=> cpt(C,0).
cpt(C,V) \ val(C,X) <=> X=V.
incr(C), cpt(V,C) <=> V1 is V+1, cpt(C,V1).
```

The type checker produces the following message:

```
! Error in "count.pl", line 3 :
  Incompatible types for C : atom and int
```

It is in fact an argument inversion: in the head of the last rule, the arguments of the constraint `cpt` were inverted.

The following example shows the result of type inference on a small solver:

[3] The type *atom* corresponds to Prolog atoms, that is symbols of arity 0, and not to the logical atoms.

Example 5. The following solver, taken from [16], computes the greatest common divisor of two numbers.

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=>
   N=<M | L is M mod N, gcd(L).
```

The type checker infers the following type:

```
:- typeof gcd(int) is chr_constraint.
```

that is `gcd` has type *int*.

Performances The speed of the type checker has been evaluated on ten CHR solvers taken from [16], on the solver for subtyping constraints, on the solver for overloading in TCLP, as well as on the complete TCLP source code. These tests were run on a 2 Ghz Pentium IV with 512 Mo of RAM, using the Sicstus Prolog implementation of TCLP for which the working memory space is limited to 256 Mo. The results are presented in table 3.

			Type check		Type inference	
Program	# lines	# rules	CHR	Total	CHR	Total
gcd	10	2	0.03 s	0.03 s	0.04 s	0.04 s
varleq	30	4	0.04 s	0.26 s	0.07 s	0.43 s
bool	173	78	1.32 s	2.13 s	4.63 s	5.96 s
listdom	73	13	0.78 s	1.45 s	1.77 s	2.75 s
interval	145	24	3.41 s	3.5 s	99.58 s	99.69 s
domain	266	84	4.30 s	6.42 s	183.92 s	186.94 s
fourier-gauss	328	30	1.98 s	5.88 s	19.04 s	30.42 s
arc	47	2	0.14 s	0.81 s	0.23 s	1.09 s
allenComp	495	490	17.48 s	17.51 s	NA	NA
subtyping	595	57	4.52 s	6.22 s	319.66 s	322.64 s
overloading	465	10	0.43 s	3.99 s	1.10 s	8.01 s
TCLP	4594	82	5.22 s	53.97 s	416.08 s	518.39 s

Table 3. Performances

The first column indicates the CLP+CHR program. The second column indicates the number of lines of codes in the program and the third one indicates the number of CHR rules in the program. Next, in column “Type checking”, the type checking times are given with type inference for variables, but without type inference for predicates or CHR constraints. Finally, the column “Type inference” indicates the times for inferring types to predicates and constraints. The typing times for CHR rules are given in columns “CHR”, while the typing times for the whole CLP+CHR programs are given in the column “Total”.

The type checking times without type inference for predicates and constraints show that the type checker is usable in practice. For example, it takes less than

18 s to check the 490 rules of the `allenComp` solver, or less than 54 s to check about 4600 lines of code constituting the source of TCLP.

In presence of subtyping, type inference needs 71 times more CPU time than type checking. In the case of `allenComp`, type inference even fails by lack of memory due to the restriction to 256 Mo. This is due to the fact that, when inferring the type of a constraint, the type checker must consider at the same time all the rules and clauses in a same connected component of the call graph, while type checking can be done rule by rule. CHR solvers often use large connected components however. One reason for this difficulty is that a few constraints used as data structures, appear in the head of numerous rules, thus creating large connected components. For example, the solver for subtyping constraints has a connected component of 54 predicates and CHR constraints. Such connected components thus require to deal with a very large number of subtyping constraints and overloaded symbols at once. Moreover, algorithms for solving subtyping constraints and overloading are potentially exponential [7, 4]. From this point of view, the performances of type inference are quite satisfactory. Type inference can be used the first time a solver is written, the inferred types being used afterwards as declarations during the rest of the development of the solver.

6 Conclusion

We have presented a type system for the *Constraint Handling Rules* CHR language [1], parameterized by the type system of the host language. In the particular case of constraint logic programming, its combination with the prescriptive type system TCLP [8] for CLP languages has been presented. Under the assumption that the well-typedness of native constraints is preserved by logical equivalence, the type system has been proved consistent w.r.t. the operational semantics of CHR and CLP+CHR respectively.

The type system for CLP+CHR is implemented as an extension of the TCLP software [9]. The reported experimental results on ten CHR solvers plus TCLP itself show that the system is already usable and useful.

As for future work, we plan to enhance the performances of type inference with large connected components, and to get some practical experience from the users of the system, in particular for the development of complex modular [17] and/or collaborative CHR solvers. It would also be interesting to study the instantiation of the type system with the one of Java in the framework of the JACK toolkit implementation of CHR [3].

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
2. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules compiler and runtime system. Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules **14** (2000)

3. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java Constraint Kit. In: Electronic Notes in Theoretical Computer Science. Volume 64. Elsevier (2000)
4. Coquery, E., Fages, F.: Subtyping constraints in quasi-lattices. In Pandya, P., Radhakrishnan, J., eds.: Proceedings of the 23rd conference on foundations of software technology and theoretical computer science, FSTTCS'2003. Lecture Notes in Computer Science, Mumbai, India, Springer-Verlag (2003)
5. Coquery, E.: Typage et programmation en logique avec contraintes. PhD thesis, Université Paris 6 - Pierre et Marie Curie (2004)
6. Stuckey, P.J., Sulzmann, M.: A theory of overloading. In Peyton-Jones, S., ed.: Proceedings of the International Conference on Functional Programming, ACM Press (2002) 167–178
7. Coquery, E., Fages, F.: Tclp: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical report, INRIA Rocquencourt (2002)
8. Fages, F., Coquery, E.: Typing constraint logic programs. Journal of Theory and Practice of Logic Programming **1** (2001) 751–777
9. Coquery, E.: (TCLP) `http//contraintes.inria.fr/~coquery/tclp/`.
10. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming. Volume 1330 of Lecture Notes in Computer Science., Linz, Springer-Verlag (1997) 252–266
11. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany, ACM (1987) 111–119
12. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial Intelligence **23** (1984) 295–307
13. Smaus, J.G., Fages, F., Deransart, P.: Using modes to ensure subject reduction for typed logic programs with subtyping. In: Proceedings of FSTTCS '2000. Number 1974 in Lecture Notes in Computer Science, Springer-Verlag (2000)
14. Pottier, F.: A versatile constraint-based type inference system. Nordic Journal of Computing **7** (2000) 312–347
15. Lakshman, T., Reddy, U.: Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Saraswat, V., Ueda, K., eds.: Proceedings of the 1991 International Symposium on Logic Programming, MIT Press (1991) 202–217
16. Frühwirth, T., Schrijvers, T.: (CHR web page) `http//www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
17. Haemmerlé, R., Fages, F.: Closures are needed for closed module systems. Technical report, INRIA (2005)

Systematic Explanation of Indexical Constraints

Ludovic Langevine*

SICS – Uppsala Science Park, SE-75183 Uppsala, Sweden
langevin@sics.se

Abstract. Several constraint solvers over finite domains, such as GNU-Prolog or SICStus Prolog, use indexicals to define their primitive constraints. An indexical specifies a set of consistent values for a given variable and can be compiled into an efficient filtering algorithm.
The explanation of a value withdrawal is a set of constraints that is sufficient to justify this withdrawal. Explanations enable the handling of over-constrained or dynamic problems, and allow the development of new search methods.
Traditionally, an explanation algorithm has to be designed for each filtering algorithm. This paper shows that explanation algorithms can be automatically derived from indexical constraints. In the case of SICStus Prolog, explanation computation can also benefit from the existing implementation.

1 Introduction

Explanations are a variety of techniques that keep track of the decisions of the constraint solver, mainly to enhance the search-procedure. Explanations have been proved very effective to speed up the resolution of some classes of problems, for instance highly disjunctive scheduling problems [11]. They also allow dynamic and over-constrained problems to be handled within a single tool. Moreover, Ferrand et al. [9] have shown that explanations could be used to debug some incorrect programs.

Despite their various uses and advantages, explanations are not widely used yet. Few solvers provide this facility, as noticed by Douence and Jussien [6]. Mechanisms of explanation computation are intrusive: instrumentations of the solver code is needed. The main barrier seems to be the development cost in the solver core itself, but also in the filtering algorithms of the primitive constraints. Solver developers are hardly enthusiastic about modifying their highly optimized code. Douence and Jussien address this issue with an aspect-oriented approach that avoids the direct modification of the solver's source code. But, in any case, the algorithm that computes explanations of the behavior of a given constraint has to be designed and developed. For basic constraints, it is an easy but tedious task. The design of the explaining algorithm is more difficult when dealing with more complex constraints. Moreover, users have to design the explanation algorithms of the constraints they define so as to fully benefit from this facility.

Some solvers use dedicated languages to define their primitive constraints. Those definitions are compiled into a general-purpose language or interpreted at runtime. Indexicals, introduced by Van Hentenryck et al. [17], are a popular language to describe

* This work has been carried out during the tenure of an ERCIM fellowship.

primitive constraints. Two remarkable implementations of indexicals are the CLP platforms GNU Prolog [5] and SICStus Prolog [2]. Indexicals are both declarative and operational: they specify the consistency level to enforce as well as providing a simple mechanism to achieve it.

As far as we know, no implementation of indexicals provides explanations. In this paper, we firstly present a scheme to compute trivial explanations of indexical constraints. Then, we propose to use the structure of the indexical language to compute more accurate elimination and failure explanations. This scheme is systematic: it avoids the design of dedicated explanation algorithms. Moreover, we show how to benefit from an existing indexical implementation to limit the extra-computation. The computed explanations emphasize the role of domain bounds: they are not inclusion-minimal but are accurate for a large subset of the indexical language.

As a motivation for our work, Section 2 reminds the reader of the guiding principles of explanations and their main applications. Section 3 presents the indexical constraints that we want to explain and the classical ways they are implemented. We expose a computation scheme that trivially explains indexical constraints in Sect. 4. A more accurate computation scheme, that emphasizes the role of the bounds of the domains, is described in Sect. 5. Section 6 discusses related work. Section 7 concludes.

2 Explanations in CP(FD)

Explanations are reasons why certain solver actions occur. An explanation is a subset of the constraints of the problem to solve that is sufficient to justify a decision. Two kinds of explanations have been widely studied: value withdrawal explanation and failure explanation. The first one justifies why a given value cannot be taken by a given variable, and is removed from its domain. The second one is a conflict set that makes the problem unsatisfiable. This Section firstly defines the explanations (Sect. 2.1). Section 2.2 details the properties of the explanations we consider in the remaining of the paper. Lastly, as a motivation of our work, we briefly describe the main applications of explanations within constraint programming (Sect. 2.3).

2.1 Formal Definition

A *constraint satisfaction problem* (CSP) is defined by a set of variables $\mathcal{V}$, a domain function $\mathcal{D}$ that assigns to each variable v a finite set of possible values $\mathcal{D}(v)$, and a set of constraints $\mathcal{C}$. We deal with finite domain variables: $\mathbb{D}$ denotes the finite set of possible values for all the variables in $\mathcal{V}$. To solve such a system, the search-procedure adds new "decision" constraints that can be retracted later. The initial domains of the variables, when different from $\mathbb{D}$, are included in $\mathcal{C}$ as unary constraints: $v \in D_v^0$ where $D_v^0 \subset \mathbb{D}$ is the initial domain of variable v.

In this paper, we use the formalism of Ferrand et al. [8]: the propagation is modeled by the computation of the downward closure of the set of local consistency operators associated to $\mathcal{C}$. This formalism captures the incompleteness due to local consistency. As a simplification, we denote by $CL{\downarrow}(\mathcal{V}, \mathcal{C})$ the domain function on $\mathcal{V}$ resulting from the propagation of the constraints in $\mathcal{C}$.

Definition 1 (Elimination Explanation). *An* elimination explanation *of the value d for the variable v is a set of constraint $E \subseteq \mathcal{C}$ such that:*

$$d \notin CL{\downarrow}(\mathcal{V}, E)(v)$$

Definition 2 (Failure Explanation). *A* failure explanation *of $(\mathcal{V}, \mathcal{C})$ is a set of constraints $E \subseteq \mathcal{C}$ that admits no solution, that is to say:*

$$\exists v \in \mathcal{V} \cdot CL{\downarrow}(\mathcal{V}, E)(v) = \emptyset.$$

In the following, each time we indiscriminately deal with *elimination* or *failure* explanation, we write "explanation of a solver decision".

2.2 Characterization of Explanations

We denote by $\mathcal{E}$ the set of already computed explanations. $\mathcal{E}$ is a partial function from $\mathcal{V} \times \mathbb{D}$ to $\mathcal{P}(\mathcal{C})$, where $\mathcal{P}(\mathcal{C})$ is the power set of $\mathcal{C}$. We use the relational notation: $\mathcal{E} \in \mathcal{V} \times \mathcal{D} \times \mathcal{P}(\mathcal{C})$.

Property 1 (Removal-Defined Explanations). $\mathcal{E}$ is a partial function whose range of definition is exactly the set of already achieved value-removals:

$$\forall v \in \mathcal{V} \cdot \quad (\ \exists (v, d, E) \in \mathcal{E} \Leftrightarrow d \in \mathbb{D} - \mathcal{D}(v)\ )$$

When explanations are computed on the fly, during the propagation process, the elimination explanation of d from $\mathcal{D}(v)$ contains the very constraint that eliminated d. It also contains the constraints that have removed every support of d and the explanation of every support elimination. We use the notion of *constructive explanations* to formalize this:

Property 2 (Constructive Explanations). Each explanation in $E \in \mathcal{E}$ is built by a constraint c on top of a (possibly empty) set of previous explanations. Formally, each (v, d, E) in $\mathcal{E}$ is such that: $E = \{c\} \cup \bigcup_{i=1}^{n} E_i$. Moreover, each E_i involves variables in $\mathrm{var}(c)$ ($\mathrm{var}(c)$ is the set of variables involved in c, comes from the locality of the computation): $\forall i, 1 \leq i \leq n, \exists v_i \in \mathrm{var}(c), \exists d_i \in \mathbb{D} \cdot (v_i, d_i, E_i) \in \mathcal{E}$.

For a given solver decision, we can generally find several explanations. Firstly, if E is an explanation of a solver decision, then any $E' \subseteq \mathcal{C}$, such that $E \subset E'$, is also an explanation of this decision, as a corollary of the monotonicity of reduction operators. Secondly, several *different* explanations of the same decision can coexist from then on there exist two explanations $E_1 \subseteq \mathcal{C}$ and $E_2 \subseteq \mathcal{C}$ such that $E_1 \cap E_2$ is not an explanation of this decision. Ågren [1] defines the *sharpest* explanation of a decision as the explanation that involves a minimal number of variables. A more widely-used criterion is the inclusion-minimality of an explanation [12]. In this paper, we construct explanations on top of already computed explanations and add the constraint that triggers off the decision. Therefore, we consider the so-called *local inclusion-minimality* [12]. Roughly speaking, an explanation E is locally inclusion-minimal if it is built on top a minimal quantity of previously existing explanations E_i.

2.3 Motivating Applications of Explanations

We are concerned by four main applications of explanations, namely debugging, solving of over-constrained problems, solving of dynamic problems, and improvement of the search procedure. In this section, we sum up those four points. This overview aims at motivating the inclusion of explanations in a constraint solver, but it is not intended to prove the interest of explanations. We refer the reader to the cited work for quantitative or technical details.

Firstly, explanations help correct a wrong program. When a specific solution is expected but is not found by the program, we can conclude that a set of constraints forbids this solution. One or several of these constraints may be incorrect. An explanation-based solver can exhibit an accurate conflict-set for a given expected solution. The programmer can then focus his attention on those constraints and find the cause of the bug, as shown by Ferrand et al. [9].

Secondly, the same guiding principle as above allows to deal with over-constrained problems. When a problem is correctly stated and implemented but unsolvable, some constraints have to be relaxed. Ouis et al. [16] show how explanations help the user choose constraints to relax. Their tool, COINS, can interact with the final-user thanks to a user-friendly display of explanations. The user needs no knowledge of constraint programming to understand the exhibited conflict and to relax constraints.

Thirdly, explanations are suitable to handle dynamic problems, where the set of constraints to be taken into account can change during the execution. Some constraints can be relaxed while other constraints have to be added to the original problem. Explanations enable the solver to relax any constraint while undoing all its past effects, with a minimal amount of computation. Furthermore, the state of the domains after the relaxation is often close to their former state. For instance, Elkhyari et al. [7] handle various dynamic scheduling problems where activities and resources can be added or retracted and whose precedence constraints can be altered. Their results for large instances show both the efficiency of the method, and a high stability of the schedules proposed by the tool.

Lastly, explanations liberate the solver from the *chronological* backtracking search. Since the solver can relax any already posted constraint, the search procedure is not necessarily a tree-traversal: any decision constraint can be relax at any time. On each failure, a conflict set is retrieved. All the collected conflict sets can be processed to adapt the heuristic dynamically. Here, the interest of explanations is twofold. They are used to learn from the failures (to choose the right decision constraint to undo) and then, to perform a dynamic backtracking. For instance, Jussien et al. [14] proposed the MAC-DBT technique.

3 Indexical Constraints

This Section recalls the basics of indexicals: their expression power (Sect. 3.1), their compilation and propagation scheme (Sect. 3.2). Sect. 3.3 presents some technical aspects of their implementations. For further details, we refer to Carlsson et al. [3] for SICStus Prolog and Codognet and Diaz [4] for GNU Prolog.

```
plus(X,Y,Z) +:                                          plus(X,Y,Z) +:
  X in min(Z)-max(Y) .. max(Z)-min(Y),                    X in dom(Z) - dom(Y),
  Y in min(Z)-max(X) .. max(Z)-min(X),                    Y in dom(Z) - dom(X),
  Z in min(X)+min(Y) .. max(X)+max(Y).                    Z in dom(X) + dom(Y).
```

Fig. 1. Specification of two consistency levels for $X + Y = Z$ (SICStus Prolog syntax)

3.1 Expression Power of Indexicals

Indexicals are constraints of the form X `in` R, where X is a variable and R is an expression representing a set of values. The expression R may use information about the domains of some other variables, such as lower/upper bounds, full set of values, ground value or cardinality (resp. denoted by `min`, `max`, `dom`, `val`, and `card`). The expression R may use various arithmetic or set operations. In this way, the domain of X is constrained to be a subset of $R(\mathcal{D})$, evaluation of R in the current state of the domains $\mathcal{D}$. Hence, it specifies a set of consistent values for X. As explained in Sect. 3.2, an indexical can be compiled into a filtering algorithm. Many standard constraints (e.g. arithmetic, boolean or symbolic) can be expressed as conjunctions of several indexicals, using one indexical for each variable.

Indexicals can encode several levels of consistency. For instance, Fig. 1 presents how the bound-consistency of the arithmetic constraint `X+Y=Z` is expressed thanks to intervals delimited by the bounds of the involved variables (`A..B` denotes the closed interval between `A` and `B`). The right part of this figure specifies hyperarc-consistency for the same constraint thanks to pointwise arithmetic.

The interest of indexicals is threefold: their language is quite simple, they are easy to design and understand and they are executable specifications. Moreover, it is possible to implement them in a very efficient way. Even if they admittedly lack expression power (they are unable to express complex filtering algorithms of global constraints), indexicals are a powerful tool to describe a wide range of simple constraints. As a matter of fact, the whole FD library of GNU Prolog [5] is implemented using the indexical language. Indexicals also enable to quickly develop application-specific solver extensions.

The exact expression power and syntax of indexicals slightly change among implementations. Here, we consider the union of the constructions of GNU- and SICStus Prolog. Throughout the paper, we adopt SICStus' concrete syntax when applicable, and GNU Prolog syntax for the specific features of this platform.

The indexical language has two expression levels: *terms* (individual values in $\mathbb{D}$), that are combined into *ranges* (subsets of $\mathbb{D}$). In an indexical X `in` R, R is a range expression. Elementary range values are either the whole domain of a variable (e.g. `dom(X)`), an enumeration of term values or an interval between two term values. Ranges may be combined using set-operations (union, intersection, complementation) or pointwise arithmetic. Pointwise arithmetic can also combine a range and a term value. Possible elementary term values are: either lower bound (e.g. `min(X)`), upper bound, cardinality or ground value of a domain, or a constant in $\mathbb{D}$. Term values can be combined using arithmetic operations. For further details, Tab. 1 will present a comprehensive list of those constructions in Sect. 5. The denotational semantics of those constructions is described by Codognet and Diaz [4].

3.2 Compilation and Propagation of Indexicals

Indexicals have a straightforward compilation and propagation scheme. Each indexical X `in` R is compiled into a mean to evaluate $R(\mathcal{D})$ (see Sect. 3.3). The awakening of an indexical leads to this evaluation and then to the computation of the intersection between the resulting set and the current domain of X. This intersection $R(\mathcal{D}) \cap \mathcal{D}(X)$ becomes the new domain of X. An inconsistency is detected when $R(\mathcal{D}) = \emptyset$.

The awakening of an indexical is triggered off by any update of one of the domain elements involved in its range R. For instance, an indexical constraint whose range refers to $\mathtt{min}(X)$, $\mathtt{max}(Y)$, and $\mathtt{dom}(Z)$ will be awoken on each increasing of the lower bound of $\mathcal{D}(X)$, decreasing of $\max(Y)$ and on any withdrawal from $\mathcal{D}(Z)$. The correctness of this propagation scheme requires the *monotonicity* of the involved ranges:

Property 3 (Monotonicity). A range is monotonic iff $R(\mathcal{D}) \subseteq R(\mathcal{D}')$, for all pairs of domain functions $(\mathcal{D}, \mathcal{D}')$ on $\mathcal{V}$ such that $\forall v \in \mathrm{var}(R) \cdot \mathcal{D}(v) \subseteq \mathcal{D}'(v)$.

When R is non-monotonic, the evaluation of $R(\mathcal{D})$ is delayed, until $\mathcal{D}$ ensures the monotonicity. The exact strategy depends on the implementation [3, 4]. In this paper, we assume that an indexical can remove values only when its monotonicity is certain. We call this property *propagation-monotonicity*.

3.3 Evaluation of Indexicals

In SICStus, indexical ranges are compiled into postfix notation and then into byte-code for a threaded-code stack machine. When an indexical X `in` R is awoken, the stack-machine evaluates $R(\mathcal{D})$, according to the current state of the domains. Then, the solver computes the relative position of $R(\mathcal{D})$ wrt. $\mathcal{D}(X)$, meaning that the intersection is not directly computed. If the two sets are disjoint, a failure is raised. Otherwise, the inconsistent values are removed from $\mathcal{D}(X)$. In Sect. 5, we propose to take advantage of those implementation details: evaluation of $R(\mathcal{D})$ by a stack-machine and comparison with $\mathcal{D}(X)$ afterward.

In GNU Prolog, each indexical is translated at compile-time into a C function that computes $R(\mathcal{D})$. When the indexical has to be awoken, this function is called. If the resulting set is not empty, the intersection $R(\mathcal{D}) \cap \mathcal{D}(X)$ is computed thanks to a bitwise “and”, and affected to $\mathcal{D}(X)$. Therefore, there is no explicit comparison between $R(\mathcal{D})$ and $\mathcal{D}(X)$. This native compilation scheme is one of the key features of GNU Prolog and enables a faster evaluation (on this point, see the benchmarks of [3]). The primitive constraints benefit from the optimization facilities of the C compiler. A first drawback is the size of the generated code: the C function generated for a given indexical is much larger than the size of the corresponding SICStus byte-code. Another drawback is the multifile structure of the program when the user wants to define a new constraint.

4 Trivial Explanations of Indexicals

As detailed in Sect. 3, the indexical language refers to the domains in five different ways, denoted by $\texttt{min}(V)$, $\texttt{max}(V)$, $\texttt{dom}(V)$, $\texttt{val}(V)$ and $\texttt{card}(V)$. In the following, we call those references to (part of) the domains *domain-elements*. The evaluation of a range expression depends on domain-elements it contains. More specifically, R is a function of its domain-elements. The absence of a specific value from $R(\mathcal{D})$ can be explained by the current state of the domain-elements involved in R. For this purpose, we define shorthands for explaining the current state of the domain-elements: $\mathcal{E}_{\min}(X)$ is the explanation of the value of $\min(\mathcal{D}(X))$, $\mathcal{E}_{\max}(X)$ is the explanation of the value of $\max(\mathcal{D}(X))$, and $\mathcal{E}_{\mathrm{dom}}(X)$ is the overall explanation of $\mathcal{D}(X)$. Their definition is straightforward:

$$\begin{aligned}
\mathcal{E}_{\min}(X) &= \bigcup_{d\in\mathbb{D}\,\wedge\, d<\min(\mathcal{D}(X))} \mathcal{E}(X,d) \\
\mathcal{E}_{\max}(X) &= \bigcup_{d\in\mathbb{D}\,\wedge\, d>\max(\mathcal{D}(X))} \mathcal{E}(X,d) \\
\mathcal{E}_{\mathrm{dom}}(X) &= \bigcup_{d\in\mathbb{D}\,\wedge\, d\notin\mathcal{D}(X)} \mathcal{E}(X,d)
\end{aligned}$$

The cardinality of $\mathcal{D}(X)$ as well as its eventual ground value depend on all the values already removed from $\mathcal{D}(X)$: $\mathcal{E}_{\mathrm{val}}(X) = \mathcal{E}_{\mathrm{card}}(X) = \mathcal{E}_{\mathrm{dom}}(X)$.

When evaluating $c \equiv X$ **in** R, we explain the removal of the inconsistent values ($\mathcal{D}(X) - R(\mathcal{D})$) from $\mathcal{D}(X)$ by $\mathcal{T}(X, R, \mathcal{D})$, with:

$$\mathcal{T}(X,R,\mathcal{D}) = \{c\} \cup \left(\bigcup_{V\in\mathrm{var}(R)} \left(\cup_{e\in\mathrm{elements}(R,V)} \mathcal{E}_e(V)\right) \right)$$

where $\mathrm{elements}(R, v)$ is the set of domain-elements referring to variable v in R.

Example 1 (Linear equality). Let consider the two versions of $X + Y = Z$ (Fig. 1), for instance the indexicals constraining X. The indexical enforcing bound-consistency only refers to the bounds of $\mathcal{D}(Y)$ and $\mathcal{D}(Z)$. Following our trivial scheme, each value-removal from $\mathcal{D}(X)$ performed by this indexical is explained by the union of the explanations of all the values previously removed outside the bounds of $\mathcal{D}(Y)$ and $\mathcal{D}(Z)$. The hyperarc-consistency version refers to the whole domains of Y and Z: its value-removals are explained by the union of the explanations of all the previous removals from $\mathcal{D}(Y)$ and $\mathcal{D}(Z)$. In both cases, the running indexical is added to the computed explanation.

Theorem 1 (Trivial Scheme Correctness). *$\mathcal{T}(X, R, \mathcal{D})$ is a correct explanation of the value removals due to the running indexical X `in` R, assuming that this indexical is propagation-monotonic, in the sense of Sect. 3.2.*

This result can be proved by strong induction on the size of $\mathcal{T}(X, R, \mathcal{D})$. The base case is the singleton explanation, that occurs when X `in` R is a unary constraint, or when all the domain-elements involved in R have their nominal value ($\min(\mathbb{D}) \in \mathcal{D}(V)$ for a `min`(V) element, $\max(\mathbb{D}) \in \mathcal{D}(V)$ for a `max`(V) element, $\mathcal{D}(V) = \mathbb{D}$ otherwise). The induction step is straightforward for `dom`-, `val`- and `card`-elements (using Property 2). Finally, we notice that the monotonicity of R implies its monotonicity wrt. the increasing of each of its `min`-elements, and wrt. the decreasing of each of its `max`-elements.

Since an empty $R(\mathcal{D})$ leads to a failure, trivial explanations can be used as failure explanations (obvious from Def. 2):

Theorem 2 (Failure Explanation). *If a running indexical X `in` R is such that $R(\mathcal{D}) \cap \mathcal{D}(X) = \emptyset$, then $\mathcal{E}_{\mathrm{dom}}(X) \cup \mathcal{T}(X, R, \mathcal{D})$ is a failure explanation of $(\mathcal{V}, \mathcal{C})$.*

Those explanations are not guaranteed to be minimal. Nevertheless, they are easy to integrate in the evaluation process. Since the set $\cup_{v \in \mathrm{var}(R)} elements(R, v)$ is known at compile-time, the explanation computation can be statically integrated within the evaluation process. They do not need any extra-computation, except for the $\cup$-operation on $\mathcal{P}(\mathcal{C})^2$. Section 5 proposes a scheme leading to more accurate (but still non-minimal) explanations.

5 Bound-Sharp Explanations

In this Section, we propose a systematic explanation scheme that computes more precise explanations than the trivial ones (Sect. 4). This new scheme dynamically benefits from the structure of the running indexical and rests on a partition of the set of values to be removed. This partition emphasizes the role of the domain bounds. Section. 5.1 motivates this choice, then we define the explanations we want to compute (Sect. 5.2) and present the computation scheme in comparison with the syntax of indexicals (Sect. 5.3).

5.1 Domain Bounds and Constraint Propagation

Domain bounds often have a strong role to play in filtering algorithms. Many filtering algorithms are based on the well known *bound consistency* technique where each domains $\mathcal{D}(v)$ is approximated by the interval $\min(\mathcal{D}(v))..\max(\mathcal{D}(v))$. It provides a weaker level of consistency than full arc-consistency algorithms but it is often a good compromise between pruning power and computation speed.

As detailed in Sect. 5.3, for many constructions of the indexical language, the update of a bound of a domain (namely, increasing of the lower bound or decreasing of the upper one) is due to the previous updates of the bounds of one or several domains. This property holds even for some constructions used to enforce full consistency.

5.2 Definition of Bound-Sharp Explanations

Let Δ_X be the values to be removed from $\mathcal{D}(X)$ by the running indexical X `in` R, $\Delta_X = \mathcal{D}(X) - R(\mathcal{D})$. We break up Δ_X into three (possibly empty) sets:

1. $\Delta_X^{\min} = \{\delta \in \Delta_X \mid \delta < \min(R(\mathcal{D}))\}$;
2. $\Delta_X^{\max} = \{\delta \in \Delta_X \mid \delta > \max(R(\mathcal{D}))\}$;

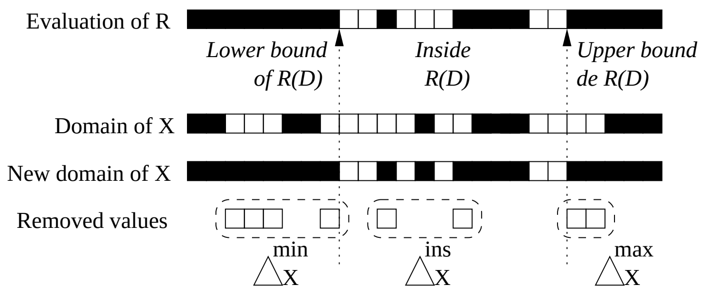


Fig. 2. Partition of the removed values based on the bounds of $R(\mathcal{D})$

3. $\Delta_X^{\text{ins}} = \Delta_X - \left(\Delta_X^{\max} \cup \Delta_X^{\min}\right)$.

Figure 2 illustrates those definitions. Domains are displayed as sets of white squares, black boxes denoting missing values. $\Delta_X^{\min}$ (resp. $\Delta_X^{\max}$) is the set of values whose removal is due to the lower (resp. upper) bound of $R(\mathcal{D})$. Δ_X^{ins} is the set of values that are removed between those bounds. Obviously, $\Delta_X^{\min}$, $\Delta_X^{\max}$, and Δ_X^{ins} constitute a partition of Δ_X. In the following, we do not compute an explanation for each removed value in Δ_X, but an explanation for each of these subparts. Therefore, each domain reduction leads to at most three distinct explanations. The computation of those three explanations will be detailed in Sect. 5.3. This computation depends on the structure of the expression R, and puts together (some of) the explanations of the domain-elements used in R.

In the following, we focus on explaining the bounds of indexical ranges and the value of term values. We extend the meaning of "explanation" as a set of constraints thats justify such values. For a range expression R_1, $\mathcal{E}_{\min}(R_1)$ denotes an explanation of the value of its lower bound $\min(R_1(\mathcal{D}))$, $\mathcal{E}_{\max}(R_1)$ denotes an explanation of its upper bound $\max(R_1(\mathcal{D}))$. $\mathcal{E}_{\text{ins}}(R_1)$ denotes an explanation of the "holes" in $R_1(\mathcal{D})$. For a term expression T_1, $\mathcal{E}_{\text{val}}(T_1)$ denotes an explanation of the value of $T_1(\mathcal{D})$.

Property 4 (Explaining the partition of Δ_X). $\{c\} \cup \mathcal{E}_{\min}(R)$ (resp. $\{c\} \cup \mathcal{E}_{\max}(R)$) is an explanation of the removal of $\Delta_X^{\min}$ (resp.$\Delta_X^{\max}$) from $\mathcal{D}(X)$, with $c \equiv X\,\texttt{in}\,R$. $\{c\} \cup \mathcal{E}_{\text{ins}}(R)$ is an explanation of the removal of Δ_X^{ins} from $\mathcal{D}(X)$.

5.3 Computation Scheme

For a given indexical X `in` R, the computations of $\mathcal{E}_{\min}(R)$, $\mathcal{E}_{\max}(R)$ and $\mathcal{E}_{\text{ins}}(R)$ are compositional, following the structure of the expression R. Those explanations are synthesized attributes of the syntactic tree associated to R. Table 1 describes all the constructions of the indexical language introduced by Sect. 3.1 (union of GNU- and SICStus Prolog constructions). First raws present elementary range values, then come combinations of range values with binary and unary operations, and possible combinations of a range value with a term. Elementary term values have already been presented: domain-elements or constants from $\mathbb{D}$. 7 arithmetic operations allow to combine

Construction Type	Syntax	Description	(a)	(b)	(c)
Range Values	$\texttt{dom}(V)$	Current state of $\mathcal{D}(V)$	×		
	$\{T_1, \ldots, T_k\}$	Set of term values *(S)*			×
	$T_1 \mathrel{..} T_2$	Interval $[\![T_1, T_2]\!] \subseteq \mathbb{D}$	×		
Composition of ranges	$R_1 \bigwedge R_2$	Intersection		×	
	$R_1 \bigvee R_2$	Union	×		
	$R_1 + R_2$	Ptw. addition	×		
	$R_1 - R_2$	Ptw. subtraction	×		
$R = R_1 \text{ op } R_2$	$R_1 * R_2$	Ptw. multiplication *(G)*	×		
	$R_1 \ / \ R_2$	Ptw. division *(G)*		×	
	$R_1 \texttt{ mod } R_2$	Ptw. modulo *(S)*			×
Range unary ops.	$\setminus R_1$	Complement			×
	$- R_1$	Ptw. opposite *(S)*	×		
Composition of a range and a term	$R_1 + T_1$	Ptw. addition	×		
	$R_1 - T_1$	Ptw. subtraction	×		
	$R_1 * T_1$	Ptw. multiplication	×		
	$R_1 \mathbin{/\!<} T_1$	Ptw. division rounded down *(G)*		×	
$R = R_1 \text{ op } T_1$	$R_1 \mathbin{/\!>} T_1$	Ptw. division rounded up *(G)*		×	
Term values	$\texttt{min}(V)$	Lower bound of $\mathcal{D}(V)$	×		
	$\texttt{max}(V)$	Upper bound of $\mathcal{D}(V)$	×		
	$\texttt{card}(V)$	Cardinality of $\mathcal{D}(V)$ *(S)*			×
	$\texttt{val}(V)$	Ground value of $\mathcal{D}(V)$			×
	n	Integer, $n \in \mathbb{D}$	×		
	$\texttt{inf}$	Lower bound of $\mathbb{D}$	×		
	$\texttt{sup}$	Upper bound of $\mathbb{D}$	×		
Composition of term values	$- T_1$	Opposite *(S)*	×		
	$T_1 + T_2$	Integer addition	×		
	$T_1 - T_2$	Integer subtraction	×		
	$T_1 * T_2$	Integer multiplication	×		
$T = T_1 \text{ op } T_2$	$T_1 \mathbin{/\!<} T_2$	Division rounded down	×		
	$T_1 \mathbin{/\!>} T_2$	Division rounded up	×		
	$T_1 \texttt{ mod } T_2$	Integer modulo	×		
Advanced features	$f(a_1, \ldots, a_k)$	Function call, the a_i are term or range values *(G)*			×
	$R_1 \ ? \ R_2$	Equal to R_2 if $R_1 \neq \emptyset$, $\emptyset$ otherwise *(S)*	×		
	$\texttt{switch}(T, L)$	Equal to $L(T)$ *(S)*	×		
$R = f(\ldots)$	$\texttt{unionof}(x, R_1, R_2) = \cup_{d \in R_1}(R_2[x \leftarrow d])$	*(S)*			×

Key: R, R_1 and R_2 are range values.
T, T_1 and T_2 are term values.
V denotes a FD variable in $\text{var}(X \ in \ R)$.
x is a local variable
$R[x \leftarrow d]$ is the substitution of x by d in R
L list of (i, R_i) with $i = j \Rightarrow R_i = R_j$, and $L(i) = R_i$
(G) labels a GNU Prolog feature.
(S) labels a SICStus Prolog feature
Ptw. stands for "pointwise"

Table 1. Unified Syntax of Indexical Constraints

Expression e	$\mathcal{E}_{\mathrm{val}}(e)$
$\mathtt{min}(V)$	$\mathcal{E}_{\mathrm{min}}(V)$
$\mathtt{max}(V)$	$\mathcal{E}_{\mathrm{max}}(V)$
$\mathtt{card}(V)$	$\mathcal{E}_{\mathrm{dom}}(V)$
$\mathtt{val}(V)$	$\mathcal{E}_{\mathrm{dom}}(V)$
$n, \mathtt{inf}, \mathtt{sup}$	$\emptyset$
$-T_1$	$\mathcal{E}_{\mathrm{val}}(T_1)$
T_1 $[+,-,/<,/>]$ T_2	$\mathcal{E}_{\mathrm{val}}(T_1) \cup \mathcal{E}_{\mathrm{val}}(T_2)$

$$\mathcal{E}_{\mathrm{val}}(T_1 * T_2) = \begin{cases} \mathcal{E}_{\mathrm{val}}(T_1) \text{ if } T_1 = 0 \\ \mathcal{E}_{\mathrm{val}}(T_2) \text{ if } T_2 = 0 \wedge T_1 \neq 0 \\ \mathcal{E}_{\mathrm{val}}(T_1) \cup \mathcal{E}_{\mathrm{val}}(T_2) \text{ otherwise} \end{cases}$$

$$\mathcal{E}_{\mathrm{val}}(T_1 \ \mathtt{mod}\ T_2) = \begin{cases} \mathcal{E}_{\mathrm{val}}(T_1) \text{ if } T_1 = 0 \\ \mathcal{E}_{\mathrm{val}}(T_2) \text{ if } T_2 = 1 \wedge T_1 \neq 0 \\ \mathcal{E}_{\mathrm{val}}(T_1) \cup \mathcal{E}_{\mathrm{val}}(T_2) \text{ otherwise} \end{cases}$$

Table 2. Bound-Sharp Explanations of Term Values

these term values into another term value. At last, four advanced features are presented. Firstly, the undocumented C-function call of GNU Prolog, with term or range values as parameters. The function f returns a range value. Secondly, SICStus provides two constructions of conditional evaluation where the evaluated range expression depends on the current value of another expression ($?$ and `switch`). Finally, SICStus' "`union`" computes the union of the evaluations of R_2 while a local variable x traverses the values in R_1 [2, Sect. 34.10.2]. For each construction, we can specify how to synthesize the explanations on top of the attributes of the subexpressions. We will firstly present the *term* level of this scheme, then we detail the scheme for a representative part of the range level.

Table 2 presents the computation scheme for the term level. The explanation of the domain-elements is straightforward using the notations of Sect. 4. The explanation of a constant term values is the indexical itself (eventually added to each explanation). Combination of term expressions are explained by combination of the explanations of the sub-expressions, using the scheme recursively. The result of a binary operation is explained by the explanations of both operands. For operations admitting an absorbent element (e.g., 0 for $\times$ and the left-operand of `mod`, 1 for the right-operand of `mod`), the term value bound to the absorbent element is sufficient to explain the result.

Table 3 presents the computation scheme for the range level. To each range construction, we attach an explanation for each bound and an explanation for its inside. Our goal is to define the explanations of the bounds on top of a subset of the explanations of the bounds of the sub-expressions. Due to lack of place, we only detail a representative collection of those expressions. The same guiding principle can be applied to missing constructions.Firstly, $\mathtt{dom}(V)$ simply inherits the explanations related to $\mathcal{D}(V)$. The bounds of $T_1..T_2$ are also easy to explain. Some other constructions loose the precision of the explanations of the bounds. For instance, in $\{T_1, \ldots, T_k\}$, each T_i can be responsible for the upper or lower bound of the result, more data about the T_i would be required to refine the computation. Other constructions always conserve the precision on the bounds (e.g. $R_1{+}R_2$). For $/\backslash$, in some cases the bounds of $R_1 \cap R_2$ can be explained by one of the bounds of the operands. In order to keep a reasonable complexity, $\mathcal{E}_{\mathrm{ins}}(e)$ often lacks of precision and is set to the union of the explanations of the whole ranges. This compromise leads to a complexity that does not depend on the cardinality of the ranges when dealing with pointwise arithmetic.

Expression e	$\mathcal{E}_{\min}(e)$	$\mathcal{E}_{\max}(e)$	$\mathcal{E}_{\text{ins}}(e)$
$\mathtt{dom}(V)$	$\mathcal{E}_{\min}(V)$	$\mathcal{E}_{\max}(V)$	$\mathcal{E}_{\text{ins}}(V)$
$T_1 \,..\, T_2$	$\mathcal{E}_{\text{val}}(T_1)$	$\mathcal{E}_{\text{val}}(T_2)$	$\emptyset$
$\{T_1, \ldots, T_k\}$	$\bigcup_{i \in [\![1,k]\!]} \mathcal{E}_{\text{val}}(T_i)$		
$-\ R_1$	$\mathcal{E}_{\max}(R_1)$	$\mathcal{E}_{\min}(R_1)$	$\mathcal{E}_{\text{ins}}(R_1)$
$R_1 \ + \ R_2$	$\mathcal{E}_{\min}(R_1) \cup \mathcal{E}_{\min}(R_2)$	$\mathcal{E}_{\max}(R_1) \cup \mathcal{E}_{\max}(R_2)$	$\mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2)$
$R_1 \ - \ R_2$	$\mathcal{E}_{\min}(R_1) \cup \mathcal{E}_{\max}(R_2)$	$\mathcal{E}_{\max}(R_1) \cup \mathcal{E}_{\min}(R_2)$	$\mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2)$
$R_1 \mathbin{/\!\backslash} R_2$	$\mathcal{E}_{\min}(e) = \begin{cases} \mathcal{E}_{\min}(R_1) & \text{if } \min(R_1) \in R_2 \\ \mathcal{E}_{\min}(R_2) & \text{if } \min(R_2) \in R_1 \\ \mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2) & \text{otherwise} \end{cases}$ $\mathcal{E}_{\max}(e) = \begin{cases} \mathcal{E}_{\max}(R_1) & \text{if } \max(R_1) \in R_2 \\ \mathcal{E}_{\max}(R_2) & \text{if } \max(R_2) \in R_1 \\ \mathcal{E}_{\text{dom}}(R_1) \cup \mathcal{E}_{\text{dom}}(R_2) & \text{otherwise} \end{cases}$ $\mathcal{E}_{\text{ins}}(e) = \ \mathcal{E}_{\text{ins}}(R_1)$		
$R_1 \ ?\ R_2$	$\mathcal{E}_{\min}(e) = \begin{cases} \mathcal{E}_{\text{dom}}(R_1) & \text{if } R_1 = \emptyset \\ \mathcal{E}_{\min}(R_2) & \text{otherwise} \end{cases}$ $\mathcal{E}_{\max}(e) = \begin{cases} \mathcal{E}_{\text{dom}}(R_1) & \text{if } R_1 = \emptyset \\ \mathcal{E}_{\max}(R_2) & \text{otherwise} \end{cases}$ $\mathcal{E}_{\text{ins}}(e) = \begin{cases} \emptyset & \text{if } R_1 = \emptyset \\ \mathcal{E}_{\text{ins}}(R_2) & \text{otherwise} \end{cases}$		

Table 3. Bound-Sharp Explanations of some Range Constructions

The correctness of this scheme is proved by strong induction on the size of the explanations, as in Sect. 4. Here, the base case and the induction step have both to be proved by induction on the structure of R, leading to a proof for each construction of the language.

The three right-most columns of Tab. 1 sum up the accuracy of $\mathcal{E}_{\min}(e)$ and $\mathcal{E}_{\max}(e)$ (or $\mathcal{E}_{\text{val}}(e)$) for all the constructions. Column (a) is checked when we always use explanations of bounds (or $\mathcal{E}_{\text{val}}(e)$) to explain the bounds of the result. Column (b) is checked when this property is conditionally verified (e.g., the intersection of two ranges). Column (c) marks range constructions that always loose this accuracy (leading to $\mathcal{E}_{\min}(e) = \mathcal{E}_{\max}(e) = \mathcal{E}_{\text{ins}}(e)$), or term values explained by a whole domain.

Example 2 (Linear equality). Following this scheme, the updates of the lower bound of $\mathcal{D}(X)$ due to the first indexical of Fig. 1 is explained by all the removals that have altered the lower bound of $\mathcal{D}(Z)$ and the upper bound of $\mathcal{D}(Y)$, and symmetrically for its upper bound.

Those explanations would be computed the same for the full consistency version. That is much more accurate than the trivial scheme, that considers the state of the whole $\mathcal{D}(Y)$ and $\mathcal{D}(Z)$ as responsible for any removal. However, the removal of an internal value will not be more precisely explained with the bound-sharp scheme.

Implementation. When the indexicals are evaluated by a stack-machine, the implementation of this scheme is straightforward: a stack of explanations is added to the machine. Each time a range or term value $e(\mathcal{D})$ is pushed on the evaluation stack, a record containing its explanations is pushed on the explanation stack. When an indexical operation is applied, the explanations of the popped operands are used to compute the explanations of the result. However, in an indexical-based solver, about half of the range evaluations are useless (the resulting $R(\mathcal{D})$ is a superset of $\mathcal{D}(X)$) [4]. The implementation proposed above computes three explanations for each indexical construction. If this evaluation does not imply any domain reduction, those explanations are useless. Moreover, if a reduction is needed, some of the computed explanations may still be useless, depending on the emptiness of $\Delta_X^{\min}$, $\Delta_X^{\max}$, and Δ_X^{ins}. Therefore, we now adapt this scheme to lazily compute useful explanations, avoiding useless calculations. Firstly, we instrument the evaluation function to keep tracks of intermediary results. Then, if needed, we compute the suitable explanations using the track of the evaluation process.

While the stack machine evaluates $R(\mathcal{D})$, the intermediate results (terms or ranges) are stored in a tree. Each operation performed by the stack-machine leads to the creation of a node of this tree, labeled by the stack-machine operation performed. By construction, this tree has the same structure as the evaluated indexical. We recycle intermediate results: the only overhead is due to the construction of the tree-structure. At the end of the evaluation process, the resulting tree is a track of this process. Once the evaluation of $R(\mathcal{D})$ is complete, $\Delta_X^{\min}$, $\Delta_X^{\max}$ and Δ_X^{ins} are computed. If needed, we recursively compute the suitable explanations using the intermediary results stored in the tree. This afterward computation may avoid irrelevant parts of the expression R.

6 Related Work

PaLM [13] is a finite domain constraint solver that provides explanations. Each primitive constraint is instrumented so as to compute a locally-minimal explanation for each value removal. Even if PaLM provides some global constraints that indexicals cannot express, its explanations of arithmetic constraints (e.g. linear equality, inequality) are equal to the ones our bound-sharp scheme produces. Therefore, our approach enable to reach the same accuracy for the constraints in its scope, and in a systematic way.

Ferrand et al [10] proposed a generic scheme to compute explanations from a fine-grained execution trace of a CP(FD) solver. Their scheme is not intrusive, as soon as the solver provides such a tracer. Such a tracer exists for GNU-Prolog [15] and one is under development for SICStus Prolog. Nevertheless, the explanations this algorithm would compute are less accurate than ours. Actually, the trace is not aware of what kind of domain information are used. Their explanations have thus to take into account all the domain reductions of the variables involved in the reducing constraint from the beginning of the computation.

Junker [12] proposed a non-intrusive technique to compute inclusion-minimal conflict sets: QUICKXPLAIN. A slight modification allows withdrawal explanations to be computed the same way. It can be applied to any solver based on incremental adding of constraints, without instrumentation. Moreover, the overhead has to be paid only when a conflict exists, that is to say when the resolution fails. Nevertheless, the method can only be applied to over-constrained problems: it cannot be used to handle dynamic problems or to enhance the search-procedure. Our scheme provides thus more diverse functionalities for indexical-based solvers.

7 Conclusion

This paper has presented a computation scheme to automatically produce explanations for indexical-based solvers. This scheme uses the structure of an indexical to explain its filterings. The scheme emphasizes the accuracy of the explanations of the bounds of the domains as a compromise between accuracy of the explanations and limitation of their overhead. It provides the same accuracy as state-of-the-art explanation-based solvers for some classes of constraints.

Our scheme enables to automatically explain various existing solvers as well as user-defined constraints using indexicals. The variety of constraints we can handle is only limited by the expression power of the indexical language.

Acknowledgments The author is indebted to Mats Carlsson and Per Mildner for valuable help about the implementation of SICStus Prolog. Fruitful discussions with Narendra Jussien raised some of the ideas exposed in this paper. Mireille Ducassé made helpful remarks about a previous draft of this paper.

References

1. M. Ågren. Tracing and explaining the execution of clp(fd) programs in sicstus prolog. Technical Report T2002:10, Swedish Institute for Computer Science, 2002.
2. M. Carlsson et al. *SICStus Prolog 3.12.0 User's Manual*. SICS R.R., 2004.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser et al., editors, *Proc. of Programming Languages: Implementations, Logics and Programs*, number 1292 in LNCS. Springer, 1997.
4. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
5. D. Diaz et al. GNU-Prolog, a clp(fd) system based on Standard Prolog (ISO), 2003. http://gprolog.sourceforge.net/ Distributed under the GNU license.
6. R. Douence and N. Jussien. Non-intrusive constraint solver enhancements. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'02)*, volume 344 of *CW Reports*, pages 26–36, 2002.
7. A. Elkhyari, C. Guéret, and N. Jussien. Constraint programming for dynamic scheduling problems. In H. Kise, editor, *ISS'04 Intl. Scheduling Symposium*. Japan Society of Mechanical Engineers, 2004.
8. G. Ferrand, W. Lesaint, and A. Tessier. Theoretical foundations of value withdrawal explanations for domain reduction. *ENTCS*, 76, 2002.
9. G. Ferrand, W. Lesaint, and A. Tessier. Towards declarative diagnosis of constraint programs over finite domains. In M. Ronsse, editor, *Proc. of the 5th Intl. Workshop on Automated Debugging, AADEBUG2003*, pages 159–170, 2003.
10. G. Ferrand, W. Lesaint, and A. Tessier. Explanations to understand the trace of a finite domain constraint solver. In S. Muoz-Hernàndez et al., editors, *14th Workshop on Logic Programming Environments*, pages 19–33, 2004.
11. C. Guéret, N. Jussien, and C. Prins. Using intelligent backtracking to improve branch and bounds methods: an application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354, 2000.
12. U. Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.

13. N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proc. of CP workshop on Techniques foR Implementing Constraint programming Systems*, 2000.
14. N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in LNCS, pages 249–261. Springer-Verlag, 2000.
15. L. Langevine, M. Ducassé, and P. Deransart. A propagation tracer for gnu-prolog: from formal definition to efficient implementation. In C. Palamidessi, editor, *Proc. of ICLP'03,*, LNCS. Springer-Verlag, 2003.
16. S. Ouis, N. Jussien, and P. Boizumault. COINS: a constraint-based interactive solving system. In A. Tessier, editor, *Proc. of the 12th Workshop on Logic Programming Environments, (*WLPE*'02)*, 2002. CoRR cs.SE/0207046.
17. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraints: Basics and Trends*, number 910 in LNCS. Springer Verlag, 1995.

Closures are Needed for Closed Module Systems

Rémy Haemmerlé and François Fages

INRIA, Projet Contraintes
Domaine de Voluceau, Rocquencourt
BP 105, 78153 Le Chesnay Cedex, France
FirstName.LastName@inria.fr

Abstract. In a classical paper of D.H.D. Warren, the higher-order extensions of Prolog were questioned as they do not really provide more expressive power than meta-programming predicates. Without disputing this argumentation in the context of a logic programming system without modules, we show that the situation is different in a closed module system. By closed we mean the property that the module system is able to prevent any call to the private predicates of a module from the other modules, in particular through meta-programming predicates. We show that this property necessitates to distinguish the execution of a term (meta-programming predicate `call`) from the execution of a closure (higher order). We propose a module system for Constraint Logic Programming with a notion of closures inspired from Linear Concurrent Constraint programming. This module system is quite simple and pretty independent of a precise language (it is currently implemented for GNU-Prolog). Although this system can be seen as a simple layer of syntactic sugar, it does provide a discipline for naming predicates and hiding code, making possible the development of libraries and facilitating the safe reuse of existing code. Furthermore we provide the module system with logical and operational semantics. This formal setting is used in the paper to compare our approach to the other module systems proposed for Prolog, which generally do not ensure full code protection.

1 Introduction

It is often easier to work with a small number of concepts in mind. Thus to be written, large programs must be divided into small parts. Although this can be done in any language, a suitable module system helps the segmentation of programs by forcing the programmer to clearly define the limits and the interface of each block. Hence many errors can be avoided, the compiler being in charge of verifying that the calls between modules respect the declared interfaces. Moreover such divisions facilitate the understanding of a program by external programmers and improve the reusability of the code.

The context of our work is the design of a fully bootstrapped implementation of a Linear logic Concurrent Constraint (LCC) [11, 25] programming language called **SiLCC**, for "SiLCC is Linear Concurrent Constraint programming". The purpose of the present paper is to present the theoretical and practical choices we made for its module system. For the sake of simplicity and because we think, as Leroy [18], that "modular programming has little to do with the particulars of any programming language" we will present our module system in the formal setting of Constraint Logic Programs (CLP) [17] rather than in the more general setting of LCC.

The purpose of a module system is not unique and it is important to list the different purposes a module system may serve:

Implementation Hiding. A programmer should have the possibility to protect his code from the intrusion from other modules. This means that it should be possible to restrict the access of a module (i.e. calls, dynamic assertions/retractions, syntax modifications, global variable assignments etc ...) from extra-modular code. In the following, we will say that a module system that ensures *code protection* is a *closed* module system, on the contrary a system that allows any call from a module to another will be called *open*.

Separation of name space. In large *flat* (i.e. without modules) system, name clashes are frequent and disturbing. Indeed, to be certain to avoid such clashes, the programmer is constrained to systematically prefix the name of his predicates. A module system has to avoid such clashes automatically.

Separate compilation. It should be possible to compile in an independent way each module of a program. In particular, standard libraries must be compiled once for all.

Bootstrapping. One of the most original aspects of SiLCC is the realization of a completely bootstrapped implementation of a constraint language from a small kernel (LCC with constraints over labeled graphs and linear logic constraints providing imperative features). Modules are essential to the bootstrap of such a complex language.

Furthermore, the design of a module system for Prolog needs to meet some other requirements:

Simplicity. The module system should not restrict the use of Prolog for rapid prototyping, in particular :
- the conciseness of Prolog code should be preserved, avoiding too many module-related declarations;
- meta-programming predicates such as the `call` predicate should be supported;
- new concepts should be limited in order to be adopted by classical Prolog programmers.

Semantics. The module system should come with a formal semantics from which one can derive its properties.

Simple implementation. We want the module system to be easy to code and to port onto different logic languages, ranging from GNU-Prolog to SiLCC.

Modularity in the context of logic programming has been considerably studied, and there has been some standardization attempts [16]. In order to define a notion of module that captures important aspects such as import/export and implementation hiding, the *logical approaches* rely on an extension of the underlying logic. For example, one can cite extensions with nested implications [19], meta-logic [2] or second order predicates [7]. Some other approaches, such as contextual logic programming [20], or object-oriented extensions [21], go even further in the direction of fully dynamic module systems.

Other proposals add constructs for declaring and handling a notion of static modules. On the one hand, there are *algebraic approaches* defining module calculi on sets of program clauses, such as the module calculus of O'Keefe [22], or of Bugliesi, Lamma et Mello [4], or the calculus of Sanella and Wallen [24] inspired from functional programming. On the other hand, the so-called *syntactic approaches* deal mainly with the alphabet of symbols. This approach is criticized in [22, 24] for its lack of logical semantics. Nevertheless syntactic module systems are often chosen for their simplicity and compatibility with existing code. For instance, the existing code of OEFAI CLP(q,r) [14] or a Prolog implementation of CHR [26] should be ported as libraries in a modular system. Most of current modular Prolog systems, such as SICStus [27], SWI [29], ECLiPSe [1], XSB [23], Ciao [3, 6, 5], fall into this category. None of them however ensures every purpose we have exposed previously. For example, the popular system SICStus does not provide any kind of implementation hiding.

In this paper, we show that closures are needed to ensure code protection in a predicate-based syntactic module system. We propose a closed module system which distinguishes meta-programming predicates from closures. We give an operational semantics which is used to prove the properties of the module system, and to compare the different existing systems. Furthermore we provide a logical semantics with a translation of modular constraint logic programs into CLP programs with a simple module constraint system.

The paper is organized as follows. The next section defines the syntax of the module system we propose for CLP. Section 3 presents the operational semantics of modular constraint logic programs (MCLP) with meta-calls and closures, and shows that this module system satisfies the code protection property. Section 4 presents the logical semantics of pure MCLP programs, i.e without meta-predicates nor closures. Then, in section 5, we discuss some pragmatic aspects of our system. Finally we compare our proposal with existing syntactic module systems and conclude.

2 Modular Constraint Logic Programs

We consider the following disjoint alphabets:

- V a set of countable variables denoted by $x, y \ldots$;
- Σ_F a set of constant and function symbols;
- Σ_C a set of constraint predicate symbols containing $=$ and $true$;
- Σ_P a set of program predicate symbols containing $call/2$, $closure/3$ and $apply/2$;
- Σ_M a set of module names,
- two relations $\overset{M}{\leftrightsquigarrow}: \Sigma_F \times \Sigma_M$ and $\overset{P}{\leftrightsquigarrow}: \Sigma_F \times \Sigma_P$

and the usual sets of terms, formed over V and Σ_F, atomic constraints, formed with predicate symbols in Σ_C, and atoms, formed with predicate symbols in Σ_P. In addition, we consider module names, noted μ,..., and atoms prefixed with a module name, noted $\mu\!:\!A$ and called *qualified atoms*.

Because Σ_F, Σ_P and Σ_M are supposed to be distinct, the two relations $\overset{P}{\leftrightsquigarrow}$ and $\overset{M}{\leftrightsquigarrow}$ will be useful to interpret function symbols respectively as predicates symbols and as module names while a meta-call. In classical Prolog systems, where function symbols , predicate symbols and module names are not syntactically distinguished, this two relations can be viewed as the trivial bijections.

For the sake of simplicity, we assume here that all atoms are qualified and do not describe the standard conventions (given in Sect. 5) that are used for prefixing automatically the atoms given without module names in a clause or a goal.

Definition 1. *A **closure** is a formula of the form*

$$closure(x, \mu\!:\!A, z)$$

where x and z are variables, A is an atom and μ is a module identifier.

The closure $closure(x, \mu\!:\!A, z)$ associates to variable z a qualified atom $\mu\!:\!A$ in which the variable x is abstracted. A closure z is applied to an argument x with the predicate $apply(z, x)$.

Definition 2. *A **MCLP clause** is a formula of the form*

$$A_0 \leftarrow c_1, \ldots, c_l | \kappa_1, \ldots, \kappa_n | \mu_1\!:\!A_1, \ldots, \mu_m\!:\!A_m.$$

where the κ_i's are closures, the A_i's are atoms, the c_i's are atomic constraints and the μ_i's are module names.

Definition 3. *A **module** is a tuple $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)$ where $\mu \in \Sigma_M$ is the* name *of the module, $\mathcal{D}_\mu$ a set of clauses is called* implementation *of the module and $\mathcal{I}_\mu \subset \Sigma_P$ is the* interface *of the module. If a symbol p belongs to the interface of a module called μ, we will say that p is* public *in μ, otherwise we will say that p is* private *in μ .*

Definition 4. *A **MCLP Program** $\mathcal{P}$ is a set of modules whose names are distinct.*

Definition 5. *A **MCLP goal** is a formula*

$$c_1, \ldots, c_l |\, \langle \nu_1 - \kappa \rangle\,, \ldots, \langle \nu_n - \kappa_n \rangle \,|\, \langle \nu'_1 - \mu_1\!:\!A_1 \rangle\,, \ldots, \langle \nu'_m - \mu_m\!:\!A_n \rangle$$

where the c_i's are atomic constraints, the κ_i are closures, the $(\mu_i : A_i)$'s are qualified atoms and both the ν_i's and the ν'_i's are module names called in this case calling contexts. *$\langle \nu_i - \mu_i\!:\!A_i \rangle$'s are called* atoms with context *whereas $\langle \nu_i - \kappa_i \rangle$'s are called* closures with context.

In the following $\langle \nu - (\kappa_1, \ldots, \kappa_n) \rangle$ will be a notation for the sequences of closures with context $(\langle \nu - \kappa_1 \rangle, \ldots, \langle \nu - \kappa_n \rangle)$ and $\langle \nu - (\mu_1 : A_1, \ldots, \mu_m : A_m) \rangle$ a notation for the sequence of atoms with context $(\langle \nu - \mu_1 : A_1 \rangle, \ldots, \langle \nu - \mu_m : A_m \rangle)$.

Example 1. The following classical implementation of the `findall/3` predicate illustrates the difference between a meta-call and the application of a closure in a module.

```
findall(X,G,_) :- call(G), asserta(found(X)), fail.
findall(_,_,L) :- collect([],L).
collect(S,L) :- retract(found(X)), collect([X|S],L).
collect(L,L).
```

The use of `call` in this implementation does not prevent intrusion from another module with a goal like `findall(X,retract(found(X)),L)`. On the other hand, an implementation of `findall` using closures and `apply` instead of `call` as below

```
findall(C,_) :- apply(X,C), asserta(found(X)), fail.
findall(_,L) :- collect([],L).
```

ensures the protection of the code. For instance, the clauses retracted by the closure in `closure(X,retract(found(X)),C),findall(C,L)` will be safely retracted in the calling module, and not in the called module.

In the reverse direction, a typical use of meta-calls instead of closures is for the writing of a top-level or a meta-interpreter.

3 Operational Semantics

3.1 Transition System

Definition 6. *Let $\mathcal{P}$ be a MCLP program. The rewriting relation $\longrightarrow$ on goals is defined as the least relation satisfying the rules in table 1.*

Modular CSLD. The *modular CSLD* resolution rule is a restriction of the classical CSLD rule for CLP [17]. The additional condition $(\nu = \mu) \vee (p \in \mathcal{I}_\mu)$ imposes that $\mu : p(\boldsymbol{t})$ can be executed only if, either the call is made from inside the module (i.e. from the calling context μ), or the predicate p is a public predicate in μ. Moreover, this rule propagates the calling context to the new atoms of the body of the selected clause.

In the following, we call *pure MCLP* languages, the class of MCLP languages built with the *modular CSLD* rule only. In this class, closures cannot be executed and are forbidden in clauses and goals.

Meta-call. The *call* rule gives an operational semantics to meta-calls. It translates two terms into a qualified atom, respecting both the conversion relations $\overset{P}{\leftrightsquigarrow}$ and $\overset{M}{\leftrightsquigarrow}$. It is worth noting that this rule does not change the calling context, which is necessary to guarantee the implementation hiding.

Modular CSLD	$\dfrac{(\mu,\mathcal{D}_\mu,\mathcal{I}_\mu)\in\mathcal{P} \qquad (\nu=\mu)\vee(p\in\mathcal{I}_\mu) \\ (p(\boldsymbol{s})\leftarrow c'\vert k\vert\beta)\theta\in\mathcal{D}_\mu \qquad \mathcal{X}\models\exists(c\wedge\boldsymbol{s}=\boldsymbol{t}\wedge c')}{(c\vert K\vert\gamma,\langle\nu-\mu:p(\boldsymbol{t})\rangle,\gamma')\longrightarrow(c,\boldsymbol{s}=\boldsymbol{t},c'\vert K,\langle\mu-k\rangle\vert\gamma,\langle\mu-\beta\rangle,\gamma')}$
Call	$\dfrac{\mathcal{X}\models\exists(c\wedge t=g\wedge s=f(\boldsymbol{x})) \qquad f\overset{P}{\leftrightsquigarrow}p \qquad g\overset{M}{\leftrightsquigarrow}\mu}{(c\vert K\vert\gamma,\langle\nu-\nu:call(t,s)\rangle,\gamma')\longrightarrow \\ (c,t=g,s=f(\boldsymbol{x})\vert K\vert\gamma,\langle\nu-\mu:p(\boldsymbol{x})\rangle,\gamma')}$
Apply	$\dfrac{\langle\mu-closure(x,\mu':A,z)\rangle\in K \qquad \mathcal{X}\models c\Rightarrow z=y}{(c\vert K\vert\gamma,\langle\nu-\nu:apply(y,t)\rangle,\gamma')\longrightarrow(c\vert K\vert\gamma,\langle\mu-\mu':A[x\backslash t]\rangle,\gamma')}$

Table 1. Transition relation for MCLP goals with calls and closures.

For the sake of simplicity, our definition of $call/2$ does not allow the meta-call of conjunctions of atoms nor the meta-call of a constraint. These calls can be emulated however, by supposing (','$/2 \overset{P}{\leftrightsquigarrow} and/2$) and by adding the clause $(and(x,y)\leftarrow\mu:call(x),\mu:call(y))$ to the implementation of any module μ.

In this formal semantics of meta-calls, the goal $(c\vert\,\langle\mu-\nu:call(t,s)\rangle)$ succeeds even if the argument of $call/2$ is a free variable. A failure would not be a better solution however, as it would not preserve the independence of the selection strategy. This can be illustrated by the following goal (`X=true, call(X)`). If the meta-call of a free variable failed, a left-to-right selection strategy would lead to the computed answer `X=true` whereas a right-to-left strategy would lead to a failure. The proper way to handle such errors is to raise an exception, which is not formalized here.

Closure. The *apply* rule allows the invocation of a closure collected by a previous predicate call. In practice, it looks for the closure associated to the closure variable (formally checks the equality of variables $z=y$), and applies the closure to the argument in the closure context. One can remark that $closure/3$ and $apply/2$ can be defined in Linear logic Concurrent Constraint programming (LCC) [11] as mere syntactic sugar

$$closure(x,A,z)\equiv\,!\forall x.((arg(z,x))\rightarrow A)$$
$$apply(t,z)\equiv arg(z,t)$$

In this definition, a closure is an LCC agent which waits for its argument given in a token (linear logic constraint) $arg(z,x)$ that is posted by the apply agent.

Example 2. Closures can be used to define general iterator predicates for data structures. In a module defining some data structure, it is possible to define the binary predicates `forall/2` and `exists/2` that check that every (resp. at least one) element of a data structure passed in the first argument, verifies a property passed as a closure in the second argument. For instance, in a library for lists, such iterators can be defined as follows:

```
forall([], C).
forall([X|T], C) :- apply(C, X), forall(T, Z).
exists([X|_], C) :- apply(C, X).
exists([_|T], C) :- exists(T, C).
```

Example 3. It is instructive to try to employ a meta-call instead of a closure in the previous example.

```
forall([], P).
forall([H| T], P):- G=..[P, H], call(G), forall(T, P).
```

Now let `foo/1` be a private predicate defined in a different module from the one defining `forall/1`. Then, a goal like `forall([1,2,3], foo)` will always fail as `foo` is not visible. To bypass this restriction, a naive solution consists of declaring as public every predicate used as high-order data (`foo/1` in our example). Because this method violates the protection of the code implied by implementation hiding, it is not acceptable. Indeed the system is no more capable to prevent any call of a predicate used as high-order data.

3.2 Implementation Hiding

In this section, we propose a formal definition of the implementation hiding property, and show that our module system respects it.

Lemma 1. *If* $(c|K|\alpha) \longrightarrow (c'|K'|\beta', \langle \nu - A\rangle, \beta'')$ *is a transition then :*

1. *Either there exists* $\langle \nu - \mu : p(\boldsymbol{x})\rangle$ *in* α*;*
2. *or there exists* $\langle \nu - closure(x, B, z)\rangle$ *in* K *with* $A = B[x\backslash t]$ *for some* t*;*
3. *or there exists* $\langle \mu - \nu : p(\boldsymbol{x})\rangle$ *in* α *such that* p *is public in* ν*.*

Proof. Let us suppose that $\langle \nu - A\rangle$ is not in α, as if it is not the case we are trivially in case *1*. If the transition is a *modular CSLD* transition, α is of the form $\gamma', \langle \nu' - \nu : p(\boldsymbol{x})\rangle, \gamma''$ with $(\nu' = \nu) \vee (p \in \mathcal{I}_\nu)$. If $(\nu' = \nu)$ then we are in case *1*, otherwise $(p \in \mathcal{I}_\nu)$ and we are in case *3*. If the transition is a *call*, α is of the form $\gamma', \langle \nu - \nu : call(s,t)\rangle, \gamma''$ which corresponds to case *1*. Finally if the transition is an *apply*, $\langle \nu - closure(x, B, z)\rangle$ is in K with $A = B[x\backslash t]$ for some term t, which corresponds to case *2*.

Lemma 2. *If* $(c_0|K_0|\alpha_0) \longrightarrow (c_1|K_1|\alpha_1) \longrightarrow \ldots \longrightarrow (c_n|K_n|\alpha_n)$ *is a derivation and if* $\langle \nu - \kappa\rangle$ *is a closure with context in the sequence* K_n *then :*

1. *Either there exists* $\langle \nu - \kappa\rangle$ *is in* K_0*;*
2. *or there exists* $\langle \nu - \nu : p(\boldsymbol{x})\rangle$ *in an* α_i *such that* $0 \leq i < n$*;*
3. *or there exists* $\langle \mu - \nu : p(\boldsymbol{x})\rangle$ *in an* α_i *such that* $0 \leq i < n$ *and* $p \in \mathcal{I}_\nu$*,*

Proof. By induction on the length of the derivation. If $n = 0$ we are trivially in case *1*. Otherwise either $\langle \nu - \kappa\rangle$ is in K_{n-1}, in wich case we conclude by the induction hypothesis, or $\langle \nu - \kappa\rangle$ is produced by a *modular CSLD* transition. In the latter case α_{n-1} is of the form $\gamma', \langle \nu' - \nu : p(\boldsymbol{x})\rangle, \gamma''$ with $(\nu' = \nu) \vee (p \in \mathcal{I}_\nu)$. If $(\nu' = \nu)$ we are in case *2*, otherwise we are in case *3*.

The property of implementation hiding formally states that to enter a module we need to pass through a public predicate of this module.

Proposition 1 (Implementation Hiding). *Let* $(c_0|K_0|\alpha_0) \longrightarrow (c_1|K_1|\alpha_1) \longrightarrow \ldots \longrightarrow (c_n|K_n|\alpha_n)$ *be a derivation. If* $\langle \nu - A \rangle$ *belongs to the sequence* α_n *then :*

1. *Either there exists* $\langle \nu - \mu : p(\boldsymbol{x}) \rangle$ *in* α_0*;*
2. *or there exists* $\langle \nu - \kappa \rangle$ *in* K_0*;*
3. *or there exists* $\langle \mu - \nu : p(\boldsymbol{x}) \rangle$ *in an* α_i $(0 \leq i < n)$ *such that* p *is public in the module* ν*;*

Proof. By induction on the length of the derivation. If $n = 0$ then we are trivially in case *1.* Otherwise by lemma 1 we know that :

- Either there exists $\langle \nu - \mu : p(\boldsymbol{x}) \rangle$ in α_{n-1}, in which case the proposition is true by induction hypothesis;
- or there exists $\langle \nu - \kappa \rangle$ in K_{n-1}, in which case the correction is deduced by lemma 2 and induction hypothesis;
- or there exists $\langle \mu - \nu : p(\boldsymbol{x}) \rangle$ in α such that p is public in ν, which corresponds to the case *3.*

4 Logical Semantics

4.1 Modules as a Constraint System $\mathcal{M}$

To a given MCLP program $\mathcal{P}$, one can associate a simple module constraint system $\mathcal{M}$, in which the constraint $allow(\nu, \mu, p)$ that states that the predicate p of module μ can be called in module ν, is defined by the following axiom schemas:

$$\frac{\nu \in \Sigma_M \quad p \in \Sigma_P}{\mathcal{M} \models allow(\nu, \nu, p)} \qquad \frac{\nu, \mu \in \Sigma_M \quad (\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \quad p \in \mathcal{I}_\mu}{\mathcal{M} \models allow(\nu, \mu, p)}$$

This constraint system depends solely on the interface of the different modules that composes the program $\mathcal{P}$, and not on its implementation.

4.2 Translation of MCLP($\mathcal{X}$) Programs into CLP($\mathcal{M}, \mathcal{X}$) Programs

The logical semantics of modular CLP programs is obtained by a formal translation of pure MCLP($\mathcal{X}$) programs into ordinary CLP($\mathcal{M}, \mathcal{X}$) programs. This also shows that the module system can be viewed as simple syntactic sugar.

The alphabet $\dot{\Sigma}_P$ of the associated CLP($\mathcal{M}, \mathcal{X}$) program, is constructed by associating one and only one predicate symbol $\dot{p} \in \dot{\Sigma}_P$ of arity $n + 2$ to each predicate symbol $p \in \Sigma_P$ of arity n. The translation Π of the MCLP program is as follows:

$\Pi_\mu(\nu:p(\boldsymbol{t})) = \dot{p}(\mu,\nu,\boldsymbol{t})$ $\Pi_\mu(A,A') = \Pi_\mu(A),\Pi_\mu(A')$ $\Pi_\mu\left(p_0(\boldsymbol{t}) \leftarrow c\vert\alpha\right) = \dot{p_0}(y,\mu,\boldsymbol{t}) \leftarrow allow(\mu,y,p_0),c\vert\Pi_\mu(\alpha)$ $\Pi_\mu\left(\bigcup\{(A \leftarrow c\vert\alpha)\}\right) = \bigcup\{\Pi_\mu(A \leftarrow c\vert\alpha)\}$ $\Pi\left(\bigcup\{(\mu,\mathcal{D}_\mu,\mathcal{I}_\mu)\}\right) = \bigcup\{\Pi_\mu(\mathcal{D}_\mu)\}$ $\Pi^{\langle\rangle}\left(\langle\nu-\mu:p(\boldsymbol{t})\rangle\right) = \dot{p}(\nu,\mu,\boldsymbol{t})$ $\Pi^{\langle\rangle}(\gamma,\gamma') = \Pi^{\langle\rangle}(\gamma),\Pi^{\langle\rangle}(\gamma')$

Table 2. Formal tanslation of MCLP($\mathcal{X}$) to CLP($\mathcal{M},\mathcal{X}$)

Proposition 2 (Soundness). *Let $\mathcal{P}$ and $(c|\gamma)$ be a pure MCLP program and a pure MCLP goal*

$$if\ \left((c|\gamma) \xrightarrow{\mathcal{P}} (d|\gamma')\right)\ then\ \left((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d, allow(y,\mu,p), y=\nu|\Pi^{\langle\rangle}(\gamma'))\right)$$

for some ν, μ, p and such that y is not free in d.

Proof. Let us suppose $((c|\gamma) \xrightarrow{\mathcal{P}} (d|\gamma'))$. Let $\langle\nu-\mu:p(\boldsymbol{t})\rangle$ be the selected atom in γ. Then γ is of the form $(\gamma_1, \langle\nu-\mu:p(\boldsymbol{t})\rangle, \gamma_2)$ for some γ_1 and γ_2. Hence we have $\Pi^{\langle\rangle}(\gamma) = \left(\Pi^{\langle\rangle}(\gamma_1), \dot{p}(\nu,\mu,\boldsymbol{t}), \Pi^{\langle\rangle}(\gamma_2)\right)$. Let $(p(\boldsymbol{s}) \leftarrow c'|\alpha)\theta$ be the selected clause in module μ. Then we have $(\dot{p}(y,\mu,\boldsymbol{s})) \leftarrow c, allow(y,\mu,p)|\Pi_\mu(\alpha))\,\theta$ in the translation of $\mathcal{P}$. We also have $d = (c, \boldsymbol{t}=\boldsymbol{s}, c')$, $X \models \exists(d)$ and $(\nu=\mu) \vee (p \in \mathcal{I})$. As $(\nu=\mu) \vee (p \in \mathcal{I})$ is true, the constraint $allow(\nu,\mu,p)$ is true in $\mathcal{M}$, hence we have $\mathcal{X},\mathcal{M} \models \exists d'$ with $d' = (c, (\nu,\mu,\boldsymbol{t})=(y,\mu,\boldsymbol{s}), c', allow(y,\mu,p)$. Therefore we have $((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d'|\Pi^{\langle\rangle}(\gamma')))$.

Lemma 3. *The functions Π_μ and $\Pi^{\langle\rangle}$ are injective.*

Proof. As they are composition of injective functions,

Π_μ on qualified atomes, atom sequences and clauses is a composition of injective functions and therefore is injective. For the same reason $\Pi^{\langle\rangle}$ is injective. Because Π_μ on modules is the pointwise extension of Π_μ defined on clauses, it is injective too.

Proposition 3 (Completeness). *Let $\mathcal{P}$ and $(c|\gamma)$ be a pure MCLP program and a pure MCLP goal*

$$if\ \left((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d|\alpha)\right) then \left((c|\gamma) \xrightarrow{\mathcal{P}} (d'|\gamma')\right)$$

where $\Pi^{\langle\rangle}(\gamma') = \alpha$ and $d' = (d, allow(y,\mu,p), y=\nu)$ for some ν, μ, p and such that y is not free in d.

Proof. Let us suppose that $((c|\mathit{\Pi}^{\langle\rangle}(\gamma)) \xrightarrow{\mathit{\Pi}(\mathcal{P})} (d|\gamma'))$. The constraint c does not contain any $allow/3$ constraint since $(c|\gamma)$ is a MCLP goal. Let $q(\boldsymbol{t})$ be the selected atom, $\mathit{\Pi}^{\langle\rangle}(\gamma)$ is of the form $(\gamma_1, q(\boldsymbol{t}), \gamma_2)$ for some γ_1 and γ_2. Hence we have $\gamma = {\mathit{\Pi}^{\langle\rangle}}^{-1}(\gamma_1), p(\nu, \mu, \boldsymbol{t'}), {\mathit{\Pi}^{\langle\rangle}}^{-1}(\gamma_2)$ with $q = \dot{p}$ and $t = (\nu, \mu, \boldsymbol{t'})$. Let $q(\boldsymbol{s}) \leftarrow c'|\beta$ be the selected clause. We have $p(\boldsymbol{s'}) \leftarrow c''|{\mathit{\Pi}_{\mu'}}^{-1}(\beta)$ in the implementation of some module μ', with $\boldsymbol{s} = (y, \mu', \boldsymbol{s'})$ and $c'' = c', allow(y, \mu', p)$ where y is fresh. We have $d = (c, c'', allow(y, \mu', p), (\nu, \mu, \boldsymbol{t'}) = (y, \mu', \boldsymbol{s'}))$ and $\mathcal{X}, \mathcal{M} \models \exists(d)$. Hence for $d' = (c, c'', \boldsymbol{t'} = \boldsymbol{s'})$, we have $\mathcal{X}, \mathcal{M} \models \exists(d')$. Therefore, for $\alpha = ({\mathit{\Pi}^{\langle\rangle}}^{-1}(\gamma_1), {\mathit{\Pi}^{\langle\rangle}}^{-1}(\beta), {\mathit{\Pi}^{\langle\rangle}}^{-1}(\gamma_2))$, we conclude that $(c|\gamma)$ can be reduced by a clause of $\mathcal{P}$ to $(d'|\alpha)$ with $\mathit{\Pi}^{\langle\rangle}(\alpha) = \gamma$.

The soundness and completeness of the translation thus show that a MCLP($\mathcal{X}$) programs are provided with simple logical semantics given by the ordinary logical semantics of their translations as a CLP($\mathcal{M}, \mathcal{X}$) programs.

5 Pragmatic Aspects

In practice, we deal with a set of *atoms* defined as in ISO Prolog [10] to represent function symbols, predicate symbols and module names. In this chapter, the term atom has thus a different meaning than in the previous sections, and we will use the term *predicate* to refer to the atomic propositions as defined in Sect. 2. As the position of an atom in a clause makes it possible to determine which functor/arity is represented, the two relations $\overset{M}{\leftrightsquigarrow}$ and $\overset{P}{\leftrightsquigarrow}$ will be the trivial bijections.

5.1 Defining a Module

A module is a set of clauses and directives contained in a single file named with the name of the module and a `.pl` extension. The file must begin with a `module/2` declaration which specifies the name of the module and its interface. The name of the module is an atom, whereas its interface must be either a list of predicate specifications of the form `Functor/Arity` for representing the set of public predicates, or a '_' denoting the fact that all the predicates of the module are public.

5.2 Using a Module

The *use of a module* consists in the import of some predicates and some declarations of the module to the current one. The programmer can use a module by means of the `use_module/2` declaration where the first argument is the name of the module used, and the second argument is the list of imported predicates.

Importing predicates The second argument of the `use_module/2` declaration is a list of the form `[Functor/Arity, ...]` which indicates the set of imported predicates. By using this declaration the programmer states that every imported predicate can be considered as a predicate of the current module. Hence it is possible to declare as public an imported predicate in the `module/2` declaration. In order to guarantee the implementation hiding, only public predicates can be imported. It is forbidden to import a predicate with the same name/arity as a local predicate or a previously imported predicate. If the programmer wants to use more that one predicate with the same name/arity (defined in different modules), at most one must be imported and a complete qualification call of the form `Module:p(Arg_1, ..., Arg_n)` must be used for the other ones. It is also possible to omit the second argument of the `use_module` declaration, in such a case the system assumes that all public predicates of the used module are imported.

Importing Declarations By using the `use_module/2` declaration, some declarations are automatically imported. It is especially the case of syntax declarations. Unlike the import of predicates, the import of declarations are not parameterized, all syntax modifications are imported.

5.3 Visibility

The formal semantics proposed in Sect. 2 and 3 (or 4) clearly define the visibility of predicates from a particular module. The predicates which can be called from a module are the predicates defined in that module plus all public predicates of imported modules. A call is either a module-qualified predicate (in the form `module:p(X1,..., XN)`) or a non-qualified predicate. If a not fully qualified predicate occurs, the compiler assumes that this call is implicitly made inside the module. In other words, in the module `module` the predicate `p(X_1, ..., X_n)` stands for `module:p(X_1, ..., X_n)`. In the same spirit, the predicate `call(X)` in the module `Module` is a shortcut for `call(Module, X)`.

For other features not modeled in our semantics, such as global variables [8] or attributes [13], clause assertions/retractions etc., we propose a very simple but radical solution : by default only the local work space (work space of the current module) is visible from outside. In other words using such features can only be done inside the module itself. Hence to allow the access to global variables or attributes or dynamic clauses from outside (in order to consult or modify), the programmer must create accessors declared public in the interface.

Nevertheless, in order to simplify the life of programmers who do not want to protect their own modules and do not want to be disturbed by such controls, it is possible to make accessible the work space of any module by declaring it public with the `public/0` directive. For instance, to assert a clause from a module `module1` in a `module2`, one uses a call of the form `module2:assertz(p(T1, ..., TN)`.

5.4 Closures

In order to create and manipulate closures, two operators have been added: `closure/3` and `apply/2`. The predicate `closure(X,G,C)` unifies `C` with the closure made from the goal `G` (supposed to be instantiated at compile-time), in which the variable `X` is abstracted, whereas `apply(C, T)` replaces in the closure `C` the abstraction variable by the term `T` and executes the resulting goal. In practice the system deals with slightly more general closures than those defined in Sect. 2, for instance a goal instead of a predicate can be given as second argument of the `closure/3` operator.

5.5 The Default Module

In our module system we allow the compilation of classical Prolog files without module declaration. The code belonging to such files is considered to belong to the special module `user`. This module is public and access to this module is never controlled by the system. Our system is thus able to execute old GNU-Prolog files. Moreover this module is the default module of every call made from the top level.

5.6 Packages

Despite the fact that the module system we have defined strongly limits conflicts of predicate names, its distributed use leads quickly to a new conflict problem : the conflict of module names. To avoid that, a notion of package similar to the Java language has been introduced. A package is defined as a sequence of atoms separate by slashes. The notion of module name is extended to an atom or a pair package - atom separated by a slash.

For sake of conciseness, the directive `import/1` has been added to import module names. For example, the use of `:-import(oefai/clpr)` indicates to the system that the atom `clpr` refers to the module `clpr` belonging to the package `oefai`. In the same way, with `:-import(oefai/_)`, the user can import all modules of the package `oefai`. The import of two modules with the same name is however forbidden.

5.7 About the Implementation

The actual implementation of our module system consists of a layer, written in Prolog, on the top of GNU Prolog RH [9], a version of GNU Prolog extended with coroutines and attributed variables. This layer is composed of:

– a preprocessor that converts modular code into non-modular code ;
– a library to handle dynamic calls ;
– a new top-level, that transparently interprets modular programs and is able to dynamically compile and load modules and their dependencies.

Attributed variables are used to represent closures, and are useful to protect them from a "reverse engineering" approach, as they forbid the user to hack the internal representation of closures. This is the only reason to use attributed variables in the implementation of the module system.

6 Comparison with other Module Systems

In this subsection, we present a brief overview of existing syntactic module systems and show how they deal with the problem of high-order.

SICStus Prolog. In SICStus [27], all predicates are public (formally for all module $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P}$ we have $I_\mu = \Sigma_P$). On the other hand, this system does not provide any kind of code protection. Thanks to the `meta_predicate` directive, that allows to explicitly declare meta-predicates, the system is able to implicitly add the calling context to meta-data. The principal defect of this approach is thus the abandonment of the principle of implementation hiding.

ECLiPSe. ECLiPSe [1] proposes an intermediary solution. In fact it provides two different predicates to invoke meta-calls. The first one `:/2` behaves as `call/2` (defined in Sect. 3) whereas the second one `@/2` does not make any permission test. It provides also a directive `tool/1`, (analogous to the `meta_predicate` directive) that allows the implicit addition of the calling context as argument of the meta-predicate. Hence for a common use the programmer employs `:/2` and `@/2` for meta-predicates. This solution has the advantage of limiting the unauthorized calls made in a unconscious way. Nonetheless, as beforehand, it is not possible to ensure the hiding of any predicate. It can be noticed that this proposal is very close to the ISO one [16].

SWI Prolog. For meta programming, SWI Prolog [29] uses a slightly different semantics from the one proposed in Sect 3. In fact, SWI handles two distinct calling contexts. The former, that we call *static context*, plays the role of the calling context in a modular CSLD transition. The latter, that we call *dynamic context*, plays the role of the calling context in a call transition. By default the dynamic context is the same that the static context. However, for a meta-predicate, the dynamic context is the context of the goal that calls it. In SWI, predicates with this behavior are called "module transparent" and are declared with the directive `module_transparent/1`. By declaring `forall/2` as "module transparent", SWI-Prolog executes the previous example as expected.

Nonetheless, a dynamic call does not behave as a static one. Therefore with a "module transparent" predicate, the two goals `p(x)` and `(G=p(x), call(G))` do not call `p/1` in the same module. The former makes the call in the static context, whereas the latter makes it in the dynamic context.

Moreover, these conventions do not provide a code protection as strong as we want. For instance, the use of the implementation of the `findall/3` predicate (defined in the example 1) does not work better either in SWI, all assertion/retraction being made in the calling module instead of the called module.

CIAO Prolog. Conceptually, CIAO Prolog [3] adopts a closed module system. In order to allow the manipulation of meta-data through the module system, it provides an advanced version of the directive `meta_predicate/1`. Before calling the meta-predicates, the system compiles on the fly meta-data following a method analoguous to the one presented in Sect. 4. Since this compilation is done before the call of the meta-predicate, the system is knows the calling context in which the meta-data must be called, to obtain the expected result. The meta-predicate handles then a compiled version of the meta-data that is possible to invoke with the predicate `call/1`. As far as the system does not document any predicate (except `call/1`) able to create or manipulate such compiled meta-data, the implementation hiding property is preserved.

Although not clearly stated in the manual, CIAO Prolog does make a distinction between terms and higher-order data (i.e. compiled versions of goal). However, instead of hiding these objects behind a directive, we propose to manipulate closures as first-class citizens.

XSB The XSB system [23] is also considered as a syntactic system, but follows a quite different approach. In fact, this system belongs to the class of *atom-based* systems, rather than *predicate based*. The main difference is that XSB modularizes function symbols too. Hence two compound terms constructed in two different modules can not be unified. In a module, it is possible however to import public symbols from another module, the system considers then that the two modules share the same symbols. The semantics of the `call/1` predicate is hence very simple : the meta-call of a term corresponds to the call of the predicate of the same symbol and arity as the module where the term has been created.

However, this solutionmainly moves the problem to the construction of the terms. Indeed in XSB, the terms constructed with `=../2`, `functor/2` and `read/1` are supposed to belong to the module `user`. As a consequence, the system does not respect anymore the independence of the selection strategy. For instance, in a module different form `user`, the goal `(functor(X,foo,1), X=foo(_))` fails, whereas `(X=foo(_), functor(X,foo,1))` succeeds.

7 Conclusion

In a classical paper of D.H.D. Warren [28], the higher-order extensions of Prolog were questioned as they do not really provide more expressive power than meta-programming predicates. We have shown here that the situation is different in the context of logic programs with modules, and that the protection of private module predicates necessitates to distinguish between term calls (meta-programming) and closures (higher-order).

The module system we propose is close to the one of CIAO Prolog in its implementation, but has the advantage of revealing the need for closures, and of positioning them w.r.t. meta-programming predicates in the framework of formal operational semantics. Furthermore, an equivalent logical semantics has been provided for pure modular programs.

Our module system has been implemented in GNU-Prolog. Some existing code has been ported as libraries using the module system, such as for instance an implementation of CHR [26]. This modularization of CHR provides an example of intensive use of the module system allowing the development of several layers of constraint solvers in CHR. Other libraries have been developed with this closed module system for the development of SiLCC. One can quote a dynamic lexer-parser allowing modular redefinitions of the syntax with a powerful generalization of the directive `op/3`. This library belongs to the bootstrap libraries of our on-going implementation of SiLCC.

As for future work, we can mention the possible implementation of so-called *functors* [18] allowing the static instantiation of parameterized modules, and the investigation of object-oriented programming features with modules and global variables in CLP.

Acknowledgements. We are grateful to Sumit Kumar for a preliminary work he did on this topic, during his summer 2003 internship at INRIA. We thank also Emmanuel Coquery and Sylvain Soliman for valuable discussion on the subject. and to Daniel de Rauglaudre and Olivier Bouissou for their comments and help in the implementation.

References

1. Abderrahamane Aggoun and al. *ECLiPSe User Manual Release 5.2*, 1993 – 2001.
2. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In *META-92: Third International Workshop on Meta Programming in Logic*, pages 105–119, Berlin, Heidelberg, 1992. Springer-Verlag.
3. F. Bueno, D. Cabeza Gras, M. Carro, M. V. Hermenegildo, P. Lopez-Garca, and G. Puebla. The ciao Prolog system. reference manual. Technical Report CLIP 3/97-1.10#5, University of Madrid, 1997-2004.
4. Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programmming*, 19/20:443–502, 1994.
5. Daniel Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System*. PhD thesis, Universidad Politécnica de Madrid, August 2004.
6. Daniel Cabeza and Manuel Hermenegildo. A new module system for Prolog. In *First International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 131–148. Springer-Verlag, July 2000.
7. Weidong Chen. A theory of modules based on second-order logic. In *The fourth IEEE. Internatal Symposium on Logic Programming*, pages 24–33, 1987.
8. Daniel Diaz. *GNU Prolog user's manual*, 1999–2003.
9. Daniel Diaz and Rémy Haemmerlé. *GNU Prolog RH user's manual*, 1999–2004.

10. Pierre Deransart Adbelali. Ed-Dbali and Laurent. Cervoni. *Prolog: The Standard.* Springer-Verlag, New York, 1996.
11. François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, February 2001.
12. Patrica M. Hill. A parameterised module system for constructing typed logic programs. Technical Report 93.12, University of Leeds, Mar 1993.
13. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. Rapport Technique TR-92-23, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1992.
14. C. Holzbaur. Oefai clp(q,r) manual rev. 1.3.2. Technical Report TR-95-09, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1995.
15. International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 1: General core*, 1995. ISO/IEC 13211-1.
16. International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 2: Modules*, 2000. ISO/IEC 13211-2.
17. Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.
18. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
19. Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.
20. Luís Monterio and António Porto. Contextual logic programming. In *Proceedings of ICLP'1989,International Conference on Logic Programming*, pages 284–299, 1989.
21. Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language.* PhD thesis, Department of Informatics, University of Beira Interior, Portugal, September 2003.
22. Richard A. O'Keefe. Towards an algebra for constructing logic programs. In *Symposium on Logic Programming*, pages 152–160. IEEE, 1985.
23. Konstantinos Sagonas and al. *The XSB System Version 2.5 - Volume 1: Programmer's Manual*, 1993 – 2003.
24. D. T. Sannella and L. A. Wallen. a calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, pages 147–177, 1992.
25. Vijay A. Saraswat and Patrick Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.
26. Tom Schrijvers and David S. Warren. Constraint handling rules and table execution. In *Proceedings of ICLP'04, International Conference on Logic Programming*, pages 120–136, Saint-Malo, 2004. Springer-Verlag.
27. Swedish Institute of Computer Science. *SICStus Prolog v3 User's Manual.* The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 1991–2004.
28. David H. D. Warren. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence*, volume 10 of *Lecture Notes in Mathematics*, pages 441–454. 1982.
29. Jan Wielemaker. *SWI Prolog 5.4.1 Reference Manual*, 1990– 2004.

A Cardinality General Normal Form (CGNF) in propositional logic including n-ary CSPs.

Lionel PARIS, Belaïd BENHAMOU, and Pierre SIEGEL

Université de Provence
LSIS - UMR CNRS 6168,
Marseille, FRANCE
{Lionel.Paris,Belaid.Benhamou,Pierre.Siegel}@cmi.univ-mrs.fr

Abstract. We investigate in this work a generalization of the known *CNF* representation which allows an efficient boolean encoding for n-ary CSPs. We show that the space complexity of the boolean encoding is identical to the one of the classical CSP representation and introduce a new inference rule whose application until saturation achieves arc-consistency in a linear time complexity for n-ary CSPs expressed in the boolean encoding. Two enumerative methods for the boolean encoding are provided: the first one (equivalent to MAC in CSPs) maintains full arc-consistency on each node of the search tree while the second (equivalent to FC in CSPs) performs partial arc-consistency on each node.

1 Introduction

Constraint solving is a well known framework in artificial intelligence. Mainly, two approaches are well used: the propositional calculus holding the satisfiability problem (SAT) and the formalism of discrete constraint satisfaction problems (CSPs).

Methods to solve the SAT problem in propositional calculus are former than the CSP ones. They have been widely studied for many years since the Davis and Putnam (DPLL for abbreviation) procedure [1, 2] was introduced, and still a domain of investigation of a large community of researchers. Several improvements of the DPLL method are provided recently [3–5]. These new methods are able to solve large scale of SAT instances and are used to tackle real applications. The advantage of SAT methods is their flexibility to solve any constraints given in the CNF form. They are not sensitive to constraint arity[1] as it is often the case for CSP methods. One drawback of the CNF formulation is the loss of the problem structure which is well represented in CSP formalism.

The discrete CSP formalism was introduced by Montanari in 1974 [6]. The asset of this formalism is its capability to express the problem and explicitly represent its structure as a graph or a hyper-graph. This structure helps to achieve constraint propagation and to provide heuristics which render CSP resolution methods efficient. In comparison to the propositional calculus, the CSP resolution methods are sensitive to constraint

[1] The arity of a constraint is its number of involved variables

arity. Most of the known methods [7] apply on binary[2] CSPs. This results in a considerable restriction since most of the real problems (see CSPlib[3]) need constraints of unspecified arity for their natural formulation. To circumvent this restriction problem, recent works provide methods to deal with more general constraints [8].

Both SAT and CSP formulations are closely linked. Several works on transformation of binary CSP to SAT forms exist [9]. The transformation in [10] is generalized to non-binary CSPs in [11]. Transformations of a SAT form to a CSP form are described in [12–14]. Unfortunately, most of the transformations from CSP to SAT result in an overhead of space and lose the problem structure. Both factors combined may slow the resolution of the problem. Our aim in this paper is to provide a more general boolean representation which includes both the CNF and the CSP formulations and which preserves their advantages. That is, a representation which does not increase the size of the problem and which keeps the problem structure. We show particularly how non-binary CSPs are well expressed and solvable in this new formulation. We provide two enumerative methods for the general boolean formulation which are based on the DPLL procedure. To enforce constraint propagation we implement a new inference rule which takes advantage of the encoded structure. The application of this rule to the considered problem is carried with a linear time complexity. We will show that the saturation of this rule on the considered problem is equivalent to enforcing arc consistency on the corresponding CSP. We prove good time complexity to achieve arc consistency for non-binary CSPs. This allows to maintain efficiently full arc consistency at each node of the search tree of a first enumerative method which is equivalent to the known method MAC [15] in CSP. We also prove that a partial exploitation of this inference rule leads to a second enumerative method which is equivalent to a forward checking (FC) method for non-binary CSPs.

The rest of the paper is organized as follows: in section 2 we recall some background on both propositional calculus and discreet CSP formalisms. We introduce in section 3 a general normal form which we use to express both SAT and CSP problems. We define in section 4 a new inference rule which we use to show results on arc consistency. We describe in section 5 two enumerative algorithms (the FC and MAC versions) based on two different uses of the introduced inference rule. We summarize in section 6 some previous related works and discuss the interest of our method. Section 7 concludes the work.

2 Background

A CNF formula f in propositional logic is a conjunction $f = C_1 \wedge C_2 \dots C_k$ of clauses. Each clause C_i is itself a disjunction of literals. That is, $C_i = l_1 \vee l_2 \dots l_m$ where each literal l_i is an occurrence of a boolean variable either in its positive or negative parity. An interpretation I is a mapping which assigns to each boolean variable the value *true* or *false*. A clause is satisfied if at least one of its literals l_i is given the value *true* in the interpretation I ($I[l_i] =$ *true*). The empty clause is unsatisfiable and is denoted by $\Box$.

[2] A binary CSP contains only two-arity constraints
[3] http://csplib.org

The formula f is satisfied by I if all its clauses are satisfied by I, thus I is a *model* of f. A formula is satisfiable if it admits at least one model, otherwise it is unsatisfiable.

On the other hand a CSP is a statement $P = (X, D, C, R)$ where $X = \{X_1, X_2, \ldots, X_n\}$ is a set of n variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains where D_i is the current domain of possible values for X_i, $C = \{C_1, C_2, \ldots, C_m\}$ is a set of m constraints, where the constraint C_i is defined on a subset of variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_{a_i}}\} \subset X$. The arity of the constraint C_i is a_i and $R = \{R_1, R_2, \ldots, R_m\}$ is a set of m relations, where R_i is the relation corresponding to the constraint C_i. R_i contains the permitted combinations of values for the variables involved in the constraint C_i. A binary CSP is a CSP whose constraints are all of arity two (binary constraints). A CSP is non-binary if it involves at least a constraint whose arity is greater than 2 (a n-ary constraint). An *instantiation* I is a mapping which assigns each variables X_i a value of its domain D_i. A constraint C_i is satisfied by the instantiation I if the projection of I on the variables involved in C_i is a tuple of R_i. An instantiation I of a CSP P is consistent (or called a solution of P) if it satisfies all the constraints of P. A CSP P is consistent if it admits at least one solution. Otherwise P is not consistent. Both propositional satisfiability (SAT) and constraint satisfaction problems (CSPs) are two closely related NP-complete problems. For the sequel we denote by n the number of variables of the CSP, by m its number of constraints, by a its maximal constraint arity and by d the size of its largest domain.

3 An encoding including SAT and CSPs

The idea of translating a CSPs to an equivalent SAT form is first introduced by De kleer in [9]. He proposed the well known *direct encoding* and since that Kasif [10] proposed the *AC encoding* for binary CSPs. More recently Bessière et al [11] generalized the *AC encoding* to non-binary CSPs. Our approach is different, it consists in providing a general boolean form including both CNF (SAT) and CSP representations, rather than translating CSPs to SAT forms. We describe in this section, a new boolean encoding which generalizes the CNF formulation, and show how n-ary CSPs are naturally represented in an optimal way (no overhead in size in comparison to the CSP representation) in this encoding.

3.1 The generalized normal form (GNF)

A generalized clause C is a disjunction of boolean formulas $f_1 \vee \ldots \vee f_m$ where each f_i is a conjunction of literals, i.e $f_i = l_1 \wedge l_2 \wedge \ldots \wedge l_n$. A formula is in Generalized Normal Form (GNF) if and only if it is a conjunction of generalized clauses. The semantic of generalized clauses is trivial: the generalized clause C is satisfied by an interpretation I if at least one of its conjunctions f_i ($i \in [1, m]$) is given the value *true* in I, otherwise it is falsified by I. A classical clause is a simplified generalized clause where all the conjunctions f_i are reduced to single literals. This proves that GNF is a generalization of CNF. We show in the sequel that each constraint C_i of a given CSP is represented by a generalized clause.

We reach the optimal size representation by using the cardinality formulas $(\pm 1, L)$ which means "exactly 1 literals among those of the list L have to be assigned the value *true* in each model", to express efficiently that a CSP variable has to be assigned a single value in its domain. We denote by *CGNF* the *GNF* augmented by the cardinality.

3.2 The CGNF encoding for n-ary CSPs

Given an n-ary CSP $P = (X, D, C, R)$, first, we define the set of boolean variables which we use in the boolean representation, and two types of clauses: the domain clauses and the constraint clauses necessary to encode the domains and respectively the constraints.

- The set of boolean variables: as in all the other encodings we associate a boolean variable Y_v with each possible value v of the domain of each variable Y of the CSP. Thus, Y_v =*true* means that the value v is assigned to the variables Y of the CSP. We need exactly $\sum_{i=1}^{n} |D_i|$ boolean variables. The number of boolean variables is majored by nd.
- The domain clauses: let Y be a CSP variable and $D_Y = \{v_0, v_1, \ldots, v_k\}$ its domain. The cardinality formula $(\pm 1, Y_{v_0} \ldots Y_{v_k})$ forces the variable Y to be assigned to only one value in D_Y. We need n cardinality formulas to encode the n variable domains.
- The constraint clauses: each constraint C_i of the CSP P is represented by a generalized clause C_{C_i} defined as follows: Let R_i be the relation corresponding to the constraint C_i involving the set of variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_a}\}$. Each tuple $t_j = (v_{j_1}, v_{j_2}, \ldots, v_{j_a})$ of R_i is expressed by the conjunction $f_j = X_{v_{j_1}} \wedge X_{v_{j_2}} \wedge \ldots \wedge X_{v_{j_a}}$. If R_i contains k tuples $t_1, t_2, \ldots, t_k$ then we introduce the generalized clause $C_{C_i} = f_1 \vee f_2 \vee \ldots \vee f_k$ to express the constraint C_i. We need m generalized clauses to express the m constraints of the CSP.

As the domain clauses are encoded in $O(nd)$, the constraint clauses in $O(mad^a)$, then the *CGNF* encoding of a CSP P is in $O(mad^a + nd)$ in the worst case. In fact, it is in $O(mad^a)$ since nd is often negligible. This space complexity is identical to the one of the original CSP encoding. This justifies the optimality in space of the *CGNF* encoding. Authors in [11] gave a spatial complexity in $O(mad^a)$ in the worst case for the k-AC encoding. As far as we understand, this complexity is rounded and does not take into account neither the at-least-one nor the at-most-one clauses. This increases the complexity to $O(mad^a + nd^2)$.

Example 1. Lets consider a constraint C of arity 3 concerning 3 CSP variables A, B and C with domains of size 3. (i.e. $D_A = D_B = D_C = \{1, 2, 3\}$). The constraint C is given by the table :

A	B	C
1	1	2
1	2	2
1	2	3
2	1	3
3	1	1

If we express it in the *CGNF*:

- We have $3 \times 3 = 9$ boolean variables: $A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2$ and C_3.
- We have two types of clauses :
  - 3 domain clauses:
    $C_{D_A} = (\pm 1, A_1 A_2 A_3)$,
    $C_{D_B} = (\pm 1, B_1 B_2 B_3)$,
    $C_{D_C} = (\pm 1, C_1 C_2 C_3)$,
  - 1 constraint clause:

$$C_{C_{ABC}} = (A_1 \wedge B_1 \wedge C_2) \vee (A_1 \wedge B_2 \wedge C_2) \vee (A_1 \wedge B_2 \wedge C_3) \vee (A_2 \wedge B_1 \wedge C_3) \vee (A_3 \wedge B_1 \wedge C_1)$$

Definition 1. *Let P be a CSP and C its corresponding CGNF encoding. Let I be an interpretation of C. The corresponding equivalent instantiation I_p in the CSP P verifies the following condition: for all CSP variable X and each value v of its domain, $X = v$ if and only if $I[X_v]$ =true.*

Theorem 1. *Let P be a CSP, C its corresponding encoding, I an interpretation of the CGNF encoding and I_p the corresponding equivalent instantiation of I in the CSP P. I is a model of C if and only if I_p is a solution of the CSP P.*

Proof. Let I be a model of C and I_p its corresponding instantiation in P. I satisfies each clause of C, since it is one of its models. We have to prove that each CSP variable is assigned a single value in I_p and I_p satisfies all the constraints of P. A domain clause $C_{D_X} = (\pm 1, X_{v_1}\ X_{v_2} \ldots X_{v_d})$of C means that one and only one boolean variable among $\{X_{v_1}\ X_{v_2} \ldots X_{v_d}\}$ is true in I. By construction $X = v$ in I_p, the unity is guaranteed since I_p is a mapping. Each constraint clause $C_{C_i} = f_1 \vee f_2 \vee \ldots \vee f_m$, has at least one f_i satisfied in I. This means that the values assigned to the CSP variables involved in the constraint C_i associated to C_{C_i} form a tuple of the relation R_i. Thus, C_i is satisfied by I_p and satisfies all the constraints of P. I_p is a solution of P.
Now, we suppose I_p is a solution of P and shall prove that I is a model of C. Because I_p is a solution, each CSP variable X is assigned to a value v of its domain. $X = v$, then $I[X_v] = true$ and clearly the cardinality formula corresponding to D_X is satisfied by I. Beside, each constraint C_i of P is satisfied. The projection of I_p on the constraint C_i results in a tuple t_j of the relation R_i. This implies that the conjunction f_j of the constraint clause C_{C_i} corresponding to the tuple t_j is true in I , and the clause C_{C_i} is satisfied by I. Thus, I is a model of C.

The *CGNF* encoding allows an optimal representation of CSPs, however it does not capture the property of arc consistency which is the key of almost all the enumerative CSP algorithms. We introduce in the following a simple inference rule which applies on the *CGNF* encoding C of a CSP P and prove that arc consistency is achieved with a linear complexity by application of the rule on C until saturation.

4 A new inference rule for the CGNF encoding

The rule is based on the preserved CSP structure represented by both the domain and the constraint clauses.

4.1 Definition of the inference rule IR

Let P be a CSP, C its *CGNF* encoding, C_{C_i} a generalized constraint clause, C_{D_X} a domain clause, L_C the set of literals appearing in C_{C_i} and L_D the set of literals of C_{D_X}. If $L_C \cap L_D \neq \emptyset$ then we infer each negation $\neg X_v$ of a positive literal[4] appearing in L_D which does not appear in L_C. We have the following rule:
IR: if $L_C \cap L_D \neq \emptyset$, $X_v \in L_D$ and $X_v \notin L_C$ then $C_{D_X} \wedge C_{C_i} \vdash \neg X_v$[5].

Example 2. If we consider the data of Example 1, we have:

$$C_{D_B} = (\pm 1, B_1 B_2 B_3)$$

and

$$C_{C_{ABC}} = (A_1 \wedge B_1 \wedge C_2) \vee (A_1 \wedge B_2 \wedge C_2) \vee (A_1 \wedge B_2 \wedge C_3) \vee (A_2 \wedge B_1 \wedge C_3) \vee (A_3 \wedge B_1 \wedge C_1)$$

The application of the rule IR on both clauses gives: $C_{D_B} \wedge C_{C_{ABC}} \vdash \neg B_3$.

Proposition 1. *The rule IR is sound (correct).*

Proof. Let X_v be a literal appearing in L_D but not in L_C and I a model of $C_{C_i} \wedge C_{D_X}$. C_{C_i} is a disjunction of conjunctions f_i and each conjunction f_i contains one literal of L_D. At least one of the conjunctions f_i, say f_j is satisfied by I since I is a model of C_{C_i}. Thus, there is a literal $X_{v'}$ ($X_{v'} \neq X_v$ since $X_v \notin L_C$) of f_j appearing in L_D, such that $I[X_{v'}] =$*true*. Because of the mutual exclusion of of literal of C_{D_X}, the $X_{v'}$ is the single literal of L_D satisfied by I. Thus, $I[\neg X_v] =$*true* and I is a model of $\neg X_v$.

4.2 The inference rule and arc-consistency

A CSP P is arc consistent iff all its domains are arc consistent. A domain $D_{X_{i_1}}$ is arc consistent iff for each value v_{i_1} of $D_{X_{i_1}}$ and for each k-arity constraint C_j involving the variables $\{X_{i_1}, \ldots, X_{i_k}\}$, there exists a tuple $(v_{i_2}, \ldots, v_{i_k}) \in D_{X_{i_2}} \times \ldots \times D_{X_{i_k}}$ such that $(v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \in R_j$. We use the inference rule IR on the *CGNF* encoding C of a CSP P, to achieve arc-consistency. We show that by applying IR on C until saturation (a fixed point) and by propagating each inferred negative mono-literals we maintain arc-consistency on C. Since, this operation is based on unit propagation, it can be done in a linear time complexity.

Proposition 2. *Let P be a CSP and C its CGNF encoding. A value $v \in D_Y$ is removed by enforcing arc-consistency on P iff the negation $\neg Y_v$ of the corresponding boolean variable is inferred by application of IR to C.*

---

[4] The CGNF encoding of a CSP contains only positive literals.
[5] $\vdash$ denotes logical inference.

Proof. Let v be a value of the domain D_Y which does not verify arc-consistency. There at least one constraint C_j involving Y such that v does not appear in any of the allowed tuples of the corresponding relation R_j. The boolean variable Y_v does not appear in the associated constraint clause C_{C_j}, but it appears in the domain clause C_{D_Y} associated to D_Y. By applying the the rule IR on both C_{D_Y} and C_{C_j} we infer $\neg Y_v$.
The proof of the converse can be done in the same way.

Theorem 2. *Let P be a CSP and C its CGNF encoding. The saturation of IR on C and the propagation of all inferred literals is equivalent to enforcing arc-consistency on the original CSP P.*

Proof. Is a consequence of proposition 2.

4.3 Arc-consistency by application of IR

To perform arc consistency on C by applying IR, we need to define some data structures to implement the inference rule. We suppose that the nd boolean variables of C are encoded by the first integers $[1..nd]$. We define a table OCC_j of size nd for each constraint clause C_{C_j} such that $OCC_j[i]$ gives the number of occurrences of the variable i in C_{C_j}. There is a total of m such tables corresponding to the m constraint clauses. If $OCC_j[k] = 0$ for some $k \in \{1 \dots nd\}$ and some $j \in \{1 \dots m\}$ then the negation $\neg i$ of the boolean variable i is inferred by IR. This data structure adds to the space complexity a factor of $O(mnd)$. The total space complexity of C is $O(mad^a + nd + mnd)$, but the factors nd and mnd are always lower than the mad^a factor, and the space complexity of C in remains $O(mad^a)$.

The principle of the arc-consistency method consists first in reading the m tables OCC_j to detect the variables $i \in [1 \dots nd]$ having a number of occurrences equal to zero ($OCC_j[i] = 0$). This is achieved by the steps 3 to 7 of Algorithm 1 in $O(mnd)$, since there are m tables of size nd. After that we apply unit propagation on the detected variables, and propagate the effect until saturation (i.e no new variable i having $OCC_j[i] = 0$ is detected). The procedure of arc-consistency is sketched in the Algorithm 1. This procedure calls the procedure *Propagate* described in algorithm 2.

The complexity of the arc consistency procedure is mainly given by the propagation of the effect of literals of the list L (lines 8 to 11 of Algorithm 1). It is easy to see that in the worst case there will be nd calls to the procedure $Propagate$. All the propagations due to the previous calls are performed in $O(mad^a)$ in the worst case. Indeed, there are at most d^a conjunctions of a literals for each constraint clause of C. The total number of conjunctions treated in line 5 of Algorithm 2 can not exceed md^a since each considered conjunction f in line 4 is suppressed in line 5 (f is interpreted to false). As there is a literals by conjunction f, the total propagation is done in $O(mad^a)$. Thus, the complexity of the arc consistency procedure is $O(mad^a + mnd)$. But the factor (nmd) is often lower than mad^a and the total complexity is reduced to $O(mad^a)$. It is linear *w.r.t* the size of C.

Algorithm 1 Arc consistency

Procedure Arc_consistency (A CGNF instance C)
var L : list of literals
$L = \emptyset$
  for each constraint clauses C_{C_j}
    for each literal i of OCC_j
      if $OCC_j[i]$ = 0 **then** add $\neg i$ in L **endif**
    endfor
  endfor
  while $L \neq \emptyset$ and $\Box \notin C$
    extract $\neg l$ from L
    propagate (C, $\neg l$, L)
  endwhile

Algorithm 2 Propagate

Procedure Propagate (A CGNF instance C, literal i, List L)
if i is not yet assigned
then
  assign i the value $false$
  for each unassigned conjunction f of C containing i
    assign f the value false
    for each literal j of f such that $i \neq j$
      withdraw 1 to $OCC_k[j]$
      (k is the label of the constraint clause containing f)
      if $OCC_k[j] = 0$ **then** add j to L **endif**
    endfor
  endfor
endif

5 Two enumerative methods for the CGNF encoding

We study in the following two enumerative methods: the first one (MAC) maintains full arc consistency during the search while the second (FC) maintains partial arc consistency as does the classical Forward Checking method in CSPs [7]. Both methods perform boolean enumeration and simplification. They are based on an adaptation of the DPLL procedure to the *CGNF* encoding.

5.1 The MAC method

MAC starts by a first call to Algorithm 1 to verify arc consistency at the root of the search tree. It then calls the procedure propagate described in algorithm 2 at each node of the search tree to maintain arc consistency during the search. That is, the mono-literals of each node are inferred and their effects are propagated. The code of *MAC* is sketched in Algorithm 3.

Algorithm 3 *MAC* algorithm for CGNF encoding.

Procedure MAC(A CGNF instance C)
Arc consistency(C)
if $\Box \in C$ **then return**(unsatisfiable)
else begin
    choose a literal $l \in C$
    if $Satisfiable(C, l, true)$ **then return**(satisfiable)
    else if Satisfiable$(C, l, false)$ **then return**(satisfiable)
        else return(unsatisfiable)
end

Algorithm 4 Function Satisfiable.

Function Satisfiable(A CGNF instance C, variable l, boolean val):boolean
var L : list of literals
$L = \emptyset$
if $val = true$ **then begin**
    assign l the value *true*
    Add each literal $i \neq l$ of the domain clause containing l in L
    while $L \neq \emptyset$ and $\Box \notin C$
        extract i from L
        propagate (C, i, L)
    end
else do $Propagate(C, l, L)$ **while** $L \neq \emptyset$ and $\Box \notin C$
if $C = \emptyset$ **then return**(true)
else if $\Box \in C$ **then return**(false)
        else begin
            Chose a literal p of C
            if $Satisfiable(C, p, true)$ **then return**(true)
            else if $Satisfiable(C, p, false)$ **then return**(true)
                else return(false)
            end.

5.2 The FC method

It is easy to obtain from MAC a *Forward Checking (FC)* algorithm version for the CGNF encoding. The principle is the same as *MAC* except that instead of enforcing full arc-consistency at each node of the search tree, (FC) does it only on the near neighbors of the current variable under assignment. This is done by restricting the propagation effect of the current literal assignment to only the literals of the clauses in which it appears. It is important to see that finding the *FC* version in our encoding is trivial, whereas it is not the case for n-ary CSP where six versions of *FC* are provided [8].

5.3 Heuristics of literal choice

Because the CGNF encodings keeps the CSP structure, we can easily find equivalent heuristics to all the well known CSP variable/value heuristics. For instance, the minimal domain heuristic (MD) which consists in choosing during the search the CSP variable whose domain is the smallest is equivalent to select in the CGNF encoding, a literal appearing in the shortest domain clause. An other example is the *DomDeg* heuristic which consists in choosing the variable which minimizes the ratio $\frac{|D_i|}{degree(X_i)}$ such that $i \in [1..n]$ and where $degree(X_i)$ gives the number of occurrences of the variables i in all the constraints of the CSP. In the CGNF encoding, this heuristic consists in choosing the variable which minimize the value $\frac{|C_{D_i}|}{occ(v_k)}$ such that $i \in [1..n]$, $v_k \in C_{D_i}$ and where $occ(v_k)$ gives the number of occurrence of the literal v_k in all the constraint clauses.

6 Related works, discussion and comparison

We summarize in the following some well known translations of CSPs to SAT and compare their properties.

– *The direct encoding* [9] is the most used translation of CSPs into SAT. It use the same set of boolean variable than our encoding. These boolean variables appear in three types of clauses: the At-least-one clauses, the At-most-one clauses, and the Conflict clauses. Given a CSP P, its direct encoding has a worst case space complexity of $O(nd + \frac{nd(d-1)}{2} + mad^a)$ and does not capture arc consistency.
– *The AC encoding* [10] is defined for binary CSPs. Its particularity is the representation of the arc consistency property in the encoding. It allows to a SAT procedure to achieve arc consistency by applying only unit propagation. This encoding differs from the direct encoding by only the conflict clauses which are replaced by the support clauses. Its space complexity in the worst case is the same as the direct encoding.
– *The k-AC encoding* [11] is a generalization of the AC encoding to non-binary CSP. It can represent supports on a subset S of variables of any size involved in a constraint C, for the instantiation of another variable subset T of any size of the same constraint C, rather than only supports of a single variable on another single variable as in the AC encoding. Its space complexity is identical to the ones of both previous encodings, and slightly greater than the one of the CGNF encoding which is in $O(nd + mad^a)$. The k-AC encoding captures a larger family of consistency than both the AC and the CGNF encodings. That is, it encodes the *relational k-arc-consistency*.
– *Comparison and discussion:* Our approach is different from these previous works. It consists in a generalization of the CNF representation which allows a natural and optimal encoding for n-ary CSPs keeping the CSP structure. Both SAT and CSP advantages are available and can be used in the new encoding. The purpose of this paper is to introduce a new promising method for solving n-ary CSPs. For the moment, we only implemented a basic DPLL procedure exploiting our inference rule. No optimization (such as *non-chronological back-tracking*, or *clause recording*)

have been added, but we think they can be. The preliminary results we obtained seem to be promising. The solver seems to behave in the same order of time as the nFCs for n-ary CSPs given in [8] (it has to be verified). We think that solving a n-ary CSP with our method would take a little bit more time that solving it by applying a sophisticated solver like Chaff [4] on its k-AC encoding (this is still to verify too), as long as we won't integrate the optimizations.

7 Conclusion

We studied a generalization of the known *CNF* representation which allows efficient boolean encodings for n-ary CSPs. We showed that the size of the CSP boolean encoding is identical to the one of its original representation and proved that arc-consistency is performed with a linear time complexity when using the boolean encoding. We implemented a new inference rule whose application until saturation achieves arc-consistency for n-ary CSPs expressed in the boolean encodings. The main point is that it can be used for constraints of any arity. This brings to two enumerative methods for solving n-ary CSPs: the first one (MAC) maintains full arc-consistency on each node of the search tree while the second (FC) performs partial arc-consistency. Both methods are well known in CSPs and are found easily in our boolean encoding. In future, this work will be completed by a strong experimentation and code optimization. We could also try to extend the Inference Rule to capture other families of consistancies like k-consistancy (still in this boolean encoding). An other point is to look for polynomial restrictions of the boolean encoding. Finally, detecting and breaking symmetries in the boolean encodings will increase the efficiency of the defined enumerative methods.

References

1. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of ACM 7 (1960) 201–215
2. Davis, M., Loveland, G.: A machine program for theorem proving. Comm. ACM 5 (1962) 394–397
3. Silva, J., Sakallah, K.: Grasp – a new search algorithm for satisfiability. (1996) 220–227
4. Malik, S., Zhao, Y., Madigan, C.F., Zhang, L., Moskewicz, M.W.: Chaff: Engineering an efficient sat solver. (2001) 530–535
5. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat solver. (2002) 142–149
6. Montanari, U.: Networks of constraints : Fundamental properties and application to picture processing. Journal Inform. Sci. **9** (1974) 95–132
7. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313
8. Bessière, C., Meseguer, P., Freuder, E.C., J.Larrosa: On forwardchecking for non-binary constraint satisfaction. Journal of Artificial Intelligence **141** (2002) 205–224
9. Kleer, J.D.: A comparison of atms and csp techniques. (1989) 290–296
10. Kasif, S.: On the parallel complexity of descrete relaxation in constraint satisafaction networks. Journal of Artificial Intelligence **45** (1990) 275–286
11. Bessière, C., Hebrard, E., Walsh, T.: Local consistency in sat. (2003) 400–407
12. Bennaceur, H.: The satisfiability problem regarded as constraint satisfaction problem. (1996) 155–159

13. Walsh, T.: Sat v csp. (2000) 441–456
14. Russel, O.: An other sat to csp conversion. (2004) 558–565
15. D. Sabin, E.F.: Understanding and improving the mac algorithm. (1997) 167–181

Gomory cuts in a hybrid constraint programming approach

Andrea Lodi[1,2], Maria Silvia Pini[3], Francesca Rossi[3]

[1] DEIS, University of Bologna, viale Risorgimento 2 - 40136 Bologna - Italy
[2] IBM T.J. Watson Research, P.O. Box 218, Yorktown Heights, NY 10598 - USA
alodi@us.ibm.com
[3] Department of Pure and Applied Mathematics, University of Padova, Italy.
{frossi,mpini}@math.unipd.it.

Abstract. Recently, many successful hybrid approaches which use both constraint programming (CP) and operations research (OR) techniques to solve a problem have been proposed. They usually use OR techniques, like cutting planes, which are specific to the problem. In this paper we investigate the use of generic cutting planes (more specifically, Gomory cuts) in a typical CP-OR hybrid approach. This allows us to have a general algorithm to be applied to all problem classes. We will show that our approach is indeed promising: on the class of partial latin square completion problems, the performance of our algorithm is comparable to that of a pure CP algorithmand better in several cases. In particular we will show that the hybrid approach solves many problem instances that the CP approach is not able to solve in a reasonable amount of time.

1 Introduction

Constraint Programming (CP) [5] and Operations Research (OR) [13] techniques are both suitable for solving combinatorial search problems. However, they have complementary advantages and drawbacks. On one hand, CP offers more sophisticated modelling tools like global constraints [15], and effective search tree pruning techniques like constraint propagation [5]. On the other hand, OR provides better tools for reasoning about optimality via the possibility of considering relaxed problems, and other pruning facilities like the cutting planes within a branch and cut strategy.

To overcome the limitations and maintain the advantages of both kinds of techniques, recently many hybrid approaches have been proposed [7–10, 12] for solving combinatorial optimization problems. Some of them are based on interleaving a CP depth-first search algorithm with problem relaxation and cutting planes. More precisely, at each step in a depth-first search, the current problem is relaxed and solved optimally via the cutting planes. The optimal solution of the relaxed problem is then used to go back to the CP side with useful information for the strategy to choose the next step (that is, the next variable-value selection).

Some existing methods which employ this approach use cutting planes which are specific to the classes of problems considered. For example, in [8] Travelling Salesmen Problems are relaxed via cutting planes which achieve subtour elimination.

In this paper, we investigate the use of generic cuts in a hybrid setting. The reason for doing this is that the resulting algorithm would be general and thus applicable to all classes of problems. In particular, we consider Gomory cuts to investigate their convenience in the hybrid approach. It was folklore knowledge that Gomory cuts were not useful. In [2] it is shown that the way they have been always applied was wrong. Our goal is to see whether they can be useful in a hybrid CP-OR environment.

To achieve our goal, we develop an experimental setting where we consider the completion of the partial latin square problems and we compare the performance of a hybrid CP-OR algorithm with Gomory cuts, a hybrid CP-OR algorithm without Gomory cuts, and a pure CP algorithm. We perform several types of experiments by varying both the parameters of the problems and the features of the hybrid algorithm.

We will show that the hybrid CP-OR algorithm we propose in this paper, which uses Gomory cuts in the OR part, is always better than the hybrid approach without Gomory cuts and for several problem instances outperforms the pure CP approach. Moreover, it provides the high-level modelling facilities typical of CP, like global constraints. Such modelling and performance considerations thus make this hybrid approach more attractive than the pure CP approach.

More precisely, while there is no algorithm which is uniformly better on all problem instances, our hybrid algorithm with Gomory cuts looks convenient for critically constrained problems (where it is better than CP in several instances) or for problems whose constrainedness cannot be known in advance. This algorithm thus combines the modelling capabilities of CP (with compactness given by global constraints), together with the convenient branch and cut strategies, typical of OR approaches. Moreover, it does not need to use domain-specific cuts.

We therefore claim that it can be feasible and convenient to use suitable generic cutting planes, like Gomory cuts, within a CP-OR hybrid approach. It remains to be checked whether other generic classes of cutting planes can be useful in a hybrid CP-OR approach, and whether the same behavior holds also for other classes of problems.

The article is organized as follows: Section 2 gives the main concepts of CP, OR, and hybrid CP-OR approaches, Section 3 introduces the algorithms we will consider in this paper, Section 4 defines the experimental setting, Section 5 gives the experimental results and discusses them, and Section 6 presents some related works. Finally, Section 7 concludes the paper by summarizing its main results.

2 Background

In this section we will introduce the basic notions about Constraint Programming and Operations Research that will be used in our algorithms.

2.1 Constraint programming

In our context, a constraint is a relation among some variables with a finite domain. A constraint problem is just a set of variables, plus a set of constraints over subsets of them.

Constraint programming [5] is a body of techniques to solve constraint problems. For the purpose of this paper, we only consider CP techniques based on depth-first search and constraint propagation.

Depth-first search orders the variables and instantiates them one at a time in its domain, in a way that is compatible with the constraints and the already instantiated variables. If a variable cannot be assigned any value, backtracking occurs and the algorithm tries to instantiate the previous variable with another value. When all variables are instantiated, we have a solution of the given problem. If not all variables are instantiated and no more backtracking is possible, the problem has no solution.

Constraint propagation propagates information from one constraint to another one. Each propagation step considers a small part of the constraint problem (most of the times just one constraint), and eliminates from the variable domains of such part some domain elements, which are recognized to be incompatible to the considered constraints. Then, the new domains (possibly smaller) are used when considering adjacent constraints. This is repeated until quiescence. For example, arc-consistency [5] considers one constraint at a time, and eliminates from the domains of its variables those values which do not have any support in the domains of the other variables according to this constraint. The new problem obtained at the end of this process is equivalent to the original one (that is, it has the same set of solutions) but with possibly smaller domains. Achieving arc-consistency requires time $O(ed^2)$ where e is the number of constraints and d is the size of the domains. In general, performing constraint propagation where each step considers a subpart with at most k variables requires $O(n^k)$ time.

Since depth-first search in the worst case has to perform as many tries for each variable as the cardinality of its domain, the smaller the domains the better it is for such search technique. Thus all depth-first search constraint programming algorithms use constraint propagation after every variable instantiation. This usually allows for much pruning within the search tree. The overall complexity of the search algorithm remains exponential in the size of the problem, but the average complexity can be improved significantly by the use of constraint propagation.

Global constraints [3, 15] are non-binary constraints which occur very often in real-life problems. For this reason, they have been equipped with a compact way to express them and specialized constraint propagation techniques which exploit the semantics of the global constraint and have a higher pruning power with respect to arc-consistency. A typical example is `alldifferent(`X_1`,...,`X_n`)` which states that variables X_1,..., X_n must have different values. If this constraint were modelled by a set of $n(n-1)/2$ inequalities of the form $X_i \neq X_j$, and arc-consistency would be applied, no domain pruning would be achieved, except in the very special case in which some domain has just one element. On the contrary, by modelling it as a global constraint, and applying its specialized propagation algorithm, we can perform a global reasoning on all the n variables, for example by comparing the cardinality of the domains to n, and thus achieve more pruning.

Constraint problems can contain both global constraints and other kinds of constraints. During search, arc-consistency (or another generic constraint propagation algorithm) will be applied to all the constraints, except to global constraints for which their specialized propagation algorithm will be used.

2.2 Cutting planes and relaxed problems in OR

An Integer Linear Programming (ILP) problem is a set of linear equalities and inequalities where variables must have integer values, plus a linear objective function, to be minimized.

The usual solving technique for ILP problems is called branch and bound [13]. As in CP, this technique involves visiting the nodes of a search tree. At each node, the branch strategy is decided on the basis of an optimal solution of a relaxed version of the current problem. More precisely, the current problem is relaxed by eliminating the requirement that variables have to be integer, such relaxed problem is solved optimally, and the solution found is used to decide the next search step. Since the relaxed problem does not contain all constraints of the original problem, the value of its optimal solutions is a lower bound of the value of any optimal solution of the original problem.

Usually a non-integer variable is chosen in the optimal solution of the relaxed problem, and it is made integer by imposing additional constraints. For example, if the optimal solution of the relaxed problem contains $X = 4.3$, we add $X \leq 4$ and $X \geq 5$ to the original problem and then we follow one of these two paths according to heuristics. When we find a solution, we continue visiting the search tree but first we add the constraint that the objective function has to be better than the value of the current solution.

Cutting planes [13] are linear inequalities of the form $\alpha x \leq \alpha_0$ which eliminate some non-optimal solutions. They are added to a relaxed version of the problem after an optimal solution is found. Thus they remove the optimal solution of the relaxed problem while not eliminating any optimal solution of the original problem. The optimal solution of the relaxed problem is used to compute the cutting planes. Adding the cutting planes allows us to obtain a new relaxed problem which is closer to the original problem and thus whose optimal solutions have values which are closer to the value of the optimal solutions of the original problem. Cutting planes can be iteratively added until the improvement of the lower bound given by the optimal solution of the relaxed problem is not cost-effective. At this point, a branching step is performed.

Gomory cuts are a class of cutting planes which can be computed by taking into consideration the values of the variables in a solution and their offset with respect to an integer. Gomory cuts are generic in the sense that they are applicable to all integer linear programming problems. Other cutting planes, which are specific to the considered problems, can be used as well. For some classes of these cuts, there exist polynomial algorithms to find them.

2.3 Hybrid CP-OR approaches

Many hybrid approaches which try to combine the techniques which are typical of CP with those of OR have been proposed in the literature [7–10, 12]. For the purpose of this paper, we consider one which relates CP and ILP and can be described as follows:

1. The problem is modelled using CP tools. This allows to take advantage of the more sophisticated CP modelling environment. For example, global constraints can be used to easily model many real-time scenarios.
2. Constraint propagation is performed to reduce the variable domains. If we have a solution (that is, one value for each variable), we stop and return the solution.
3. The current problem is transformed into an ILP problem. Of course this can be done only if the CP problem does not contain constraints which cannot be modelled by an ILP. Many global constraints can be modelled within an ILP [14], and also many other classes of constraints. However, a CP problem could also contain, for example, quadratic constraints, which cannot be modelled in an ILP.
4. Given the current ILP, the integer constraint is relaxed, and the relaxed problem is solved with OR techniques, possibly using also cutting planes, that can be specific for the considered problem, to tighten the relaxation.
5. The optimal solution of the relaxed problem is used to decide the next variable to be instantiated in the CP side, and the value to which it has to be instantiated. The instantiation is then performed.
6. Go back to step 2.

Summarizing, this algorithm can also be seen as a CP search algorithm where the next variable-value instantiation is decided using information coming from an ILP environment.

3 Gomory cuts in a hybrid approach: our algorithms

In this section we will describe the algorithms for solving constraint optimization problems which we will consider in the following of the paper. In particular, we will use a pure CP algorithm, a pure ILP algorithm with and without Gomory cuts, and a hybrid CP-OR algorithm with and without Gomory cuts.

The pure CP algorithm we consider in this paper is a classical depth-first search algorithm with backtracking, which applies constraint propagation at each search step. At each node of the search tree, arc-consistency is performed and then an uninstantiated variable among those with the smallest domain is chosen and instantiated to its smallest value. In the following, this algorithm will be called `CP`.

The pure OR algorithm we use is a branch-and-cut algorithm where cutting planes are used to improve the solution of the relaxed problem. In the following, this algorithm will be called `ILP` if it does not use Gomory cuts, and `ILP-Gomory` if it uses them.

The hybrid algorithm with Gomory cuts follows the scheme described in Section 2.3, where arc-consistency is used to propagate constraints. The ILP version of the CP problem is obtained via the methodology in [14] (for the class of considered problems this step is always possible), it is solved via the primal (for the tree root) or dual (for the other nodes) Simplex algorithm, and Gomory cuts are applied to refine its relaxed optimal solution. The next branch step is perfomed by considering the variable whose decimal part is closer to 0.5, and by instantiating it to the closer integer. In the following, this algorithm will be called `H-Gomory`. The hybrid algorithm without Gomory cuts is like the one just explained above, except that it doesn't apply Gomory cuts. In the following, this algorithm will be called `H`.

4 Experimental setting

To compare the behavior of the above described algorithms, we have considered a specific class of problems: partial Latin Square completion [9]. These problems are easy to model but occur in a wide variety of real-life scenarios, from conflict-free wavelength routing in wide-area network, to statistical design, to error-correcting codes.

A Latin Square (LS) of order n is an $n \times n$ matrix such that each cell contains exactly one of the integers from 1 to n, and each of such integers appears at most once in any row and any column. A partial Latin Square (PLS) of order n is an LS of order n where some cells can be empty. A PLS can be completed if it is possible to fill the empty cells with integers so to obtain a LS.

A PLS completion problem is the problem of starting from a PLS and obtaining, if possible, one of its completions. The problem of deciding if a PLS can be completed is NP-complete [4].

In this section we briefly describe the problem encodings we considered.

In the CP framework, a PLS completion problem is modelled as a constraint satisfiability problem (CSP). More precisely, a PLS is modelled by using n^2 variables x_{ij}, corresponding to all the cells c_{ij} of the matrix $n \times n$. If a variable corresponds to an empty cell, its domain is $\{1, \ldots, n\}$, otherwise it is a singleton domain containing only one integer (the integer chosen for that cell). Moreover, we have an `alldifferent` constraint involving the variables in each row and each column. That is:

$$x_{ij} \in \{1, \ldots, n\} \; \forall i, j \in \{1, \ldots, n\}$$

$$x_{ij} = k \ \forall i,j \text{ such that } c_{ij} = k$$
$$\texttt{alldifferent}(x_{i1}, x_{i2}, \ldots, x_{in}) \ \forall i = 1, \ldots, n$$
$$\texttt{alldifferent}(x_{1j}, x_{2j}, \ldots, x_{nj}) \ \forall j = 1, \ldots, n$$

In the OR framework, the PLS completion is modelled as an Integer Linear Programming (ILP) problem, with an objective function to be maximized. The ILP model contains n^3 variables x_{ijk}, each corresponding to a cell c_{ij} with value k. A variable x_{ijk} has domain $\{0, 1\}$ if c_{ij} is empty. If c_{ij} has value k, then $x_{ijk} = 1$. The constraints model the row and column restrictions, and make sure that each cell has at most one value. Finally, the objective function, to be maximized, computes the number of variables which have a value different from 0, which is equal to the number of filled cells. That is:

$$\max \ \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_{ijk}$$

subject to

$$\begin{gathered}
\textstyle\sum_{i=1}^{n} x_{ijk} \leq 1 \ \forall j,k \\
\textstyle\sum_{j=1}^{n} x_{ijk} \leq 1 \ \forall i,k \\
\textstyle\sum_{k=1}^{n} x_{ijk} \leq 1 \ \forall i,j \\
x_{ijk} \in \{0,1\} \ \ \forall i,j,k \\
i,j,k = 1,2,...,n
\end{gathered}$$

If in the optimal solution the number of filled cells is equal to n^2, then the PLS can be completed, i.e., a feasible solution exists.

In the algorithm `CP` we have used the CSP formulation to model the completion PLS, whereas in the hybrid algorithms, at each search step the current CSP problem is transformed into an ILP problem by following the same lines as in [14].

To test our algorithms over PLS completion problems, we have developed a generator of such problems, which is based on the following parameters: n, the size of the matrix, and k, the number of filled cells. Starting from this parameters, we generate a matrix $n \times n$ and then we choose randomly the cells to fill, and the value (over {1,. . . ,n}) to put in each of such cells. Moreover, we make sure that such partially filled matrix satisfies the definition of a PLS.

5 Experimental results

We have implemented all our algorithms using the ILOG libraries [11]. In particular, we used Solver **5.6** for algorithm `CP` and for solving the CSP formulation in algorithms `H` and `H-Gomory`, and we used CPLEX **8.1** for solving the continuous relaxation of the ILP formulation in the hybrid approaches `H` and `H-Gomory`. To test the influence of Gomory cuts in a pure OR environment, we have also used CPLEX with two different settings, one requiring the separation of Gomory cuts (in algorithm `ILP-Gomory`) and one forbidding Gomory cuts (in algorithm `ILP`).

All the experiments have been performed on an AMD Athlon 1250 MHz.

n	k	ILP	ILP-Gomory
10	2	0.69	2.00
10	5	0.63	3.50
10	7	0.52	4.60
15	3	11.41	73.71
15	7	11.32	95.33
15	11	11.00	96.60

Table 1. Time in seconds to find an optimal solution of PLS completion problems using `ILP` and `ILP-Gomory`. All PLS problems generated are solvable with objective function at the maximum value, i.e., $n^2 - k$. This means that all the cells have been filled. Times are averaged over 10 PLS completion problems.

Table 1 shows the CPU time needed to find an optimal solution for `ILP` and `ILP-Gomory`, for some values of n and k. All experimental results are obtained by running the algorithms on 10 instances of PLS completion problems, and by taking the average of the times.

It is easy to see that forcing the separation of Gomory cuts (in `ILP-Gomory`) is not convenient for PLS, at least in the standard setting of CPLEX. In fact, for any n and any k in Table 1, as well as in many other experimental results non provided here, the performance of `ILP` is always better than that of `ILP-Gomory`.

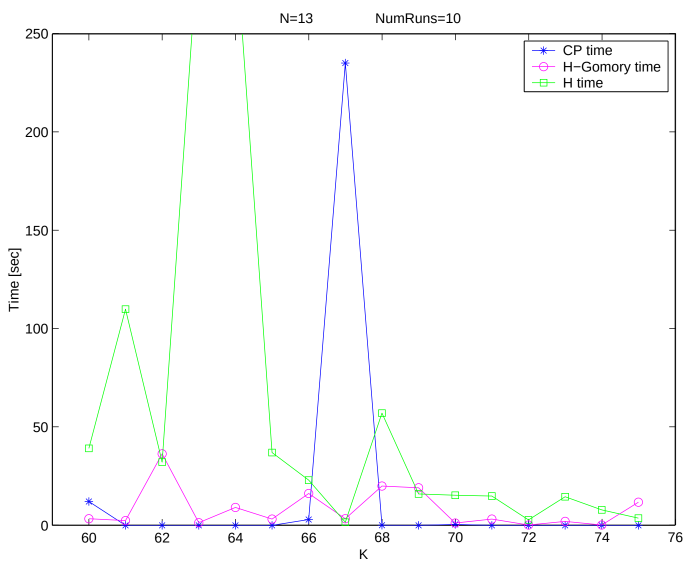


Fig. 1. Computing times (averaged over 10 PLSs) of `CP`, `H-Gomory` and `H` to find a solution for a PLS completion problem, or to discover that there is no completion.

Figure 1 compares the behavior of algorithms `CP`, `H-Gomory` and `H`, over PLS completion problems with $n = 13$ and k which varies in a way such that PLS in input has between nearly 35% and 40% of filled cells, that is the critical phase for algorithm `CP`. In fact, when we have a PLS with less than 35% or more than 45% filled cells, `CP` finds a solution very quickly, whereas in this region it sometimes takes a very long time. Notice that all the PLS problems represented in Figure 1 are completable, i.e., they always produce a success in the search tree.

Algorithm `CP` is usually much more convenient than any of the two hybrid algorithms (`H` and `H-Gomory`) for many values of k. However, `CP` has also very high peaks, which cannot be found in `H-Gomory`.

If we compare `H` and `H-Gomory`, `H-Gomory` is always more convenient than algorithm `H` which does not use Gomory cuts, since in some cases `H` needs much more memory than the available one and so it is not able to find a solution even if the PLS is completable. Therefore `H-Gomory` is always better than `H`, which means that applying Gomory cuts in the hybrid CP-OR approach is always useful.

We can conclude that `CP` is almost always better, but that by using `CP` we may run the risk of incurring in a very high peak. Thus we may use `CP` for underconstrained PLS's (filled up to 30-35%) and overconstrained ones (filled for more than 40-45% of the cells). In the middle region, it is safer to use `H-Gomory`, which shows a better behavior offering also a higher-level modelling capabilities, and a tollerable slow-down with respect to `CP` while avoiding its high peaks.

Table 2 compares algorithms `CP`, `H` and `H-Gomory` over PLS's with various values of n, while Figure 1 considered only $n = 13$.

It is easy to note again that algorithm `H-Gomory` is always better than algorithm `H`. This allows us to claim that applying Gomory cuts in a hybrid CP-OR approach is better than not applying them. With respect to algorithm `CP`, the behavior is rather clear: either `CP` is able to solve the problem very fast or it is unable to solve it in a reasonable amount of time (20 minutes of CPU time in the table), while algorithm `H-Gomory` is a very reliable approach which consistently gives quite quick solutions, thus denoting a very robust behavior.

Similar conclusions has been drawn in [9], where a pure `CP` algorithm and a hybrid CP-OR algorithm have been compared over PLS problems. However, their hybrid algorithm did not explore the use of cuts in general nor of Gomory cuts in particular.

Thus, the use of general purpose cuts like Gomory cuts in hybrid CP-OR approaches (which was totally disregarded until now in the literature) seems promising. Indeed, such cuts can be separated rather quickly from current commercial and non-commercial ILP solvers which are also able to manage them in an efficient way avoiding numerical issues that are sometimes related to their use. Their effect in our context is twofold: first they improve each continuous relaxation by allowing a better selection of the branching variable (and value), second they act as a primal heuristic procedure to obtain feasible solutions for optimization problems.

		CP		H			H-Gomory			
n	k	fails	time	fails	choice points	time	fails	choice points	Gomory time	runs
13	69	8.4e+06	328.54	0	7	9.45	0	2	3.60	10
14	68	1.7e+06	68.78	0	42	75.15	0	1	4.79	10
14	80	$>$ 2.8e+07	>20m	0	24	33.55	0	6	15.12	10
15	46	1.4e+07	574.84	≥ 0	$>$ 488	>10m	0	11	90.04	10
15	72	>3.0e+07	>20m	0	$>$ 384	$>$ 10m	0	7.4	27.79	10
15	75	>3.1e+07	>20m	0	70	155.02	0	0.1	1.972	10
15	76	>3.1e+07	>20m	0	73	163.85	0	47.9	144.78	10
15	77	>3.1e+07	>20m	0	69.3	157.70	0	13	49.01	10
15	85	>3.0e+07	>20m	0	43.8	86.37	0	9.2	34.75	10
15	89	>2.9e+07	>20m	0	50	90.43	0	5	15.78	10
16	82	$>$ 2.9e+07	>20m	0	99	274.18	13	24	130.04	10
16	95	1.9e+07	863.44	≥ 0	$>$ 467	>10m	0	4.5	39.81	10
16	102	$>$ 2.9e+07	>20m	≥ 0	>516	>10m	0	35	115.79	10
16	109	>8.1e+07	$>$ 1h	0	42	71.25	0	3	12.24	10
16	110	>2.7e+07	>20m	0	4.8	78.57	0	4.8	15.87	10
16	113	>2.6e+07	>20m	$>$ 263	>323	$>$ 312(mem)	0	19	54.42	10
16	114	>2.5e+07	>20m	0	46	90.5	0	14	38.35	10
16	115	>2.5e+07	>20m	$>$ 50	>260	$>$ 264(mem)	0	0	0.5	10
16	116	>2.6e+07	>20m	>1	>50	$>$ 90.85(mem)	0		63.35	10
14	70	246.8	0.013	0	23	42.70	0	6	16.72	10
15	40	2.4	0.06	$\geq$0	$>$ 36	$>$ 185 (mem)	0	18	156.25	10
15	55	30.8	0.009	≥ 0	>35	$>$ 115.5 (mem)	0	8	48.95	10
15	70	3.4	0.03	0	72	312.34	0	20	72.93	10
16	70	106.6	0.07	$\geq$0	>27	>135.5 (mem)	0	15	120.05	10
16	75	86655	3.56	0	50	203.35	0	21	132.34	10
16	85	65	0.07	>0	$>$ 25	$>$ 81 (mem)	0	2	15.57	10

Table 2. Comparison among algorithms `CP`, `H` and `H-Gomory`. "Mem" means memory limit exceeded.

6 Related Work

The work which is closest to ours is the one presented in [9], where different methods to find a solution of PLS completion problems are proposed and experimentally tested. However, their hybrid approach does not focus on forcing the use of cutting planes to integrate the CP and the OR appraoches. Moreover, the heuristics to choose the variables and their values are different from ours, and are probabilistically based on an LP rounding. As in our paper, also in [9] the conclusion is that none of the methods uniformly dominates the others.

In [6] they tackle the same class of problems but they just use a pure CP approach to solve them. Experimental results show that their CP approach can solve some PLS completion problems of order 45 in the phase transition (with 60% of preassigned cells), which [9], as well as our approach, can't solve with hybrid approach. However, they use specific constraints which are tailored to the class of considered problems, while we employ generic Gomory cuts.

Another related work integrating constraint programming and integer programming is [1], where two algorithms to integrate CP and IP are compared to a pure CP algorithm and a pure IP one. While one of their two hybrid algorithms is similar to ours, they use generic cuts, whereas we use Gomory cuts. Moreover, we consider Latin Square Problems whereas they consider Mutually Orthogonal Latin Squares problem, which are tuples of Latin Squares where all pairs of corresponding elements in two of the problems are different. Their experiments show that integrating CP and IP is convenient as the problem size grows.

7 Conclusions

In this paper we considered the class of Partial Latin Square completion problems and we have studied the behavior of three complete search algorithms on several instances of such problems. One algorithm follows a pure CP search strategy, another one is a hybrid CP-ILP algorithm with no Gomory cuts, and a third one is a hybrid CP-ILP algorithm with Gomory cuts.

Other hybrid CP-OR approaches have been proposed in the past, and several works have considered Partial Latin Square Problems. However, in this paper we studied in particular the influence of Gomory cuts in the OR part of the algorithm.

Gomory cuts are not useful in our pure ILP approach. However, they are convenient in our hybrid CP-ILP algorithm. In fact, by comparing the algorithm `CP`, `H`, and `H-Gomory`, our experimental results show that:

- `H-Gomory` is always better than `H`;
- `H-Gomory` is better than `CP` in several instances in the critically constrained area, that is, when the problems have from 10% to 40% of the constraints (filled cells for PLS problems);
- `H-Gomory` is comparable to `CP` in the over-constrained region.

Thus, while there is no algorithm which is uniformly better on all problem instances, the hybrid algorithm with Gomory cuts looks convenient in terms of solving time especially for critically constrained problems (where it is better than `CP`) or for problems whose constrainedness cannot be known in advance. This algorithm thus combines the modelling capabilities of CP (with compactness given by global constraints) and its constraint propagation techniques, together with the convenient branch and cut strategies, typical of OR approaches.

Acknowledgments

We would like to thank Michela Milano and Alessio Guerri for many useful suggestions and comments on this work.

References

1. G. Appa, I. Mourtos and D. Magos. Integrating constraint and integer programming for the orthogonal latin squares problem. *CP 2002*, 17–32, 2002.
2. E. Balas, S. Ceria, G. Cornuejols and N.Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1-9, 1995.
3. N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Math. Comput. Model ling*, 20(12), 1994.
4. C. J. Colbourn. The complexity of completing partial latin squares. *Dicrete Applied Mathematics*, 8:25-30, 1984.
5. R. Dechter. *Constraint processing*. Morgan Kauffmann, 2003.
6. I. Dotú, A. del Val and M. Cebrián. Redundant Modeling for the Quasigroup Completion Problem. *CP 2003*, 288–302, 2003.
7. F. Focacci, A. Lodi and M. Milano. Cost-based domain filtering. *Proc. CP 99*, 189–203. Springer Verlag, LNCS 1713, 1999.
8. F. Focacci, A. Lodi and M. Milano. Cutting Planes in Constraint Programming: an hybrid approach. *Proc. CP 2000*, 187–201. Springer Verlag, LNCS 1894, 2000.
9. C. Gomes and D. Shmoys. Completing Quasigroups or Latin Square: A structured Graph Coloring Problem. *Proc. Computational Symposium on Graph Coloring and Generalizations*, 2002.
10. J. Hooker. *Constraint Programming and Integer Programming*. Tutorial at CP 2002. URL: `http://ba.gsia.cmu.edu/jnh/cornell.ppt`.
11. ILOG Solver 5.3 User's Manual, ILOG Hybrid Cooperating Optimizers 1.3.1 User's Guide and Reference, CPLEX 8.1 User's Manual, 2002.
12. M. Milano. Integration of OR and AI constraint-based techniques for combinatorial optimization. Tutorial at IJCAI 2001, Seattle. URL: `http://www-lia.deis.unibo.it/Staff/MichelaMilano/tutorial1IJCAI2001.pdf`.
13. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.
14. P. Refalo. Linear Formulation of Constraint Programming Models and Hybrid So lver. *Proc. CP 2000*, 369–383. Springer Verlag, LNCS 1894, 2000.
15. J.C. Regin. A filtering algorithm for constraints of difference in CSPs. *Proc. AAAI-94*, 362–367, 1994.

CP and MIP in the resolution of hard combinatorial problems: A case study with nesting problems *

Maria Antónia Carravilla[1 2], Cristina Ribeiro[1 2]

[1] FEUP—Faculdade de Engenharia da Universidade do Porto
[2] INESC — Porto
Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
{mcr,mac}@fe.up.pt

Abstract. The integration of Constraint Programming (CP) and Mixed Integer Programming (MIP) has been explored to take advantage of the feasibility-oriented features of CP together with the optimization-oriented behavior of MIP. The integration strategy depends on the problem nature and typically requires some insight on the communication between the two models.

Nesting problems are NP-hard combinatorial problems where polygon-shaped pieces are placed over a plate; the goal is to minimize the length of the resulting layout. These problems have been addressed via MIP and heuristic methods in the OR community. The solution via CP models has also been studied.

As a first step in exploring hybrid CP/MIP models for nesting problems, we present a case study where MIP and CP models are compared in a nesting problem instance with convex pieces. The detailed formulation of both the MIP model and two CP models highlights the relationship between the intrinsic problem variables and the variables required in each model and opens the way for the development of combined solution strategies. We then propose a hybrid approach to nesting problems combining a modified CP approach and LP optimization.

Keywords: Packing, Nesting, Constraint Programming, Mixed Integer Programming, Linear Programming, Hybrid CP/MIP solutions.

1 Introduction

Mixed Integer Programming (MIP) is a well-established approach to the optimization of combinatorial problems. The complexity of MIP models restricts their application to small-sized instances. For larger instances heuristic methods provide satisfactory solutions in many application domains.

* Partially supported by FCT, POSI and FEDER (POSI/SRI/40908/2001 (CPackMO)) and (POSI/SRI/45379/2002 (GlobalNest))

Constraint Programming (CP) has also addressed the resolution of combinatorial optimization problems. The CP paradigm is appropriate for representing the constraints that characterize problem solutions as well as for programming resolution strategies. A CP program can be regarded as a flexible problem model that can be used both for optimization and for applying heuristic methods.

The integration of CP and MIP has been explored to take advantage of the feasibility-oriented features of CP together with the optimization-oriented behavior of MIP [1,2,3,4,5]. The integration strategy depends on the problem nature and typically requires some insight on the communication between the two models.

Nesting problems are NP-hard combinatorial problems where polygon-shaped pieces are placed over a plate; the goal is to minimize the length of the resulting layout. The problem has been addressed via MIP and heuristic methods in the OR community. The solution via CP models has also been studied. Nesting problems are particularly demanding in what concerns modelling: the natural problem variables are points in a two-dimensional space that make constraint propagation hard; it is also hard to predict for a partial solution its potential for being part of a compact layout.

As a first step in exploring hybrid CP/MIP models for nesting problems, we present a case study where MIP and CP models are compared in a nesting problem instance with convex pieces. We then propose a hybrid approach to nesting problems combining a modified CP approach and LP optimization.

The paper is organized as follows. Section 2.1 details the specification of nesting problems and briefly introduces the nofit polygon, a geometric transformation to handle polygon overlapping. In Section 3 the problem constraints are expressed as linear inequalities. Section 4 elaborates on the constraint expressions and on the nofit polygon concept to describe a MIP and two CLP models for nesting. Section 5 is the case study of the former models on a 3-piece convex set of pieces. Section 6 describes the hybrid approach proposed; the conclusions follow.

2 The Nesting Problem and the Nofit Polygon

In nesting problems a given set of small pieces must be placed over a large stretch of material (the large plate) while trying to minimise the total length used. A more detailed description of nesting problems and cutting and packing problems can be found in [6,7].

The input data in a nesting problem are the width and an upper bound on the length of the plate, together with the description of the small pieces as polygons (see Figure 1). The output data are the positioning points (X_i, Y_i) of all the pieces. The objective is to determine all the positioning points, in such a way that the pieces do not overlap and the length of the plate used to place them is minimised. In the variant of the problem that will be handled rotation of the pieces will not be considered.

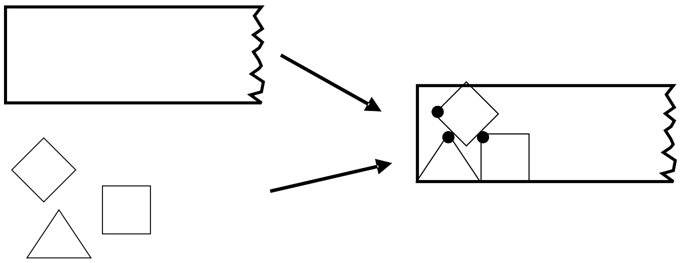


Fig. 1. Input and output data of a Nesting Problem.

Nesting problems have been tackled by several approaches, ranging from simple heuristics to local optimisation techniques and meta-heuristics [8,9,10,11,12,13]. Another traditional way of tackling combinatorial optimization problems, building mixed integer programming models and solving them with appropriate software, allows the resolution of small problem instances. Constraint Programming has also been applied to nesting [14,15,16].

2.1 Dealing with the geometry

In nesting problems the combinatorial problem coexists with a geometric problem, since solutions must be "geometrically" feasible: pieces may not overlap and must completely fit inside the plate. It is not straightforward to state that two polygons do not overlap. A geometric transformation will be used to this end. The concept of *nofit polygon* was first introduced by Art [17]; Mahadevan [18] has presented a comprehensive description of an algorithm to build it.

The nofit polygon of piece B relative to piece A (NFP_{AB}) is the locus of points traced by the reference point of B, when it slides along the contour of A. The relative orientations of A and B are maintained during this orbital movement. Piece B (the orbital piece) must never intersect piece A (the stationary piece) and they must be always in contact (see Figure 2).

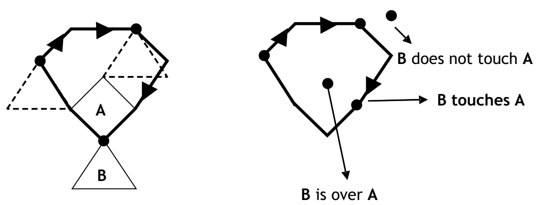


Fig. 2. Convex nofit polygon

In [19] the concept of nofit polygon for convex polygons is termed “overlapping polytope”.

Using the nofit polygon concept, the fact that polygon B does not overlap polygon A is equivalent to the positioning point of polygon B being over or outside the nofit polygon of B with respect to A. This can be turned into a relation between the positioning point of B and the edges of the nofit polygon.

To achieve a feasible and tight layout, each piece should have its reference point on the boundary of at least one nofit polygon relative to another piece and in the exterior of the nofit polygons relative to all the other pieces.

The combinatorial problem arises when choosing a placement point for each piece, among its set of feasible placement points, knowing that the choice made will restrict the feasible set of placement points of all the other pieces.

3 Linear constraints for the Nesting Problem

The goal The decision variables for a Nesting Problem are the x and y coordinates of the positioning points of all the pieces. The goal is to minimize the used length of the plate which is equivalent to minimizing the x-coordinate of the positioning point of the rightmost piece, i.e. the piece n whose $x_n + x_{n_{max}}$ is maximum: $\min \max_{n\in[1\ldots N]} \ (x_n + x_{n_{max}})$.

The constraints There are two kinds of constraints to be satisfied in a nesting problem. On one hand the pieces must fit inside the plate and on the other they must not overlap.

The constraints relating the positioning point of each piece to the boundaries of the plate are built using the rectangular envelope of the pieces.

Considering piece A and piece B, the corresponding NFP_{AB} and that the contour of the polygon is travelled clockwise, then piece B does not intersect piece A if the positioning point of B is over or on the left-hand side of at least one of the edges of NFP_{AB} (see Figure 3).

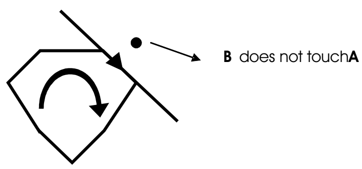


Fig. 3. Convex nofit polygon

In the following we will consider that in the coordinate system the xx values increase from left to right, and the yy values increase from top to bottom (see Figure 4).

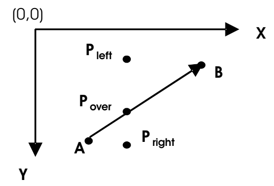


Fig. 4. Point P and a directed line segment

Considering a directed line segment from A to B ($\overrightarrow{AB}$), where (x_a, y_a) are the coordinates of point A and (x_b, y_b) are the coordinates of point B, and considering point P with coordinates (x_p, y_p), then:

◇ P is on the left-hand-side of the line generated by $\overrightarrow{AB}$ if

$$(x_a - x_b)(y_a - y_p) - (y_a - y_b)(x_a - x_p) < 0$$

◇ P is on the right-hand-side of the line generated by $\overrightarrow{AB}$ if

$$(x_a - x_b)(y_a - y_p) - (y_a - y_b)(x_a - x_p) > 0$$

◇ P is over the line generated by $\overrightarrow{AB}$ if

$$(x_a - x_b)(y_a - y_p) - (y_a - y_b)(x_a - x_p) = 0$$

The constraint "point P is over or on the left-hand side of NFP_{AB}" can therefore be expressed by OR*ing* a set of constraints of the form

$$(x_a - x_b)(y_a - y_p) - (y_a - y_b)(x_a - x_p) \leq 0,$$

one for each edge of the NFP_{AB}.

4 Models for Nesting

MIP model Considering only **convex** nofit polygons, to be able to build a conventional linear model, two components presented in the previous section must be linearised, the max operator in the objective function and the "OR" in the constraints "over or on the left-hand side".

To linearise the objective function, we introduce a new variable z, and N new constraints relating z and the values of $x_i + x_{i_{max}}$. On the one hand, z will be greater or equal to all the $x_i + x_{i_{max}}$ and on the other z is to be minimized.

$$\begin{aligned} min \quad & z \\ & \forall_n \; x_n + x_{n_{max}} \leq z \end{aligned}$$

The constraints under the "OR" are intended to guarantee that at least one of the inequations is satisfied. To linearise it, we must first introduce a term

$-M \times \lambda_{mnv}$ with a new binary variable λ_{mnv} for each equation and a new equation relating λ_{mnv} and V_{mn}, where v is the index for the vertice and V_{mn} stands for the total number of vertices of NFP_{mn}.

$$\sum_{v=1}^{V_{mn}} \lambda_{mnv} \leq V_{mn} - 1$$

It is interesting to estimate the number of equations and number of variables in the MIP model after the linearisation. The **number of continuous variables** is the number of (x, y) coordinates of all the positioning points plus z, i.e. $2 \times N + 1$. The **number of binary variables** (λ) is equal to the total number of vertices of all NFP_{mn}: $\sum_{m=1}^{N-1} \sum_{n=m+1}^{N} V_{mn}$. The **number of equations** is $4\times$ the number of pieces N, (due to the constraints relating the positioning points and the boundaries of the plate) plus the number of pieces N, (due to the linearisation of the objective function) plus the total number of vertices of all NFP_{nm}, plus the number of NFP_{mn}, (due to the linearisation of the "OR"): $5 \times N + \sum_{m=1}^{N-1} \sum_{n=m+1}^{N} V_{mn} + \sum_{m=1}^{N-1} \sum_{n=m+1}^{N} 1$.

The number of binary variables involved in the MIP model, and the fact that it only models nesting problems where all the nofit polygons are **convex** are a major drawback on the use of MIP models in solving nesting problems.

Heuristics The work on algorithms for nesting problems [7], includes techniques using integer linear programming, local search algorithms (simulated annealing, tabu search, GRASP and genetic algorithms), constructive algorithms and some simple but fairly effective heuristics. These algorithms may be analyzed under two main aspects: the resolution/search technique (combinatorial problem) and the geometric manipulation of the irregular pieces (geometric problem). In what concerns the combinatorial component, there are three groups of approaches. The first is based in the resolution of mixed integer linear programming models [20,21,22], which either locally improve solutions of the nesting problem generated by other algorithms, as the layout compaction [20], or are quite limited in the instances dimension which they are able to solve (10 to 15 pieces [22]) until optimality. The second one groups approaches based in the iterative improvement of the solutions, especially meta-heuristics such as simulated annealing, tabu search, GRASP and genetic algorithms [23,24,9,13]. They are not able to generate optimal solutions. The third includes approaches based in constructive algorithms, i.e., algorithms that build solutions by successively adding pieces to the layout [25,26]. In what concerns the geometric manipulation, the polygon manipulation technique most widely used is the nofit polygon [26,18].

CP model The efficiency of a CP program depends on a choice of variables, constraints and a search strategy such that non-feasible solutions are abandoned early.

It is possible to make a CP model for nesting using the linear constraints "over or on the left-hand side". This approach has been tested but results in poor performance, due to very limited constraint propagation [14].

CP model with a global constraint Constraint programming systems allow users to explore the features of their problems and design dedicated propagation procedures. So, instead of expressing the nesting constraints via their very fragmented linear expression, it is possible to state them at a higher level. This has also been explored [16] and results in an improved solution.

In several combinatorial problems hybrid CP/MIP approaches have been shown to produce good results [1,2,3,4,5]. Developing hybrid models for a problem requires a clear view of the variables and search strategy used in both approaches. The next section contributes to this exploration for nesting problems.

5 A Case Study with MIP and CP

The comparison of MIP and CP models will use as case study a very small problem instance: 3 convex pieces and a rectangular plate, as depicted in Figure 5.

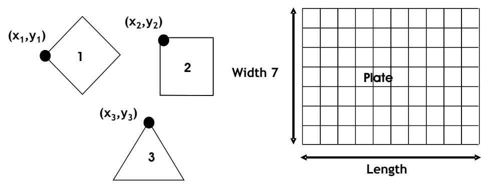


Fig. 5. An instance of the nesting problem

5.1 The MIP Model

Figure 6 is a graphical representation of the model. The first horizontal section represents the objective function, the second has the constraints concerning the location on the plate. Each of the following sections has the set of constraints that has to do with the non overlapping of two specified pieces.

Figure 7 illustrates the intermediate and final results of the "branch and bound" search strategy, where the values of λ_{mnv} are fixed along the search tree. Using a "branch and bound" strategy the number of explored nodes adds up to nearly 5000 even in a small problem instance such as this one.

The actual execution trace of CPLEX on the given example can be seen in Figure 8, where the number of search nodes is considerably reduced.

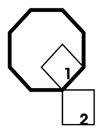
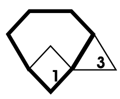
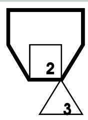
Minimize z -1z+ 1x1 <= -4 -1z+ 1x2 <= -3 -1z+ 1x3 <= -2	minimize plate lenght	
1x1 >= 0 1x2 >= 0 1x3 >= 2 1x1 <= 6 1x2 <= 7 1x3 <= 8 1y1 >= 2 1y2 >= 0 1y3 >= 0 1y1 <= 5 1y2 <= 4 1y3 <= 4	piece 1 inside plate piece 2 inside plate piece 3 inside plate	
-2x1 +2y1 +2x2 -2y2 -M*λ121 <= -6 -3x1 +0y1 +3x2 +0y2 -M*λ122 <= -9 -2x1 -2y1 +2x2 +2y2 -M*λ123 <= -12 0x1 -3y1 +0x2 +3y2 -M*λ124 <= -15 2x1 -2y1 -2x2 +2y2 -M*λ125 <= -14 3x1 +0y1 -3x2 +0y2 -M*λ126 <= -12 2x1 +2y1 -2x2 -2y2 -M*λ127 <= -8 0x1 +3y1 +0x2 -3y2 -M*λ128 <= -6 λ121+λ122+λ123+λ124+λ125+λ126+λ127+λ128 <= 7 λ121,λ122,λ123,λ124,λ125,λ126,λ127,λ128 binary	or { over or lhs edge 1 over or lhs edge 2 .... over or lhs edge 8	
-3x1 +2y1 +3x3 -2y3 -M*λ131 <= 0 -2x1 -2y1 +2x3 +2y3 -M*λ132 <= -10 0x1 -4y1 +0x3 +4y3 -M*λ133 <= -20 2x1 -2y1 -2x3 +2y3 -M*λ134 <= -18 3x1 +2y1 -3x3 -2y3 -M*λ135 <= -12 2x1 +2y1 -2x3 -2y3 -M*λ136 <= -8 -2x1 +2y1+ 2x3 -2y3 -M*λ137 <= 0 λ131+λ132+λ133+λ134+λ135+λ136+λ137 <= 6 λ131,λ132,λ133,λ134,λ135,λ136,λ137 binary	or { over or lhs edge 1 over or lhs edge 2 .... over or lhs edge 7	
-3x2 +2y2 +3x3 -2y3 -M*λ231 <= -6 -3x2 +0y2 +3x3 +0y3 -M*λ232 <= -6 0x2 -7y2 +0x3 +7y3 -M*λ233 <= -21 3x2 +0y2 -3x3 +0y3 -M*λ234 <= -15 3x2 +2y2 -3x3 -2y3 -M*λ235 <= -15 0x2 +3y2 +0x3 -3y3 -M*λ236 <= -9 λ231+λ232+λ233+λ234+λ235+λ236 <= 5 λ231,λ232,λ233,λ234,λ235,λ236 binary	or { over or lhs edge 1 over or lhs edge 2 .... over or lhs edge 6	

Fig. 6. The MIP model for the instance

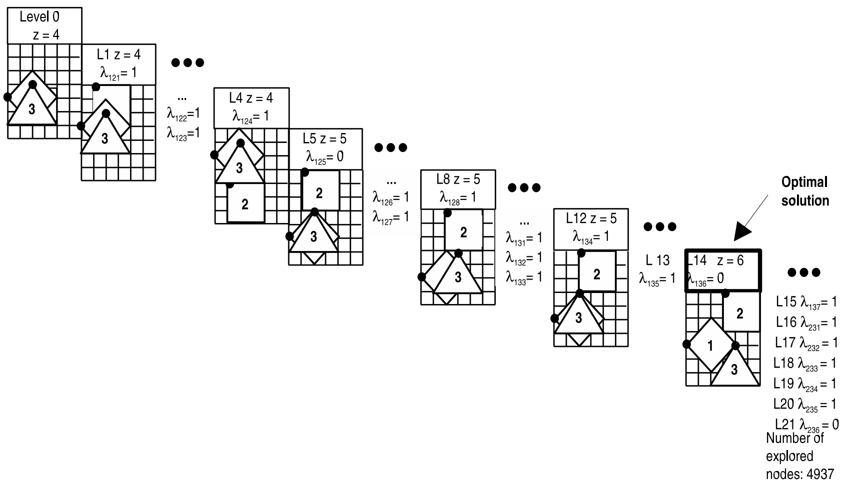


Fig. 7. The search strategy for the MIP model

```
Tried aggregator 1 time.
MIP Presolve eliminated 12 rows and 0 columns.
MIP Presolve modified 21 coefficients.
Reduced MIP has 27 rows, 28 columns, and 114 nonzeros.
Presolve time =    0.00 sec.
Clique table members: 1
MIP emphasis: balance optimality and feasibility
Root relaxation solution time =    0.00 sec.

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

      0     0        4.0000     6                      4.0000       22
*     0+    0                   0        7.0000        4.0000       22   42.86%
                     4.0000     6        7.0000     Cuts:  13       34   42.86%
*    23    10                   0        6.6667        4.0000      108   40.00%
*    34     7                   0        6.6000        4.0000      158   39.39%
*    40    10                   0        6.1667        4.0000      181   35.14%
*    60     4                   0        6.0000        5.0000      229   16.67%

Implied bound cuts applied:  11
Flow cuts applied:  2
Gomory fractional cuts applied: 4

Integer optimal solution:  Objective =    6.0000000000e+00
Solution time =    0.02 sec.  Iterations = 243  Nodes = 66
```

Fig. 8. Trace of CPLEX

5.2 A CLP Model with Linear Constraints

A CP model can be built on the basis of the "over or on the left-hand side" constraints for ensuring that each positioning point is outside the NFP's relating the piece with each of the other ones. Figure 9 represents the model graphically. The constraints of being inside the plate are simply captured in the initial domains for the variables. The constraints of "non-overlappedness" are expressed on the basis of the linear "over or on the left-hand side" inequalities.

The code extracts in Figure 9 are from a Sictsus Prolog [27] version of the model. The representation of the problem is straightforward, there is no need either to transform the disjunctions or to process the objective function. The search tree for the 3-piece instance is represented in Figure 10.

5.3 A CLP Model with a Global Constraint

The global constraint for nesting is intended to improve the propagation in this problem by imposing "non-overlappedness" on the basis of a whole NFP instead of breaking it down into edge constraints. It is therefore a more compact constraint, stating that some (positioning) point must be outside some NFP.

The propagation algorithm for the so-called "outside" constraint is easy to present visually, see Figure 11. The constraint is activated in the situation where some piece (piece 1 in the example) is fixed and some other piece (piece 3 in the example) has just been assigned its x coordinate. Propagation results in removing from the domain of the y coordinate those points that lie inside de NFP (NFP_{13} in this case).

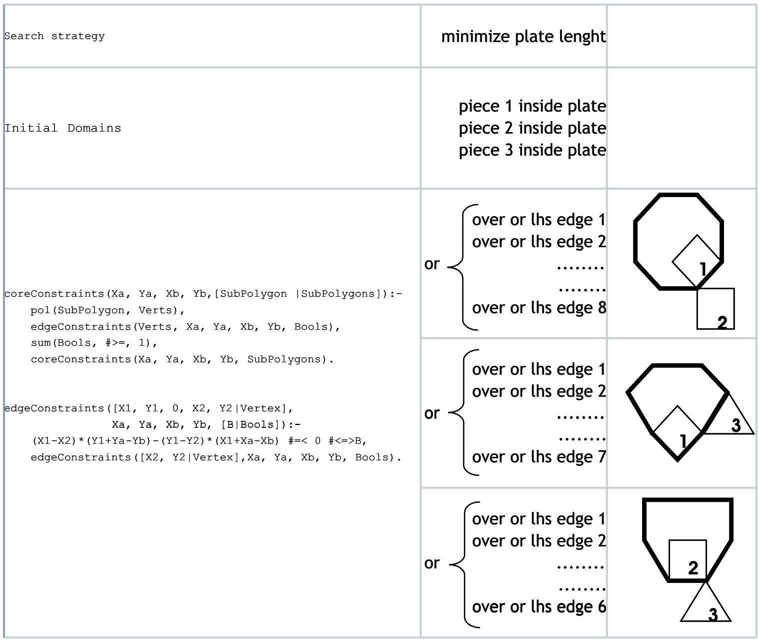


Fig. 9. A CP Model for Nesting

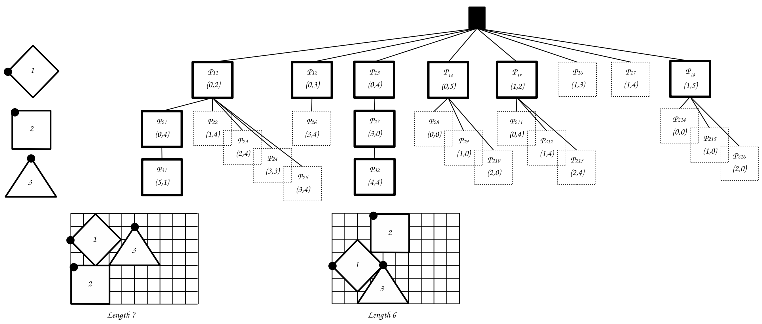


Fig. 10. The CP Search Strategy

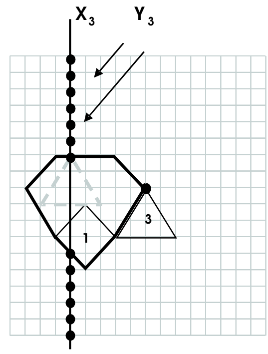


Fig. 11. The “outside” Global Constraint

Figure 12 represents the model based on the global constraints. The constraint statement is obviously succinct; the NFP are as in the two previous cases.

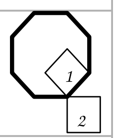
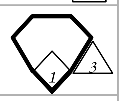
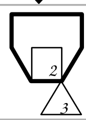
Search strategy	minimize plate lenght	
Initial Domains	piece 1 inside plate piece 2 inside plate piece 3 inside plate	
Global constraint “outside”	outside (P_1, NFP_{12})	
	outside (P_2, NFP_{12})	
	outside (P_2, NFP_{23})	

Fig. 12. A Global Constraint-based Model for Nesting

6 Hybrid Models for Nesting

The models presented in detail for the simple instance of the nesting problem have been used for observing the CP and MIP models and choose paths to explore in building hybrid approaches.

Nesting does not facilitate the evaluation of partial solutions (those where only part of the pieces have been positioned). An apparently good solution for $n-1$ pieces may appear as unacceptable as the last one is positioned, and it is hard to find lower bounds on the optimal solution.

The very global and very rigid nature of nesting makes it hard to try techniques that work well for other problems. This is a kind of problem for which it is easy for people to analyze a solution, qualitatively evaluate it and even devise strategies for improving it. It is however hard to define systematic approaches for generating optimal solutions, either automatically or manually, without exhaustive search.

The study of the MIP model for nesting shows that its combination with a CP approach would need to consider the problem constraints at the low level of the "over or on the left-hand side" constraints. As it is also not promising to try to optimize nesting by setting requirements on partial solutions, what we propose is to work incrementally based on a complete solution, with the expected result of improving it. Each "improvement" will make use of a hybrid CP/LP method.

Consider then a complete layout such as the one on the first part of Figure 13. Besides the actual pieces, the NFP's for a selected piece (the bold S piece) are also represented. Suppose that we try to improve the layout and focus attention on this piece. Due to the non-overlapping constraint, we know that the positioning point for piece S must be "over or on the left-hand side" of an edge for each of the represented NFP's. This will be the basis for a refinement step.

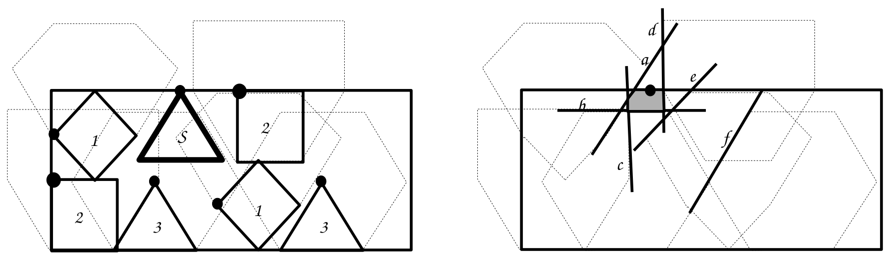


Fig. 13. The Hybrid Approach

On the second part of Figure 13 the positioning point for S is represented and, for each NFP, an edge for which "over or on the left-hand side" holds is highlighted. It is possible now to perform an optimization step whereby the position of piece S is adjusted, subject to the tractable set of constraints imposed by the selected edges. Note that this does not allow full freedom for moving S with respect to the remaining pieces. If this strategy is extended to all the pieces, the resulting set of constraints can be used in a LP optimization. This approach has already been used on so-called "compaction" problems [28].

To complete the hybrid scheme we need a strategy for driving the optimization step. Starting from a compact layout, we must change the location of some pieces so that the compaction algorithm becomes effective. To this end, our proposal is to use CP and a global constraint, and to focus on relatively small pieces. Even in a tight layout, it is quite possible that a small piece fits in several places without altering it substantially, namely without increasing the length. If we try alternative positions for a selected small piece, the resulting layout is a good candidate for the LP compaction step.

In summary, the proposed hybrid approach is articulated as follows:

Input Data: An initial layout for a set of pieces, the associated plate and the NFP's for each pair of pieces.

CP step: Identify the piece that will trigger the refinement step. Candidates are pieces that have alternative locations in the current layout without length increase. A global constraint for "point outside set of NFP's" is used to find alternative positions for the selected piece.

LP step: For each piece in the layout, select one edge (e.g. a, b, c, d, e, f) from each of its NFP's such that the "over or on the left-hand side" constraint holds. Conjoin the sets of constraints for all the pieces, and use them on an LP optimization of the problem goal.

Iterate: Repeat the refinement step with a different position or a different piece.

It is possible to use different criteria for selecting the piece to relocate. A less strict rule than the one presented is to allow a piece to be moved to the tail of the layout with some increase on the length. In this case the result of the optimization step is kept if the resulting length is smaller than the initial one.

7 Conclusions and Further Work

In nesting the geometrical bidimensional constraints originate models that are hard to deal with in MIP, for which problems of reasonable size are out of question, and also in CP, where neither the expression of constraints in a natural form nor an optimization-oriented behavior are easy to achieve.

This work proposes a new strategy to address the same problem, starting with previously built CP tools and using techniques that come from OR approaches to the same problem. The analysis of previous CP and MIP solutions, instantiated in the case study, led to the conclusion that the structure of the nesting problem does not make it a candidate for the hybrid CP/MIP models that proved successful in other problems.

The hybrid solution we propose requires further work in the global constraint "outside" proposed for nesting in [16], in the sense of making it more effective when dealing with complete layouts where some pieces may have alternative positions. Some experimentation is needed in what concerns the strategies for choosing the pieces to move. The LP compaction algorithm can be adapted from the one reported in [28].

This work has been developed within a group where approaches to the same problem using heuristics and local optimisation techniques are producing relevant results. Hybrid approaches based on constraint programming will be subject to the benchmarks of the state-of-the-art OR approaches. Any promising results yielded by the CP-based hybrids will motivate new insights and hopefully better combinations of CP and OR models and algorithms.

8 Acknowledgements

We thank the referees from CPAIOR'04 for many insightful comments on the CP approaches to nesting problems. We thank our colleagues José Fernando Oliveira and António Miguel Gomes for many stimulating discussions and for some of the ideas on hybrid models for nesting.

References

1. Milano, M., Ottosson, G., Refalo, P., Thorsteinsson, E.S.: The Role of Integer Programming Techniques in Constraint-Programming's Global Constraints. INFORMS Journal on Computing **14** (2002) 387–402
2. Ottosson, G., Thorsteinsson, E.S., Hooker, J.N.: Mixed Global Constraints and Inference in Hybrid CLP-IP Solvers. Annals of Mathematics and Artificial Intelligence **34** (2002) 271–290
3. Rodosek, R., Wallace, M.G., Hajian, M.T.: A new approach to integrate mixed integer programming with CLP. Annals of Operational Research, Recent Advances in Combinatorial Optimization **86** (1999) 63–87
4. Yunes, T.H., Moura, A.V., de Souza, C.C.: A hybrid approach for solving large scale crew scheduling problems. In: PADL. (2000) 293–307
5. Bockmayr, A., Kasper, T.: Branch-and-Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming. INFORMS Journal on Computing **10** (1998) 287–300
6. Dowsland, K., Dowsland, W.: Packing problems. European Journal of Operational Research **56** (1992) 2–14
7. Dowsland, K., Dowsland, W.: Solution approaches to irregular nesting problems. European Journal of Operational Research **84** (1995) 506–521
8. Dowsland, K., Dowsland, W., Bennell, J.: Jostling for position: Local improvement for irregular cutting patterns. Journal of the Operational Research Society **49** (1998) 647–658
9. Błażewicz, J., Hawryluk, P., Walkowiak, R.: Using tabu search approach for solving the two-dimensional irregular cutting problem in tabu search. In Glover, F., Laguna, M., Taillard, E., de Werra, D., eds.: Tabu Search. Volume 41 of Annals of Operations Research. J. C. Baltzer AG (1993)

10. Stoyan, Y., Yaskov, G.: Mathematical model and solution method of optimization problem of placement of rectangles and circles taking into account special constraints. International Transactions on Operational Research **5** (1998) 45–57
11. Milenkovic, V., Daniels, K.: Translational polygon containment and minimal enclosure using mathematical programming. International Transactions in Operational Research **6** (1999) 525–554
12. Bennell, J.A., Dowsland, K.A.: Hybridising tabu search with optimization techniques for irregular stock cutting. Management Science **47** (2001) 1160–1172
13. Gomes, A.M., Oliveira, J.F.: A 2-exchange heuristic for nesting problems. European Jornal of Operational Research **141** (2002) 359–370
14. Carravilla, M.A., Ribeiro, C., Oliveira, J.F.: Solving nesting problems with non-convex polygons by constraint logic programming. International Transactions in Operational Research **10** (2003) 651–663
15. Ribeiro, C., Carravilla, M.A., Oliveira, J.F.: Applying constraint logic programming to the resolution of nesting problems. Pesquisa Operacional **19** (1999) 239–247
16. Ribeiro, C., Carravilla, M.A.: A global constraint for nesting problems. In: CPAIOR. (2004) 256–270
17. Art, R.: An Approach to the Two-Dimensional, Irregular Cutting Stock Problem. Technical Report 36.008, IBM Cambridge Centre (1966)
18. Mahadevan, A.: Optimization in Computer-Aided Pattern Packing. PhD thesis, North Carolina State University (1984)
19. Beldiceanu, N., Guo, Q., Thiel, S.: Non-overlapping constraints between convex polytopes. Lecture Notes in Computer Science **2239** (2001)
20. Li, Z., Milenkovic, V.: Compaction and separation algorithms for non-convex polygons and their applications. European Journal of Operational Research **84** (1995) 539–561
21. Scheithauer, G., Terno, J.: Modellierung von anordnungsproblemen. Technical Report MATH-NM-02-1992, Technische Universität Dresden (1992)
22. Stoyan, Y.: Precise and approximate methods of problems in irregular allocation of non-convex polygons in a strip of minimal length. In: IFORS'93 Conference, Lisboa. (1993)
23. Gomes, A.M., Oliveira, J.: A grasp approach to the nesting problem. In: Extended Abstracts of MIC2001 – 4th Metaheuristics International Conference, Porto, Portugal. (2001)
24. Oliveira, J., Ferreira, J.: Algorithms for nesting problems. In Vidal, R.V.V., ed.: Applied Simulated Annealing. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag (1993) 255–273
25. Albano, A., Sapuppo, G.: Optimal allocation of two-dimensional irregular shapes using heuristic search methods. IEEE Transactions on Systems, Man and Cybernetics **SMC-10** (1980) 242–248
26. Oliveira, J.: Problemas de Posicionamento de Figuras Irregulares : Uma Perspectiva de Optimização. PhD thesis, Faculdade de Engenharia da Universidade do Porto (1995)
27. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. In Glaser, H., Hartel, P., Kucken, H., eds.: Programming Languages: Implementations, Logics, and Programming. Volume 1292 of Lecture Notes in Computer Science., Southampton, Springer-Verlag (1997) 191–206
28. Gomes, A.M., Oliveira, J.: Solving Irregular Strip Packing problems by hybridising simulated anealing and linear programming. European Journal of Operational Research (2005) Article in press.

Hybrid Search Algorithm for Solving Constraint Satisfaction Problems

A. Reza Hatamloo[1] and M. Reza Meybodi[2]

[1] Department of Computer Science, Khoy University
Khoy, Iran
reza_hatamloo@yahoo.com
[2] Department of Computer Science, Amir Kabir University
Tehran, Iran
meybodi@akt.ac.ir

Abstract. In this paper we present a hybrid search algorithm for solving constraint satisfaction and optimization problems. This algorithm combines ideas of two basic approaches: complete and incomplete algorithms which also known as systematic search and local search algorithms. Different characteristics of systematic search and local search methods are complementary. Therefore we have tried to get the advantages of both approaches in the presented algorithm. The major advantage of presented algorithm is finding partial sound solution for complicated problems which their complete solution could not be found in a reasonable time. This algorithm results are compared with other algorithms using the well known n-queens problem.

1 Introduction

Many problems in the fields of artificial intelligence, network, database, engineering and other areas of computer science can be viewed as special cases of constraint satisfaction problems, including image processing, natural language parsing, routing, circuit design, scheduling and more. A CSP is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take [7]. The search space of CSPs is often exponential because the problem is NP-complete. Therefore a number of different approaches to the problem have been proposed to reduce the search space and find a feasible solution in a reasonable time. Based on the search space exploring and variable selection heuristics, those algorithms can be divided into two major groups: (1) complete; (2) incomplete.

Complete algorithms [2, 7] seek any solution or all solutions of a CSP. Or they try to prove that no solution exists. These algorithms also divided into subgroups. The consistency (or constraint propagation) techniques try to eliminate values that are inconsistent with some constraints. There are many consistency techniques ensuring different levels of consistency. The systematic search methods explore systematically (and exhaustively) the whole search space. Most common are so called tree search methods. They view the search space as a search tree. Each node represents mutually exclusive choices which partition the remaining search space into disjoint sub-spaces. This structure enables to remember with acceptable memory requirements, which parts of the search space have already been visited. Usually, a node corresponds to an assignment of a particular value to a particular variable. The efficiency of tree search methods may be significantly improved with the help of consistency techniques. There are two basic ways of employing consistency techniques together with tree search methods. Look back schemes are invoked when the algorithm is preparing to backtrack after encountering a dead-end. Look ahead schemes can be employed whenever the algorithm is preparing to assign a value to the next variable.

Incomplete search methods [4, 8] do not explore the whole search space. They search the space either non-systematically or in a systematic manner, but with a limit on some resource. They may not provide a solution but their computational time is reasonably reduced. They can not be applied to find all solutions or to prove that no solution exits. However, they may be sufficient when just some solution is needed. Another application is to seek a feasible solution of an optimization problem. There are two basic approaches for incomplete search. The constructive algorithms gradually extend a partial solution to a complete one. They are based on tree search algorithms for satisfaction problems and may benefit from constraint propagation. Iterative repair methods start with an initial solution, found by some other approach. Or it may be found randomly. They incrementally alter the values to get a "better" one, and eventually an optimal or at least a good enough solution. Their non-systematic nature generally voids the guarantee of "completeness", but they are often able to get quickly close to the optimal solution and overcome the algorithms based on tree search. These algorithms may be applied to satisfaction problems, as well. The search will move throughout all assignments and the quality of the assignment will be determined by the number of violated constraints. Important iterative repair methods are the so called local search methods [4, 8]. Local search is based on making small (local) changes in assignments to variables. A way of moving from one solution to another is called a move. A move is problem-specific. A set of all solutions that differ from the current one in only one move is called a neighborhood. At each iteration step, a solution is selected from the current neighborhood (typically a better one). And it becomes the current solution. If the algorithm selected only better solutions all the time, it could get entrapped in so called local optimum. It is a solution whose neighborhood contains only solutions with equal or worse cost, and it is not the optimal solution (global optimum). To avoid this state, local search is often equipped with various metaheuristics randomizing the search, i.e., allowing also worse neighbors to be selected under certain conditions the algorithm is usually bounded by some stopping criterion. It may be based on computational time, number of moves or some other indicator.

Systematic search algorithms suffer from the disadvantage of tree search. If a wrong decision is made early in the search tree, it is necessary to undone it by backtracking. Many so far assigned values are thrown away. Look-back and look-ahead schemes try to reduce the importance of this problem, but they can not solve it completely.

The local search algorithms decide upon complete solutions and the absence of systematicity allows them to modify assignments in any order. However, when a problem is tightly constrained, a local search algorithm may not get to a solution. Either the initial solution may be too far from it, or it may get entrapped in a local optimum.

While algorithms based on tree search are due to constraint propagation effective at finding solutions for tightly constrained problems with complex and overlapping constraints, the local search methods can be superior at optimization problems that are loosely constrained.

2 Hybrid Search Algorithm

The above mentioned considerations lead to conclusion that the different characteristics of systematic search and local search methods are complementary. Therefore promising hybrid algorithms trying to get the advantages of both approaches have been proposed recently [1, 5, 6].

The hybrid search algorithm, that we propose here, is based on ideas of local search methods [4, 8]. However, in contrast to classical local search techniques, it operates over feasible, though not necessarily complete solutions. In such a solution, some variables can be left unassigned. Still all hard constraints on assigned variables must be satisfied. Similarly to systematic search algorithms, this means that there are no violations of constraints. Working with feasible incomplete solutions has several advantages compared to the complete infeasible assignments that usually occur in local search techniques. For example, when the algorithm is not able to find a complete solution, an incomplete (but feasible) one can be returned, e.g., a solution with the least number of unassigned variables found. Moreover, because of the iterative character of the search, the algorithm can easily start, stop, or continue from any feasible solution, either complete or incomplete.

The search proceeds iteratively (see Fig. 1 for algorithm). During each step, an unassigned or assigned variable is initially selected. Typically an unassigned variable is chosen like in systematic search. An assigned variable may be selected when all variables are assigned but the solution is not good enough. Once a variable is selected, a value from its domain is chosen for assignment. Even if the best value is selected, its assignment to the selected variable may cause some conflicts with already assigned variables. At this point this algorithm decides to accept or reject the current variable and its value. If the number of conflicts was reasonable, such conflicting variables are removed from the solution and become unassigned and the selected value is assigned to the selected variable. But, if the number of these conflicting variables was great, current variable shall be rejected and a new variable shall be selected. The reasonability for number of conflicts is a heuristic problem. For presented algorithm we have defined 20 percentages for reasonability. This means that if the number of conflicts between current variable and assigned variables was great than 20% of all assigned variables, the current variable must be rejected. The algorithm attempts to move from one (partial) feasible solution to another via repetitive assignment of a selected value to a selected variable. During this search, the feasibility of all constraints in each iteration step is enforced by unassigning the conflicting variables. The search is terminated when the requested solution is found or when there is a timeout, expressed e.g., as a maximal number of iterations or available time being reached. The best solution found is then returned.

```
procedure hybrid_search(unlabeled, answer, max_repeat, max_conflict)
   /* unlabeled is a list of un-labeled variables and answer
      is a incomplete answer (empty at the start) */
    repeats=0;
    while unlabeled not empty & repeats<max_repeat
        repeats ++;
        variable = Variable_Chooser(unlabeled, answer);
        unlabeled -= variable;
        value = Value_Chooser(variable, answer);
        conflicts = Conflict_counter(variable, value,answer);
        if conflicts < max_conflict then
           unlabeled += label(answer, variable, value);
              /* label the variable and return conflict variables */
        else
           unlabeled += variable;
    end while;
    return answer;
end hybrid_search
```

Fig. 1. A kernel of the presented hybrid search algorithm

The above algorithm schema is parameterised by three functions: the variable chooser, the value chooser and the conflict counter.

There are several guidelines how to select a variable. In local search the variable participating in the largest number of violations is usually selected first. In systematic search algorithms, the first-fail principle is often used, i.e., a variable whose instantiation is most complicated is selected first. This could be the variable involved in the largest set of constraints or the variable with the smallest domain etc [3, 4, 7]. It is possible to select the worst variable among all unassigned variables but due to complexity of computing the heuristic value, this could be rather expensive in some cases. Therefore in the presented algorithm we select a subset of unassigned variables randomly and then select the worst variable from this subset. The results will not be much worse and we can select the variable much faster.

After selecting the variable we need to find a value to be assigned. Typically, the most useful advice is to select the best-fit value [3, 4, 7]. So, we are looking for a value, which is most preferred for the variable and also which causes the least trouble. It means that we need to find a value with minimal potential future conflicts with other variables. To remove cycling, it is possible to randomise the value selection procedure. For example, it is possible to select five best values for the variable and then choose one of them randomly.

3 Results

In this section we will present the efficiency of the described hybrid search algorithm on the N-queens problem. We will compare the achieved results with another local search and complete search algorithm. All these algorithms were implemented in Java and all presented results were measured on Intel Pentium IV 2.4GHz, 256MB RAM, Windows XP.

The N-queens problem is to place n queens on a n×n chessboard so that no queen is under a direct attack from any other one.

The algorithms which we used in this comparison are min-conflicts [4, 7, 8] and full look ahead [7]. Min-Conflicts algorithm is one of the typical local search algorithms and it is highly efficient on the N-queens problem and full look ahead algorithm is one of the complete search algorithms.

The time for solving the N-queens problem is given in the table 1. It is not surprising that the local search algorithms are much more efficient than complete search on this problem. So, let's focus only on the differences between the presented hybrid search algorithm and the minimal conflicts algorithm. As we can see on figure 1, the presented algorithm is nearly as fast as the min-conflicts algorithm.

number of queens	hybrid algorithm	incomplete algorithm	complete algorithm
1000	150 ms	70 ms	20372 ms
2000	623 ms	271 ms	-----
3000	1305 ms	562 ms	-----
4000	2037 ms	1013 ms	-----
5000	3575 ms	1696 ms	-----
6000	5115 ms	2551 ms	-----
7000	7072 ms	3552 ms	-----
8000	9149 ms	4982 ms	-----
9000	11516 ms	7220 ms	-----
10000	14543 ms	9952 ms	-----

Table 1. Time to solve the N-queens problem; mean value from 10 measurments

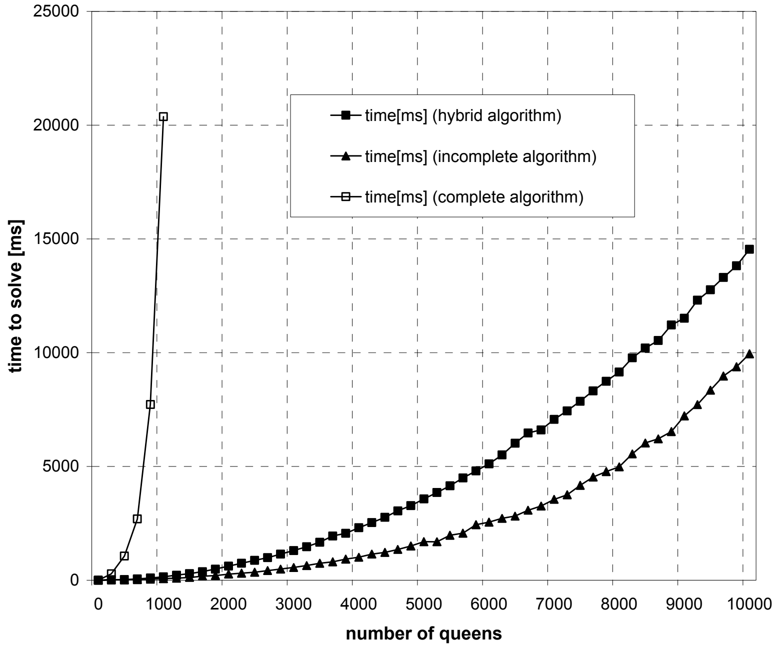


Fig. 2. Comparison of the time for the described algorithms; mean value from 10 measurements

4 Conclusion

We have presented a hybrid approach for solving constraint satisfaction and optimization problems which combines ideas of local search and systematic search algorithms. The basic motivation was to design an algorithm for solving complicated problems which their complete solution could not be found in a reasonable time. We have compared the efficiency of the described hybrid search algorithm with another local search and systematic search algorithm on the N-queens problem. The achieved results show that the presented algorithm is nearly as fast as the local search algorithm. We believe that this algorithm can be successfully applied to many constraint satisfaction problems especially when working with sound incomplete solutions is needed in general.

References

[1] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence, 139(1):21–45, 2002.

[2] Vipin Kumar. Algorithms for constraint satisfaction problems: A survey. AI Magazine, 13(1):32–44, 1992.

[3] K. Marriot, P. J. Stuckey. Programming with Constraints: An Introduction. The MIT Press, 1998.
[4] Z. Michalewicz and D. B. Fogel. How to Solve It: Modern Heuristics. Springer-Verlag, 2000.
[5] W. Ruml. Incomplete tree search using adaptive probing. In Proceedings of Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01), 2001.
[6] Andrea Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97), pages 1254–1259, Nagoya, Japan, 1997.
[7] E. Tsang. Foundations of Constraint Satisfaction. Academic Press, 1993.
[8] Stefan Voß. Meta-heuristics: State of the art. In Alexander Nareyek, editor, Lcal search for planning and scheduling: revisited papers, pages 1–23. Springer-Verlag LNCS 2148, 2001.

Benders decomposition in Constraint Programming

Hadrien Cambazard, Narendra Jussien
email: {hcambaza,jussien}@emn.fr

École des Mines de Nantes, LINA CNRS FRE 2729
4 rue Alfred Kastler – BP 20722 - F-44307 Nantes Cedex 3, France

Abstract. Recent work have exhibited specific structure among combinatorial problem instances that could be used to speed up search or to help users understand the dynamic and static intimate structure of the problem being solved. Several Operations Research approaches apply decomposition or relaxation strategies upon such a structure identified within a given problem. This paper presents how Benders decomposition could be adapted to constraint programming when specific relationships between variables are exhibited. It discusses the way a decomposition framework could be embedded in constraint solvers, taking advantage of structures for a non expert user in a generic way. To achieve the interaction between structures, it explores the possibility of deriving logic Benders cuts using an explanation based framework for Constraint Programming.

1 Introduction

Benders decomposition [2] has been successfully used in numerous situations in operations research. It is often introduced as a basis for models and techniques using the complementary strengths of constraint programming and optimization techniques. Hybridation schemes have appeared recently and provided interesting computational results [13, 6, 12, 14, 3]. They have been extended [5, 15] to take into account other kinds of sub-problems and not only the classical programming linear ones. However, decomposition has never been used to our knowledge in a pure constraint programming approach.

Real industrial problems often exhibit different sets of variables playing distinct business roles: they are in fact often structured problems. This paper discusses the way a decomposition framework could be embedded in constraint solvers, taking advantage of structures for a non expert user in a generic way. We explore the possibility of deriving logic Benders cuts using an explanation based framework for Constraint Programming. We evaluate how the use of explanations could take advantage of a master-slaves relationship detected in the problem and describes Benders decomposition as a nogood recording strategy in Constraint Programming. Moreover we propose to apply this technique in the case of approximated structures for the sub-problems.

Section 2 will introduce explanation based constraint programming as well as the basis of Benders decomposition. Section 3 and 4 will then focus on the use of explanation as Benders cuts and the main algorithm implemented on the `PaLM` framework. First practical results will be presented in Section 5.

2 Context

2.1 Explanations for constraint programming

An explanation records information to justify a decision of the solver as a reduction of domain or a contradiction. It is made of a set of constraints C' (a subset of the original constraints of the problem) and a set of decisions $dc_1, dc_2, \ldots dc_n$ taken during search (assignments are typical decision constraints, but other more generic constraints can be considered). An explanation of the removal of value a from variable v, $expl(x \neq a)$ will be written as follows:

$$C' \wedge dc_1 \wedge dc_2 \wedge \cdots \wedge dc_n \Rightarrow v \neq a$$

When a domain is emptied, a contradiction is identified. An explanation for this contradiction is computed by uniting each explanation of each removal of value of the variable concerned. Intelligent backtracking algorithms that question a relevant decision appearing in the conflict are then conceivable [4]. By recording a relevant part of the explanations involved in conflicts, a learning mechanism can be implemented [9]. Notice that the set of decisions is also called a nogood. It represents a partial assignment that can not be extended to a solution. On the previous definition, $dc_1 \wedge dc_2 \wedge \cdots \wedge dc_n \wedge v = a$ is the corresponding nogood.

2.2 Principles of Benders decomposition

Benders decomposition can be seen as a form of *learning from mistakes*. It is a solving strategy that uses a partition of the variables of the problem into two sets: x, y. The strategy can be applied to a problem of this general form:

$$\begin{array}{l} \text{P : Min } f(y) + cx \\ \text{s.t : } g(y) + Ax \geq a \text{ with : } y \in D, x \geq 0 \end{array}$$

A master problem considers only a subset of variables y (often integer variables, D is a discrete domain). A sub-problem (SP) tries to complete the assignment on x. If it is possible, the problem is solved, but if not, a cut (rejecting at least the current assignment on y) is produced and added to the master problem: it is called a Benders cut.

This cut has the form $z \geq h(y)$ (z represents the objective function – $z = f(y) + cx$) and constitutes the key point of the method, it is inferred by the dual of the sub-problem. Let us consider an assignment y^* given by the master, the sub-problem (SP) and its dual (DSP) can be written as follows:

$$\begin{array}{ll} \text{SP : Min } cx & \text{DSP : Max } u(a - g(y^*)) \\ \text{s.t } Ax \geq a - g(y^*) \text{ with : } x \geq 0 & \text{s.t } uA \leq c \text{ with : } u \geq 0 \end{array}$$

Duality theory ensures that $cx \geq u(a - g(y^*))$, $u(a - g(y^*))$ is therefore a lower bound of cx. As feasibility of the dual is independent of y^*, $cx \geq u(a-g(y))$ and the inequality $f(y)+cx \geq f(y)+u(a-g(y))$ is valid, leading to the Benders cut: $z \geq f(y)+u(a-g(y))$. Moreover, according to duality, the optimal value of u^* maximizing $u(a - g(y^*))$ corresponds to the same optimal value of cx. Even if the cut is derived from a particular y^*, it is valid for all y and excludes a large class of assignments which share common characteristics. The number of solutions to explore is reduced and the master problem can be written at the k^{th} iteration:

$$\begin{aligned}&\text{MP : Min } z\\ &\text{s.t : } z \geq f(y) + u_i^*(a - g(y)) \qquad \forall i < k\end{aligned}$$

From all of this, it can be noticed that duals variables or multipliers[1] need to be defined to apply the decomposition. However, a generalized scheme has been proposed in 1972 by Goeffrion [15]. Hooker and Ottosson [5] proposed also to overcome this limit and to enlarge the classical notion of *dual* by introducing an *inference dual* available for all kinds of sub-problems. They refer to a more general scheme and suggests a different way of thinking about duality: a Benders decomposition based on *logic*. Duality means now to be able to produce a proof, the logical proof of optimality of the sub-problem and the correctness of cuts inferred. In classical Benders decomposition, this proof is established thanks to classical duality theorems. For a discrete satisfaction problem, the resolution of the dual consists in computing the infeasibility proof and determining under what conditions the proof remains valid: this is exactly what explanations are designed for.

3 A decomposition approach in CP

In this paper, we consider problems which can be represented:

$$\begin{aligned}\text{P : } &\text{Min } obj\\ &\text{s.t : } Ct(x,y)\\ &\text{with : } x \in D_x, y \in D_y\end{aligned}$$

$Ct(x, y)$ denotes a set of constraints on variables x, y and obj can be equal to $\{f(x, y), f(y), 0\}$ (respectively, an objective function on the whole set of variables, an objective function restricted to the y variables and a satisfaction problem). The problem P will be denoted $\{P_{xy}, P_y, P_0\}$ according to the corresponding objective functions. The decomposition scheme is done among x and y. We suppose that the remaining problem over x can be formulated using n sub-problems exhibiting strong intra-relationships and weak inter-relationships. Ideally, there should be as small and independant as possible to ensure the remaining sub-problem to be easy. So we make the assumption for the sub-problem to offer such an ideal or approximate structure. P is supposed to have an ideal

[1] Referring to linear programming duality.

structure whereas it is denoted $P^{'}$ in case of such an approximated structure over the x variables (so we get in the same way $\{P^{'}_{xy}, P^{'}_{y}, P^{'}_{0}\}$).
Master problem and sub-problems have then the generic form :

$$
\begin{array}{rl|rl}
\text{MP}: & \text{Min } z & SP^k: & \text{Min } sz_k \\
& \text{s.t} : Ct(x,y) & & \text{s.t} : Ct(x_k,\overline{y}) \\
& z < \overline{z} & & x_k \in D_x \\
& y \in D_y & &
\end{array}
$$

3.1 Benders cuts as explanations

The benders cut is a logic expression over the y variables, generated from the sub-problem solution, the cut must ensure that the algorithm terminates and finds the optimal solution. At iteration k of the master, the added Benders cut ensures that :

1. It is valid; it does not exclude any feasible solution over the x, y variables of the original problem (according to the current upper bound of z).
2. It must exclude at least the current instantiation $\overline{y}$ of the master that has been proved as sub-optimal or inconsistent

(2) ensures the termination of the algorithm and (1) ensures optimality as the master problem is proved to remain a valid relaxation and to provide a lower bound of P. While solving the sub-problem, classical cuts on the objective function are added $sz < sz^*$. Once the optimal solution has been found, the problem becomes inconsistent with the previous cut and the solver provides an explanation where the set of decision constraint taken at the sub-problem level is empty. This explanation give a set of constraints (original constraints $Ct(x, y)$ and decision constraints $dc(y)$) to explain why the sub-problem can not be extended to a better solution than sz^* written as : $expl(sz \geq sz^*)$.

As the explanation is a subset of the decisions taken by the master, it excludes at least the current assignment. An empty set indicates an infeasible problem P whereas the complete set excludes only the current assignment. The explanation is proved to be valid as long as each constraint computes a valid explanation in the same way that it performs a valid pruning.

Note that the structure of the dual is used through the explanation algorithms embedded within constraints. In fact, the computation of explanations is *lazy*[2], the first explanation is taken even if several explanations exist. One cannot look for the minimal explanation for evident scalability reasons. Therefore, such an inference dual provides an arbitrary[3] dual solution but not necessarily the optimal one. Obviously, the success of such an approach depends on the degree to which accurate explanations can be computed for the constraints of the sub-problem.

[2] Not all possible explanations are computed when removing a value. Only the one corresponding to the solver actual reasoning is kept.
[3] This can also be accounted for linear duality where any dual solution is a bound for the primal problem.

3.2 Decomposition scheme

One of the key point of the Benders decomposition is to be able to derive a master problem that provides a valid lower bound for the original problem P. We used the following master problems to get the decomposition for initial problems P_y, P_0 and P_{xy} :

$$\begin{array}{ll|ll|ll} MP_y : & \text{Min } f(y) & MP_0 : & \text{Min } 0 & MP_{xy} : & \text{Min } r(y) = relax(f(x,y)) \\ & \text{s.t} : Ct(x,y) & & \text{s.t} : Ct(x,y) & & \text{s.t} : Ct(x,y) \\ & y \in D_y & & y \in D_y & & y \in D_y \end{array}$$

One can notice here that MP_y provides a valid lower bound as it is a relaxation of P_y. It is also the case for P_0 as it is a satisfaction problem. However, it is not true in the general case of P_{xy} where a specific master problem must be designed. In fact, a new objective function called $r(y)$ defined on y variables and providing a lower bound has to be defined by the user.

There are some cases where the original function is itself a relaxation. Consider for example a graph coloring problem where the number of different colors must be minimized. In this case, one can directly use the original objective as a relaxation.

$$\begin{array}{ll} \text{P1} : & \text{Min } NbDifferentValues(x,y) \\ & \text{s.t} : x_i \neq x_j \forall (i,j) \in G \end{array}$$

We have obviously $NbDifferentValues(y) \leq NbDifferentValues(x,y)$ and we can use $NbDifferentValues(y)$ as a valid objective for the master problem. In a generic case, the master problem take the form of a feasibility problem where the cuts added can be seen as $expl(z \geq z^*) \Rightarrow z \geq z^*$

We also consider the following example which will be developed later with an optimisation function given as a linear combination:

$$\begin{array}{lll} \text{P2} : & \text{Min} & y_1 + y_2 + 4x_1 + 2x_2 + 5x_3 + x_4 \\ & \text{s.t} : & |y_1 - y_2| < 2 \\ & & 5x_1 + x_2 + 2y_1 + y_2 \geq 5 \\ & & 3x_3 + 2x_4 + y_2 \geq 4 \\ & & x_1 \neq y_2 \\ & & occurrence(0, \{x_3, x_4, y_2\}) = 1 \\ & & occurrence(0, \{x_3, x_4, y_1\}) = 1 \\ & & y_1, y_2, x_1, x_2, x_3, x_4 \in [0,5] \end{array}$$

In this case again, the use of Min $y_1 + y_2$ gives a correct master problem.

4 A Benders decomposition algorithm for constraint programming

Figure 1 presents our algorithm. It has been implemented as a library of the Java version of the `PaLM` [8] solver embedded within the `choco`[4] constraints solver. The

[4] See `http://choco.sf.net`.

standard *CP* model taken as input by the solver is only enriched by indicating for each variable the problem to which it belongs (the master or the index of the sub-problem). Classical branching can be specified for the variables of each sub-problem. The cuts are managed as a single global constraint that filters the last non instantiated variable of each cut. Improvements can be expected from that side by using a GAC algorithm for example. Note that line 10 of the algorithm is used in the case of approximated structures for P_0 and P_y whereas lines 6, 8, 16, and 17 concerns P_{xy}.

Benders algorithm :
(1) **input** : an initial solution to the master problem $\overline{y}$,
(2) `begin`
(3) `repeat`
(4) $Cut = \emptyset$
(5) `for each` sub-problem spb_k `do`
(6) P_{xy} : update upper bound of spb_k with computeUb(k) using $\{\overline{z},\overline{sz_i}, \forall i < k\}$)
(7) solve spb_k on $(\overline{y}, x_k)$ to optimality
(8) P_{xy} : add its optimality explanation $expl_k$ to Cut
(9) P_0, P_y : if spb_k is infeasible, add its inconsistency explanation $expl_k$ to Cut
(10) $P_0^{'}, P_y^{'}$: $spb_{k+1} = \bigcup_{i \leq k, i > k'} spb_i$, with k', the last infeasible sub-problem.
(11) `endif`
(12) `endfor`
(13) `if` $(Cut \neq \emptyset)$ `then`
(14) Cut = computeCut(Cut)
(15) add all explanations $\in Cut$ to the master problem
(16) P_{xy} : update the upper bound of z with computeUb(0) using $\{\overline{z}, \overline{sz_1}, \ldots, \overline{sz_n}\}$)
(17) P_{xy} : store $(\overline{y}, \overline{x})$ if it is an improving solution
(18) solve the master problem to optimality
(19) `endif`
(20) `until` the master problem is infeasible $\bigvee Cut = \emptyset$
(21) P_y, P_0, $P_y^{'}$, $P_0^{'}$: the solution $(\overline{y}, \overline{x})$ is optimal if $Cut = \emptyset$ otherwise, P is infeasible.
(22) P_{xy} : the solution $(\overline{y}, \overline{x})$ is the optimal solution of P otherwise P is infeasible.
(23) `end`

Fig. 1. A Generic Benders algorithm for P_0, P_y and P_{xy}

4.1 Specific handling for problems P_0 and P_y

P_0 and P_y are closely related because they both use satisfaction problems as sub-problems. Backjumping algorithms are used for the sub-problems to compute explanations (we do not use them to perform dynamic backtracking, we therefore do not maintain the explanation network among constraints[5]) to provide dual informations on the sub-problems. Moreover, the use of backjumping for the master is possible for P_0 (which is a traditionnal *CSP*) and allows the partial

[5] This is possible with the new version of `choco` which contains a `mac-cbj` mode.

avoidance of thrashing on the master problem when adding the cuts. This is a response to Thorsteinsson [13] concerns about possible significant overhead due to redundant computations. Termination of the algorithm can be seen at line 22: an infeasible master problem for P_0 and P_y means that the whole problem is infeasible, otherwise an optimal or feasible solution has been found.

One word has to be said about approximated structures: at any iteration, once the master problem and k sub-problems have been solved, the next sub-problem $k+1$ considered is done according to the following rule. Let $k' < k$ be the index of the last infeasible sub-problem $spb_{k'}$, spb_{k+1} is then defined as : $spb_{k+1} = \bigcup_{i \leq k, i > k'} spb_i$. So if one sub-problem is consistent, the next one starts from its solution and consider for branching the variables of both problems. Such a strategy does not imply any overhead compared to solving one single sub-problem (it only changes the variable ordering) but hopes to benefit from the relative independency of sub-problems to derive disjoint cuts. There is obviously a compromise between the time spent for solving sub-problems and the quality of the retrieved information. For example, as soon as one cut has been derived, it seems reasonable to limit the search for other cuts to the sub-problems themselves.

4.2 Specific handling for problem P_{xy}

As we want to keep isolated sub-problems, we do not add the objective function as a constraint which could propagate from one sub-problem to another. Instead, we provide a way to compute the bound of one problem according to other known bounds (master and slaves) with an empty explanation. So the propagation is done *at hand* to only incriminate the master problem solution for each sub-problem. The interest of this implementation is to allow to solve separatly each sub-problem in order to get at the end several valid explanations for the master. Take the example of P_1, one could stop once we get a first sub-problem with a coloring number higher than our current upperbound, or one can continue solving the next sub-problem to derive disjoint or maybe more accurate explanations. The key is to take advantage of the structures and to find from all subproblems a cut that will avoid as more thrashing as possible on the master.

- *computeCut(Explanation[] expls)* (line 14): computes the explanation(s) to be added to the master according to the objective function. Consider for example the problem P_2, it can be implemented as a union over explanations of sub-problems whoses sum of their optimum exceed the best known upperbound; For P_1, as a subset of non including explanations of sub-problems exceeding the coloring number of the master. Note that it can be overridden by a user also for generic problems P_0 and P_y to implement a specific heuristic to manage cuts such as keeping only the smallest ones or enough disjoint ones.
- *computeUb(int k)* (line 16): computes an upper bound on z_k of sub-problem k according to $\overline{y}$ and the currently known z_i with $i < k$. In the case $k = 0$ (the master problem) it computes the upper bound of the overall objective

function z if all bounds of sub-problems are known (every sub-problem was feasible).

At each iteration, a lower bound is obtained once the master problem has been solved. The algorithm stops once the lower bound meets the upper bound computed after the slaves. One can notice that the upper bound does not necessarily follow a decreasing trend whereas the lower bound is only growing ensuring the termination of the algorithm as long as variables have finite domains. Note that when considering the generic problems P_{xy}, the sub-problems have to be solved to optimality to provide the explanations.

Let us use our example problem P_2 to illustrate how algorithm 1 works. Notice that once y_1 and y_2 are instantiated, we get the following sub-problems :

$\text{SP1 : Min } sz_1 = 5x_3 + x_4$	$\text{SP2 : Min } sz_2 = 4x_1 + 2x_2$
$3x_3 + 2x_4 \geq 4 - \overline{y_2}$	$5x_1 + x_2 \geq 5 - 2\overline{y_1} - \overline{y_2}$
$occurrence(0, \{x_3, x_4, \overline{y_2}\}) = 1$	$x_1 \neq \overline{y_2}$
$occurrence(0, \{x_3, x_4, \overline{y_1}\}) = 1$	

Iteration 1	$(\overline{y_1}, \overline{y_2}) = (0,0)$	
	$sz_1^* = 6$	$expl(sz_1 \geq 6) = \{y_2 = 0\}$
	$sz_2^* = 4$	$expl(sz_2 \geq 4) = \{y_2 = 0\}$
	$z = 10$	$y_1 + y_2 < 10,\ y_2 \neq 0$ added
Iteration 2	$(\overline{y_1}, \overline{y_2}) = (1,1)$	
	$sz_1^* = 2$	$expl(sz_1 \geq 2) = \{y_2 = 1\}$
	$sz_2^* = 4$	$expl(sz_2 \geq 4) = \{y_1 = 1, y_2 = 1\}$
	$z = 8$	$y_1 + y_2 < 8, y_1 \neq 1 \vee y_2 \neq 1$ added
Iteration 3	$(\overline{y_1}, \overline{y_2}) = (2,1)$	
	$sz_1^* = 2$	$expl(sz_1 \geq 2) = \{y_2 = 1\}$
	$sz_2^* = 0$	$expl(sz_2 \geq 0) = \{y_1 = 2, y_2 = 1\}$
	$z = 5$	$y_1 + y_2 < 5, y_1 \neq 2 \vee y_2 \neq 1$ added
Iteration 4	$(\overline{y_1}, \overline{y_2}) = (1,2)$	
	$sz_1^* = 1$	$expl(sz_1 \geq 2) = \{y_2 = 2\}$
	SP2 infeas	$expl(sz_2 \geq 0) = \{y_1 = 1, y_2 = 2\}$
	$z = 5$	$y_1 \neq 1 \vee y_2 \neq 2$ added
Iteration 5	$(\overline{y_1}, \overline{y_2}) = (2,2)$	
	$sz_1^* = 1$	$expl(sz_2 \geq 2) = \{y_2 = 2\}$
		$y_2 \neq 2$ added

At iteration 1, one can notice that two explanations exist for $sz_1 \geq 6$: $\{y_2 = 0\}$ or $\{y_1 = 0\}$. Each one is sufficient to explain that $sz_1 \geq 6$. The explanation retrieved will depend of the propagation order of the two occurrence constraints. Let us say that $occurrence(0, \{x_3, x_4, y_2\})$ has been propagated first, removing the lower bound 0 of x_3 and x_4. After solving the sub-problem, we know that z cannot be less than 10 as long as y_2 is null. For iteration 5, the upper bound computed with $computeUb(2)$ is negative and the explanation for $SP2$ is empty. After iteration 5, the master problem becomes inconsistent and the algorithm terminates with the optimal solution $(y_1, y_2, x_3, x_4, x_1, x_2) = (2, 1, 0, 2, 0, 0)$.

5 The interest of decomposition in Constraint Programming

We describe in this section several applications and preliminary experimental results on the decomposition.

5.1 Accuracy of cuts on random structured binary problems

We study in this section the interest of taking advantage of a structure in the generic context of classical random binary problems. Random instances are usually characterized by the tuple $< n, d, p_1, p_2 >$ (we use the classical B model [1]) where n is the number of variables, d the unique domain size, p_1 the density of the constraint network and p_2 the tightness of the constraints. To build structured instances, we use the following model $< nm, ns, nbs, p_1, p_m, p_s, p_{ms} >$, where nm, ns are the number of variables of master and slaves, nbs, the number of slaves and p_m, p_s, p_{ms} the tightness of master, slaves and master-slave relationship. Figure 2 shows the results on structured problems RP_1 of the form $< 10, 10, nbs, 50\%, p_2, p_2, p_2 >$. So the same tightness is used for both master, slaves and master-slaves relationship. Each point represents the average computation time and number of nodes explored for our Benders decomposition algorithm (`Benders` from now on) and a classical search performed on the whole problem using the `mindom` heuristic (`mindom` from now on) over the same series of 100 instances randomly generated at the phase transition of RP_1.

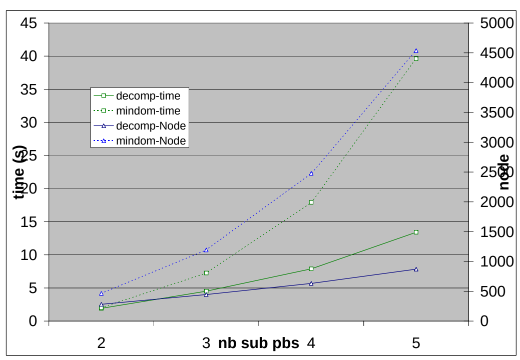


Fig. 2. Time and number of nodes needed for `mindom` and `Benders` (decomp) on structured problem $< 15, 6, nbs, 50\%, p_2, p_2, p_2 >$ at the transition phase for p_2 with nbs varying from 2 to 5 sub-problems.

As expected, the gap between the two techniques increases quickly with the degree to which the whole problem is structured. Surprisingly, a relative small number of sub-problems (compared to similar random experiments in pure linear programming of [5]) is needed to make `Benders` really more efficient. However this can only be due to the good ordering performed by the decomposition. So, we then tried to isolate the effect of the decomposition itself, *i.e* the learning of cuts and the study of several parts of the problem before coming back to the master. We apply a branching strategy denoted as *ad hoc branching* that uses the mindom heuristic among the master variables and each sub-problem in the same order as `Benders`. It is sufficient on the previous benchmark to close the gap between the two strategies.

So we investigated what kind of relationships could make our Benders decomposition more efficient. We discovered that the *ad hoc* strategy seems to experience difficulties just before the phase transition of the sub-problems (a tightness around 82-83% for 6 variables). Hard sub-problems seems to make `Benders` more efficient. We adapt the previous benchmark and fix this time the sub-problem tightness to 82% (the phase transition of sub-problems) to get problems of the following form $< 15, 6, nbs, 50\%, p_2, 82\%, 3\% >$. We then report in Figure 3 the number of unsolved problems for both *ad hoc* and `Benders` strategy for different tightness of the master and number of sub-problems within a time limit of 2 minutes. The master problem becomes in fact sometimes itself difficult only due to the interaction with slaves without instantiating them.

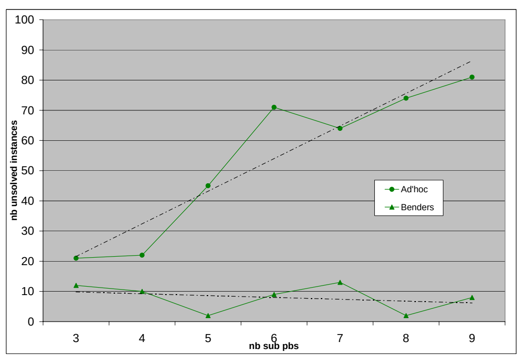


Fig. 3. Number of unsolved instances for `Benders` and *ad hoc strategy* on structured problems with sub-problems at the phase transition.

Benders strategy allow to solve more and more instances as the number of sub-problems grow. *Ad hoc* branching is in fact more subject to thrashing on the master tree search as it tries to solve each sub-problem one after another without looking at the rest of the problem which can sometimes help pruning efficiently the master as Benders do.

5.2 Application to real time scheduling

The approach described in this paper has been firstly applied in [3] for a real time scheduling problem. It proposed at that time to implement a dedicated *logical* duality to infer nogoods, tried to enforce the constraint model and finally performed an incremental resolution of the master problem using a dynamic backtracking algorithm for the master. The problem was to assign periodic tasks to processors in the context of fixed priorities preemptive scheduling (a task is periodically activated and can be preempted by a higher priority task). Real-time schedulability enforces the duration between the activation date of a task and its completion time to be bounded by its deadline. It guarantees that relative deadlines of tasks will be met.

This problem fits in the generic problems P_0 framework. Moreover, the set of variables x for the sub-problem is empty. Instead of providing a filtering algorithm within a global constraint for real-time schedulability, we attempted to derive accurate explanations for the unschedulability of each processor and therefore built a specific instance of the decomposition framework described here. So the partition of the problem was rather made among the constraints of the problem and the sub-problems considered had to evaluate the real-time schedulability of a set of tasks (jobs) on a processor (machine). In a sense, it was not far from the hybrid framework *Branch and Check* of [13] which consists in checking the feasibility of a delayed part of the problem in a sub-problem. In case of failure, it returned a minimal (regarding inclusion) subset of tasks responsible for this unschedulability. The computation of this minimal set was obtained by coupling analytical techniques from real time scheduling with a conflict based algorithm, QuickXplain [10]. It is a good specific example of application as sub-problems were polynomials and the cuts generated were quite efficient because we could compute the minimal ones. The whole scheme could take advantage of polynomiality of sub-problems as well as symmetry among processors (a cut derived on one processor was also valid for any processor of the system).

6 Conclusion

We have investigated in this paper how to derive logic Benders cuts using an explanation based framework for Constraint Programming. Accuracy of cuts using explanations was nevertheless questionnable. Indeed, remaining sub-problems are not polynomial (compared to a traditional MILP approach for Benders and assuming that LP is polynomial) and explanations constitute a weaker cut as a lazy computation is used. We therefore tried to show on random problems that explanations could provide a relevant cut in case of hard sub-problems. We attempted to demonstrate how such a decomposition framework could be embedded in constraint solvers to take advantage of explicit or implicit structures within a given problem/instance. Moreover, we believe that the presence of *backdoors* [11] or subset of variables exhibiting a strong impact over the whole problem could be explicitly and efficiently used by such an approach. Our next step is to apply the technique on hard academical problems and we are currently investigating how hard latin square instances could be decomposed.

Numerous perspectives exist. One promising application not described in Section 5 is Stochastic Constraint Programming [17] as Benders decomposition is used as an efficient way of tackling scenario-based approach for those problems. A current limitation for this technique is to be able to consider any kind of recourse and not only linear recourse. Concerning the whole scheme, the management of a large base of explanations could be improved considering that lot of variables are shared among explanations. Finally, indicators based on impact computation [16] could be used to automatically discover relevant master and slaves problems.

References

1. D. Achlioptas, L. Kirousis, E. Kranakis, D. Krizanc, M. Molloy, and Y. Stamatiou. Random constraint satisfaction: a more accurate picture. In *Proceedings CP 1997*, pages 121–135, Linz, Austria, 1997.
2. J. F. Benders. Partitionning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.
3. Hadrien Cambazard, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, Narendra Jussien, and Yvon Trinquet. Decomposition and learning for a real time task allocation problem. In *Proceedings CP 2004*, pages 153–167, 2004.
4. Bruno de Backer and Henri Béringer. Intelligent Backtracking for *CLP* Languages: An Application to *CLP($\mathcal{R}$)*. in Proceedings of ILPS'91: Proceedings International Logic Programming Symposium, pages 405–419, 1991.
5. J.N. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96:33–60, 2003.
6. Vipul Jain and I. E. Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing*, 13:258–276, 2001.
7. Narendra Jussien. *The versatility of using explanations within constraint programming*. Habilitation thesis, Université de Nantes, France, 2003. also available as RR-03-04 research report at École des Mines de Nantes.
8. Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.
9. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002.
10. Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle, WA, USA, August 2001.
11. Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *Proceedings IJCAI 2003*, 2003.
12. J.N. Hooker, G. Ottosson, E. S. Thorsteinsson, and H. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review, special issue on AI/OR*, 15(1):11–30, 2000.
13. Erlendur S. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *CP'01*, 2001.
14. T. Benoist, E. Gaudin, and B. Rottembourg. Constraint programming contribution to benders decomposition: A case study. In *CP'02*, pages 603–617, 2002.
15. A. M. Geoffrion. Generalized Benders Decomposition. *Journal of Optimization Theory And Practice*, Vol. 10, No. 4, 1972.
16. H. Cambazard, N. Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05), 2005.

17. S. Manandhar, S. A. Tarim, T. Walsh. Scenario-Based Stochastic COnstraint Programming. Proceedings of IJCAI-2003, Acapulco, Mexico, 2003, 257-262.

A Hybrid Benders' Decomposition Method for Solving Stochastic Constraint Programs with Linear Recourse

S. Armagan Tarim[1] and Ian Miguel[2]

[1] University College Cork, Cork Constraint Computation Centre, Cork, Ireland,
at@4c.ucc.ie, http://www-users.cs.york.ac.uk/~at/

[2] University of St.Andrews, School of Computer Science, St.Andrews, Scotland,
ianm@dcs.st-and.ac.uk, http://www-users.cs.york.ac.uk/~ianm/

Abstract. We adopt Benders' decomposition algorithm to solve scenario-based Stochastic Constraint Programs (SCPs) with linear recourse. Rather than attempting to solve SCPs via a monolithic model, we show that one can iteratively solve a collection of smaller sub-problems and arrive at a solution to the entire problem. In this approach, decision variables corresponding to the initial stage and linear recourse actions are grouped into two sub-problems. The sub-problem corresponding to the recourse action further decomposes into independent problems, each of which is a representation of a single scenario. Our computational experience on stochastic versions of the well-known template design and warehouse location problems shows that, for linear recourse SCPs, Benders' decomposition algorithm provides a very efficient solution method.

1 Introduction

Stochastic constraint programming (*SCP*, see [12, 15]) extends constraint programming to deal with both *decision* variables, which can be set by the decision-maker, and *stochastic* variables, which follow some discrete probability distribution function. This framework is designed to model a wide variety of decision problems involving uncertainty and probability. Examples include nurse rostering given an uncertain workload and constructing a balanced bond portfolio.

Tarim *et al.* [15] provide a semantics for stochastic constraint programs based on *scenarios*, where a scenario is a possible set of values for the stochastic variables. Based on this semantics, they compile stochastic constraint programs down into conventional (non-stochastic) constraint programs. The advantage of this compilation is that existing constraint solvers can be used without modification. However, the number of scenarios grows exponentially with the number of decision stages, where each stage consists of a set of decision variables and a set of stochastic variables whose combined assignments determine the structure of the next stage. Tarim *et al.* propose a number of scenario reduction algorithms to reduce the scenario tree considered. These algorithms determine a subset of scenarios and a redistribution of probabilities relative to the preserved scenarios. Generally, however, this approach yields sub-optimal solutions.

This paper presents Benders' decomposition (*BD*) algorithm as an optimisation method for stochastic constraint programs with linear recourse (*SCPwLR*). *SCPwLR* constitutes an important subgroup of stochastic constraint programs. In this type of SCP the initial stage decisions involve discrete variables, whereas the following recourse actions comprise only continuous decision variables. A typical example of *SCPwLR* is the Warehouse Location Problem (WLP) (see [10], [11] and [13]) with stochastic demand. In Section 5, the capacitated version of the WLP with stochastic demand is addressed, along with a stochastic version of the Template Design problem [14], and the computational performance of Benders' algorithm is investigated.

The paper is organised as follows. Section 2 provides background, including an overview of Benders decomposition. Section 3 introduces the linear recourse stochastic constraint programs and an application of BD to such programs. Section 4 gives an illustrative example of BD applied to *SCPwLR* by means of the classical news vendor problem. In the following section, the computational efficiency of BD is investigated. Section 6 concludes the paper and points out important future work.

2 Background

This section gives the necessary background detail in stochastic constraint programming and Benders decomposition. We begin with the former.

2.1 Stochastic Constraint Programming

A stochastic constraint satisfaction problem [12] consists of a 6-tuple $\langle \mathcal{X}, \mathcal{S}, \mathcal{D}, \mathcal{P}, \mathcal{C}, \theta \rangle$. $\mathcal{X}$ is a set of decision variables, and $\mathcal{S}$ is a set of stochastic variables. $\mathcal{D}$ is a function mapping each element of $\mathcal{X}$ and each element of $\mathcal{S}$ to a domain of potential values. A decision variable $x \in \mathcal{X}$ is *assigned* a value from its domain. $\mathcal{P}$ is a function mapping each element of $\mathcal{S}$ to a probability distribution for its associated domain. $\mathcal{C}$ is a set of constraints, where a constraint $c \in \mathcal{C}$ on variables $x_i, \ldots, x_j$ specifies a subset of the Cartesian product $\mathcal{D}(x_i) \times \ldots \times \mathcal{D}(x_j)$ indicating mutually-compatible variable assignments. The subset of $\mathcal{C}$ that constrain at least one variable in $\mathcal{S}$ are *chance* constraints. θ is a function mapping each chance constraint to the interval [0,1], indicating the fraction of scenarios in which the constraint must be satisfied. Note that a chance constraint with a threshold of 1 is equivalent to a hard constraint.

A stochastic CSP consists of a number of *decision stages*. In a one-stage stochastic CSP, the decision variables are set before the stochastic variables. In an n-stage stochastic CSP, $\mathcal{X}$ and $\mathcal{S}$ are partitioned into n disjoint sets, $\mathcal{X}_1, \ldots, \mathcal{X}_n$ and $\mathcal{S}_1, \ldots, \mathcal{S}_n$. To solve an n-stage stochastic CSP an assignment to the variables in $\mathcal{X}_1$ must be found such that, given random values for $\mathcal{S}_1$, assignments can be found for $\mathcal{X}_2$ such that, given random values for $\mathcal{S}_2$, ..., assignments can be found for $\mathcal{X}_n$ so that, given random values for $\mathcal{S}_n$, the hard constraints are satisfied and the chance constraints are satisfied in the specified fraction of scenarios. As noted in the introduction, the *SCPwLR* is a 2-stage stochastic CSP in which the domains of $\mathcal{X}_1$ are discrete, but the domains of $\mathcal{X}_2$ are continuous. Furthermore, the constraints on $\mathcal{X}_2$ are all linear.

2.2 Benders Decomposition

Although Benders' decomposition algorithm dates back to the 1960s and there is now a sizeable OR literature in this area extending the original approach, it has only recently been used by the constraint programming community in developing hybrid models. The reader is directed to Benoist et al. [3], Xia et al. [17], Eremin and Wallace [4], Hooker and Ottosson [7], Thorsteinsson [16], Jain and Grossmann [9] for applications of BD in constraint programming.

Benders decomposition [2, 5] was presented for solving models of the type:

$$\max\{c^T x + f(y)|Ax + F(y) \leq b, x \in \mathbb{R}^p_+, y \in S\} \tag{1}$$

where $x \in \mathbb{R}^p_+$ (the p-dimensional non-negative Euclidean space), $y \in \mathbb{R}^q$, and S is an arbitrary subset of $\mathbb{R}^q$. Furthermore, A is an (m, p) matrix, $f(y)$ is a scalar function and $F(y)$ an m-component vector function both defined on S, and b and c are fixed vectors in $\mathbb{R}^m$ and $\mathbb{R}^p$, respectively.

A key concept in Benders' algorithm is that of partitioning the variables into two sets – x and y – and projecting the problem onto the complicating variables, y. Benders' method decomposes this model in such a way that it can be solved as an alternating sequence of linear programs and programs of "complicating" variables. In the case of Eq.(1), once y is fixed to $\bar{y}$, the initial linear program is:

$$f(\bar{y}) + \min\{(b - F(\bar{y}))\lambda|A^T\lambda \geq c, \lambda \in \mathbb{R}^m_+\}, \tag{2}$$

In other words, the algorithm partitions the given problem in Eq.(1) into two such subproblems: a programming problem (which may be linear, non-linear, discrete, etc.) defined on S, and a linear programming problem defined in $\mathbb{R}^p_+$. An example is the mixed-integer programming problem in which certain variables may assume any value on a given interval, whereas others are restricted to integral values only. Then, in order to avoid the laborious calculation of a complete set of constraints for the feasible region in the first problem, a multi-step procedure is designed leading, in a finite number of steps, to a set of constraints determining an optimum solution of the given problem.

The classic BD algorithm was proposed for mixed-integer linear programming problems, the cut generation of which is based on the duality theorem of linear programming. The algorithm functions as follows: It determines trial values for the addressed problem by solving a program called the master problem – the program of complicating variables. The cost of this trial plan is determined using the so-called slave problem, Eq.(2). The slave problem also calculates dual multipliers, λ, which measure the marginal change in the trial plan. These dual multipliers are used to form new constraints that are added to the master problem, which is then re-solved to determine a new trial plan. The process continues alternately solving the master and slave problems, until the algorithm has found an optimal plan or one that is within an acceptable tolerance of optimality.

3 Benders Decomposition Applied to *SCPwLR*

In this section, BD is proposed as a solution method for *SCPwLR*. We consider only the 2-stage *SCPwLR*, but the method can easily be extended to address multi-stage *SCPwLR* without loss of generality. The method described is a *hybrid* since it requires a collaboration of CP and LP methods.

As mentioned, BD partitions the decision variables into two sets, x and y. For *SCPwLR*, partitioning is made with respect to the two decision stages: the first stage decision variables, which constitute the set of complicating variables y, form a CP model, and, since the recourse action is assumed to be linear, the second stage decision variables form an LP model. Hence, the "master" problem is a CP, whereas the "slave" problem, corresponding to the scenarios and the recourse actions taken, is an LP. In this context, Benders decomposition achieves separability of the second stage decisions, solving a separate LP for each scenario.

Consider the following 2-stage *SCPwLR*: min $\{f(y)+\sum_{k=1}^{K} p_k Q_k(y)|y \in Y\}$. Here, k indexes the finitely-many scenarios, with p_k the probability of scenario k. The first-stage variables y are set before the scenario is observed. After the kth scenario is observed, the set of second-stage decision variables x_k are set. The cost (assumed to be linear) of the second stage in scenario k is $Q_k(y) = \min\{q_k x|W_k x = h_k - T_k y, x \geq 0\}$. That is, x is a recourse, which must be chosen so as to satisfy some linear constraints in the least costly way.

We assume that recourse is *complete*, i.e., for any choice of y and scenario, there is always a non-empty set of x, $\{x|W_k x = h_k - T_k y, x \geq 0\} \neq \emptyset$. The objective is to minimize the expected total costs of both stages.

The deterministic equivalent model is a large-scale problem, which simultaneously selects the first-stage variables y and the second-stage variables x_k for every scenario k.
SCPwLR Model:

$$z = \min \ \{f(y) + \sum_{k=1}^{K} p_k q_k x_k \ | \ T_k y + W_k x_k = h_k, \quad x_k \geq 0, \quad y \in Y\} \tag{3}$$

3.1 Independent subproblems

Given an arbitrary first stage decision $\overline{y}$, define a function $Q_k(\overline{y})$ equal to the optimum of the second stage for each scenario $k = 1, ..., K$:
Slave (*Primal*) *Model*:

$$Q_k(\overline{y}) = \min \ \{q_k x_k \ |W_k x_k = h_k - T_k \overline{y}, \quad x_k \geq 0\} \tag{4}$$

Now, an upper bound on the optimal value of z, defined in (3), is:

$$f(\overline{y}) + \sum_{k=1}^{K} p_k Q_k(\overline{y}) \tag{5}$$

Under the assumption of complete recourse, the linear programming dual of the second-stage problem for scenario k, as given in (4), is the linear program:
Slave (*Dual*) *Model*:

$$Q_k(\overline{y}) = \max \ \{(h_k - T_k\overline{y})\lambda_k \ | \ W_k^T\lambda_k \leq q_k, \quad \lambda_k \text{ free}\} \tag{6}$$

Note that the constraints are now independent of y; in other words, the feasible region is not affected by the choice of y. Denote by $\Lambda_k = \{\lambda_k | W_k^T\lambda_k \leq q_k\}$ the polyhedral feasible region of the second-stage problem for scenario k. Denote by $\widehat{\lambda}_k^i$ the ith extreme point of Λ_k, $i = 1, ..., I_k$, where I_k is the total number of extreme points of the problem for scenario k. By enumerating the large, but finite, number of extreme points of Λ_k, we can write, $Q_k(y) = \max_{i=1,\ldots,I_k}\{\widehat{\lambda}_k^i(h_k - T_k y)\}$, which demonstrates that $Q_k(y)$ is a piecewise-linear convex function.

3.2 Complete and Partial Master Problems

Benders' "complete master problem" then uses this representation of $Q_k(y)$ to provide an alternative method for evaluating z,
Complete Master Model:

$$z = \min \ \{f(y) + \sum_{k=1}^{K} p_k \max_{i=1,\ldots,I_k} \{\widehat{\lambda}_k^i(h_k - T_k y)\} \ | \ y \in Y\} \tag{7}$$

While it is possible in principle to solve the problem using Benders' complete master problem, in practice the magnitude of the number of dual extreme points makes it prohibitively expensive. However, if a subset of the dual extreme points of Λ_k are available then we obtain an underestimate of $Q_k(y)$, which we denote:

$$Q_k^{'}(y) = \max_{i=1,\ldots,M_k} \{\widehat{\lambda}_k^i(h_k - T_k y)\} \tag{8}$$

where $M_k \leq I_k$.

Using dual information obtained after M evaluations of $Q_k(y)$, we obtain a "partial master problem", which provides a lower bound on the solution of z:
Partial Master Model:

$$\min \ \{f(y) + \sum_{k=1}^{K} p_k \max_{i=1,\ldots,M_k} \{\widehat{\lambda}_k^i(h_k - T_k y)\} \ | \ y \in Y\} \tag{9}$$

However, there is no guarantee that partial master problem yields a bounded solution. If it produces an unbounded solution then the direction of the *extreme ray* must be determined and the Benders cut, $0 \geq \widehat{\lambda}_k^r(h_k - T_k y)$, must be added accordingly to bound the unbounded polyhedral set.

Benders' algorithm solves the current "partial master problem", obtaining a new $\overline{y}$ (a new trial solution) and, an underestimate $\sum_{k=1}^{K} p_k Q_k^{'}(\overline{y})$ of the associated expected second-stage cost.

Algorithm 1: BD-SCPwLR

input : Set of scenarios, S
$z = \min\{f(y) + \sum_{\forall k \in S} p_k q_k x_k | T_k y + W_k x_k = h_k, x_k \geq 0, y \in Y\}$

output : $\{z^*, y^*, x_k^*\}$

begin
$\bar{y} \leftarrow$ an initial feasible y
$c \leftarrow f(\bar{y})$
$up \leftarrow \infty$
$low \leftarrow -\infty$
$\forall k \in S,\ \ Cut_k \leftarrow \emptyset$
while $up - low > \epsilon$ **do**
  for $k \in S$ **do**
    $Slave_k \leftarrow Q_k(\bar{y}) = \max\{(h_k - T_k\bar{y})\lambda_k | W_k^T \lambda_k \leq q_k, \lambda_k \text{ free}\}$
    $\widehat{\lambda}_k \leftarrow Solve(Slave_k)$
    $Cut_k \leftarrow Cut_k \cup \{\widehat{\lambda}_k(h_k - T_k y)\}$
  $c \leftarrow f(\bar{y}) + \sum_{\forall k \in S} p_k Q_k(\bar{y})$
  if $c < up$ **then** $up \leftarrow c$
  $Master \leftarrow \min\{f(y) + \sum_{\forall k \in S} p_k \max\{Cut_k\} | y \in Y\}$
  $\bar{y} \leftarrow Solve(Master)$
  $c \leftarrow \min\{f(\bar{y}) + \sum_{\forall k \in S} p_k \max\{Cut_k | y = \bar{y}\}\}$
  if $c > low$ **then** $low \leftarrow c$
return $z^* \leftarrow low$, $y^* \leftarrow \bar{y}$, $\forall k \in S \quad x_k^* \leftarrow$ dual vars from $Solve(Slave_k)$

end

3.3 Iterative Process

The actual expected second-stage cost, $\sum_{k=1}^{K} p_k Q_k(\overline{y})$, is then evaluated by solving the second-stage problem for each scenario. Additional terms, in the form of $\{\widehat{\lambda}_k(h_k - T_k y)\}$, are added to the partial master problem to complete the iteration. Each additional term is actually another cut added to the model.

At each iteration of Benders' algorithm, then, the slave problem solution provides an upper bound for z, and, the partial master solution provides a lower bound for z. It can be proved that the above iterative procedure terminates in a finite number of iterations. An attractive feature of this algorithm is the availability of upper and lower bounds on the optimal objective value, which both converge to this value as optimality is achieved. The upper bound is generated by a sequence of feasible solutions to the problem, so the best of these may be taken as a solution if the procedure is terminated short of optimality.

The complete BD algorithm for the *SCPwLR* is presented in Algorithm 1.

4 An Illustrative Example: News Vendor Problem

We now illustrate the *SCPwLR* solution method using a modified version of the well-known "news vendor problem", a stochastic inventory replenishment problem which can be described as follows. Given a stochastic distribution for the demand of a product, what is the optimal order quantity, y^*, if only one order can be placed before actual demand is observed. Assume we have no initial inventory. The decision maker has to order an amount $y \geq 0$ at unit price $c = 3$. Unsold goods can be returned to the supplier with a salvage value of $v = 1$. In case of high demand, the firm expedites to avert an impending stockout with a cost of $e = 8$ per unit of excess demand. The maximum amount of units that can be ordered is initially $R = 20$. However, the quota can be increased by $r = [11, 13, 15, 17]$ in a nested manner, following a fixed payment of $h = [10, 12, 14, 16]$ which is also nested. Therefore, the maximum quota of 76 units can be obtained with a cost of 52. Demand is a discrete random variable denoted by ξ. ξ takes the values of $\{15(0.1), 25(0.2), 35(0.3), 45(0.3), 55(0.1)\}$, in which the values in parentheses are the probabilities, p.

A stochastic constraint program for the above problem is as follows:

$$\begin{aligned}
&\min cy + \textstyle\sum_{i=1}^{4} h_i k_i + \sum_{s=1}^{5} p_s(ex_{1s} - vx_{2s}) \\
&\text{s.t.} \\
&y + x_{1s} - x_{2s} = \xi_s, \quad s = 1, ..., 5 \\
&\textstyle\sum_{i=1}^{4} r_i k_i + 20 >= y \\
&k_i = 0 \Rightarrow k_{i+1} = 0, \quad k_i \in \{0, 1\}, \quad i = 1, ..., 3
\end{aligned}$$

where x_{1s} and x_{2s} denote expedited order and salvage amounts, respectively.

In the above model, y and k_i denote the first-stage decision variables, x_{1s} and x_{2s} are the second-stage decision variables, where s denotes a scenario, and the cost term $\sum_{s=1}^{5} p_s(ex_{1s} - vx_{2s})$ corresponds to the expected recourse cost. This problem has linear recourse. This partitioning of decision variables yields the following master and slave problems:

Master Problem

$$\begin{aligned}
&\min \quad cy + \textstyle\sum_{i=1}^{4} h_i k_i + \\
&\textstyle\sum_{s=1}^{5} p_s \max\{\widehat{\lambda}_s(\xi_s - y)\} \\
&\text{subject to} \\
&\textstyle\sum_{i=1}^{4} r_i k_i + 20 >= y \\
&k_i = 0 \Rightarrow k_{i+1} = 0 \quad i = 1, ..., 3 \\
&k_i \in \{0, 1\}
\end{aligned}$$

Slave Problems $\quad s = 1, ..., 5$

$$\begin{aligned}
&\max \ (\xi_s - \overline{y})\lambda_s \\
&\text{subject to} \\
&\lambda_s \leq e \\
&\lambda_s \geq v \\
&\lambda_s \text{free}
\end{aligned}$$

Table 1 presents the step-by-step application of Benders' algorithm to our *SCPwLR* problem.

We start with a feasible solution of $y = 0$ and $k = [1, 1, 1, 1]$, which is actually the worst possible ordering policy. At the first iteration 5 independent trivial slave problems are solved. The optimal solutions are $\lambda_{1,...,5} = 8$. Next we calculate the upper bound that these solutions imply. Eq.(5) provides an upper bound on the optimal solution to the original problem, which is 340 here. At the second step of the first iteration the partial master problem is solved with the added Benders cuts, $\max\{8(\xi_s - y)\}$, giving a lower bound of -40. Subsequent iterations of the algorithm can be followed from Table 1. In this instance the lower and upper bounds converge to the optimal solution of 155.8 in five steps.

Problem	Iteration	y^*	k^*	λ^*	obj value	Lower Bound	Upper Bound
Initial values	0	0	[1,1,1,1]	–	–	$-\infty$	$+\infty$
Slave	1	–	–	[8,8,8,8,8]	340	–	340
Master	1	76	[1,1,1,1]	–	-40	-40	–
Slave	2	–	–	[1,1,1,1,1]	240	–	240
Master	2	44	[1,1,0,0]	–	155.8	155.8	–
Slave	3	–	–	[1,1,1,8,8]	155.8	–	155.8

Table 1. Steps of Benders' Decomposition

5 Computational Experiments

This section presents computational results of using Benders' Decomposition in *SCPwLR* on the capacitated version of the Warehouse Location Problem (CWLP) (see [1], [6], [10], [11] and [13]), and stochastic version of the Template Design Problem [14].

5.1 Stochastic Capacitated Warehouse Location Problem

Let $I = \{1, ..., N\}$ be potential warehouse locations to supply a uniform product. A facility can be opened in any location $i \in I$. Opening a facility at location i has a non-negative fixed cost, f_i. Each open facility i can provide a limited amount C_i of commodity. Let $J = \{1, ..., M\}$ denote stores that are supplied by the open warehouses. For any pair (i, j) given, there is a unit production and transportation cost $g_{ij} \geq 0$. Each store can be supplied by exactly one warehouse. The probabilistic customer demands, ξ_j, are only known following stores' order placements to warehouses. Stores incur a fixed penalty cost for each unit they backlog, e_j, and fixed holding cost for each unit of excess inventory, h_j, they have. The goal is to determine a subset of the set of potential warehouse locations at which to operate warehouses, and an assignment of all clients to these facilities so as to minimize the expected total cost of operating the system. This problem is a generalisation of the well-known set covering problem and, therefore, an NP-hard problem in the strong sense.

A constraint model for the deterministic version of the above problem is given in [8]. This model is extended to comply with the stochastic demand assumption. The decision variables are:

- $k_i \in \{0, 1\}$ denoting whether warehouses i is in operation or not,
- $u_j \in I$ showing the supplier for store j,
- $y_{u_j,j} \geq 0$ is the amount warehouse u_j delivers to store j,
- $x^+_{j,s}$ and $x^-_{j,s}$ denote the excess inventory and shortage, respectively, at the end of the period at store j, if scenario $s \in S$ is realised.

The certainty equivalent CP model is

$$\begin{aligned}
&\min \ \sum_{j\in J} g_{u_j,j} y_{u_j,j} + \sum_{i\in I} f_i k_i + \quad \sum_{j\in J}\sum_{s\in S} p_s (e_j x^-_{j,s} + h_j x^+_{j,s})\\
&\text{s.t.}\\
&k_{u_j} = 1 && \forall j \in J\\
&\sum_{j\in J} y_{u_j,j}(u_j = i) \le C_i && \forall i \in I\\
&y_{u_j,j} - \xi_{j,s} = x^+_{j,s} - x^-_{j,s} && \forall j \in J, \forall s \in S
\end{aligned}$$

In the above formulation, the first stage (k_i, u_j, $y_{u_j,j}$) and second stage ($x^+_{j,s}$, $x^-_{j,s}$) decision variables are employed to partition the given model. In this case, the master and slave problems for the stochastic CWLP are defined as

Master Problem

$\min \ \sum_{j\in J} g_{u_j,j} y_{u_j,j} + \sum_{i\in I} f_i k_i +$

$\sum_{s\in S} \max\{p_s \sum_{j\in J} \widehat{\lambda}_{j,s}(\xi_{j,s} - y_{u_j,j})\}$

s.t.

$k_{u_j} = 1, \ \forall j \in J$

$\sum_{j\in J} y_{u_j,j}(u_j = i) \le C_i, \ \forall i \in I$

Slave Problems $\forall s \in S, \forall j \in J$

$\max \ (\xi_{j,s} - \overline{y}_{u_j,j})\lambda_{j,s}$

s.t.

$-h_j \le \lambda_{j,s} \le e_j$

$\lambda_{j,s}$ free

Although the slave problems are expressed in a linear program structure, in which $\lambda_{j,s}$ are dual decision variables, the simplicity of the resultant independent linear programs can be exploited to solve the problems to optimality without resorting to Linear Programming. The optimal solution to any sub-problem is in the form:

$$\lambda^*_{j,s} = \begin{cases} -h_j & \text{if} \ \ \xi_{j,s} - \overline{y}_{u_j,j} \le 0 \\ \ \ e_j & \text{if} \ \ \xi_{j,s} - \overline{y}_{u_j,j} \ge 0 \end{cases}, \quad \forall s \in S, \forall j \in J \tag{10}$$

The objective function of the master problem represents a "multiple-cut" approach to the Benders' decomposition. In this version, each scenario contributes to the cut generation process with a single cut. This excessive number of cuts may increase the size of the master problem to the point at which finding a solution is prohibitively long. Alternative approach is to aggregate scenario cuts into one, so as to reduce the detrimental effect of size on the solution performance. This so-called "single-cut" version is then in the form of $\Phi \ge \max\{\sum_s \sum_j p_s(\xi_{j,s} - y_{u_j,j})\widehat{\lambda}_{j,s}\}$, which leads to the objective function:

$$\min \ \sum_{j\in J} g_{u_j,j} y_{u_j,j} + \sum_{i\in I} f_i k_i + \Phi. \tag{11}$$

Although the single-cut approach does not provide cuts as strong as the multiple-cut approach, it is still computationally less expensive.

Experiments were conducted on 50 SCWLP instances using a 1.2 GHz computer. These instances vary mainly in numbers of warehouses (between 2 to 10), stores (between 2 to 5) and scenarios (between 4 to 1331). The holding cost is taken as 1 throughout the experiments. Three different shortage cost values are used: 4, 6 and 8. The warehouse capacities are chosen in the range of [20,110]; fixed operating costs, [30,120]; random demands, [5,50]; transportation costs, [1,4]. We assume that the number of states random demand variables have in one instance is the same for all stores. Given n random variable states and j stores, the number of scenarios in an instance is n^j.

In the first step of the experiment, instances were modelled as a stochastic constraint program whose formulation is given above. The SCP models were solved using OPLStudio Solver6.0 with a time limit of 1 hour. The upper bounds for variables are chosen as follows: for y_j and x_j^-, maximum possible demand for store j; for x_j^+, the difference between the maximum and minimum demand values. The variable ordering heuristic employed assigns u_j, k_{u_j}, and $y_{u_j,j}$ in that order. The solution time (in seconds) and the number of nodes visited during search are given in Table 2 under the column heading "SCP".

Step two of the experiment repeated the first with an embedded linear relaxation. The linear relaxation enables solver to produce lower/upper bounds by solving an LP at each node of the CP search tree. The constraint solver also uses the LP solution to guide its search. The results of using embedded linear relaxation are given in Table 2 in the next two columns with a heading "SCP with LR". We also tried using linear relaxation with the variable ordering used in the first step. However, the results show that in this case linear relaxation does not prune the search space further, but incurs overhead.

In the next step of the experiment, Benders decomposition was applied to the test suite. The algorithm was implemented using ILOG Cplex9.0 and Solver6.0. The initial feasible solution was defined by adopting a no-order policy for all stores. However, since each store must be assigned to a warehouse, the least costly warehouse is operated and its fixed cost is incurred to be able to serve all stores. In this step, the "multiple-cut" version of BD was used. The results are given under the heading "BD with Multiple-Cuts". The results displayed in column "step", show the number of steps (solving a master or a slave problem defines a single step) BD takes before obtaining and proving the optimality of a solution. The column headed "Gap" gives the optimality gap for the first feasible solution, which is in the 5% gap. This experiment was repeated with "single-cuts". The results are listed under the heading "BD with Single-Cuts".

We experimented with two heuristics to test the utility of starting with a more informed solution. In heuristic-I, an expected value problem is designed to find an initial solution by replacing random demands with their maximum possible values. In heuristic-II, half the value of the maximum demands are taken as deterministic demand values. The deterministic CP models were solved to generate initial solutions. Results are presented in Table 2.

From Eq.(10), it is clear that there are only two possible values – either $-h_j$ or e_j – for any dual variables λ_j of the subproblems. Therefore, it is possible to built the "complete master" model, which can be solved to optimality without

No	# W	# S	# Scen	SCP		SCP with LR		BD with Multiple-Cuts					BD with Single-Cuts					Heuristic-I		Heuristic-II	
				Time	# Choice	Time	# Choice	Time	Step	Gap	Time	Step	Time	Step	Gap	Time	Step	Time	Step	Time	Step
1	2	2	4	0.07	361	0.22	634	0.07	5	2.7	0.07	4	0.03	5	2.7	0.03	4	0.04	8	0.02	6
2	2	2	9	0.22	1031	0.38	848	0.13	5	2.4	0.13	4	0.05	7	3.7	0.03	4	0.06	10	0.05	8
3	2	2	16	0.32	1687	0.56	1015	0.22	5	0	0.22	5	0.04	7	2.5	0.03	5	0.06	10	0.05	8
4	2	2	25	0.87	4464	0.74	1048	0.33	5	0	0.33	5	0.04	6	1.7	0.03	5	0.07	9	0.06	7
5	2	2	36	1.1	4913	1.2	1337	0.57	5	0	0.57	5	0.05	8	3.3	0.03	5	0.07	10	0.06	7
6	2	2	49	2.0	8068	2.3	1685	1.6	6	0.43	0.98	5	0.09	8	4.1	0.04	5	0.11	11	0.10	10
7	2	2	64	2.0	11190	2.5	2212	1.7	6	0.48	1.1	5	0.08	9	4.6	0.04	5	0.11	12	0.09	9
8	2	2	81	2.0	10724	3.3	2577	2.2	6	0.45	1.4	5	0.08	8	1.3	0.05	6	0.10	11	0.06	10
9	2	2	100	2.9	14669	4.3	3040	2.7	6	0.2	1.7	5	0.11	11	3.0	0.06	6	0.11	12	0.09	9
10	2	2	225	15	48301	39	15343	19	7	4.4	11	5	0.60	23	2.4	0.12	9	0.37	18	0.43	20
11	3	3	8	3.7	7598	11	36148	9.7	9	3.6	6.3	6	0.73	11	0.0	0.73	11	0.70	10	0.52	10
12	3	3	27	7.0	14798	24	37565	62	11	0.58	30	6	2.1	21	3.8	1.5	16	2.5	24	1.8	22
13	3	3	64	16	35607	52	36690	91	9	0.59	61	6	2.1	19	4.0	0.84	10	2.3	22	1.7	18
14	3	3	125	33	59942	128	42856	280	11	1.3	180	9	3.1	25	2.8	1.1	14	4.1	32	3.2	26
15	3	3	216	62	88078	250	51250	660	11	1.5	420	9	3.5	27	2.5	1.1	14	3.7	28	3.9	28
16	3	3	343	130	156102	390	54552	1000	11	1.3	640	8	3.8	28	3.9	1.1	14	3.5	28	3.1	32
17	3	3	512	450	363736	2200	250467	2600	11	0.82	1700	9	2.8	22	3.9	0.67	10	2.9	23	3.4	28
18	3	3	729	950	487978	3400	216757	–	–	0.33	3600	9	3.6	23	3.8	0.99	13	3.6	25	4.6	23
19	3	3	1000	1500	600508	–	–	–	–	–	–	–	4.8	25	3.1	1.3	12	4.4	25	4.6	21
20	3	3	1331	2300	797668	–	–	–	–	–	–	–	5.0	25	3.1	1.3	12	4.3	28	5.3	23
21	6	4	16	620	927903	800	1405123	1200	7	3.4	1200	6	13	9	3.1	13	8	4.6	3	4.2	4
22	6	4	81	2900	1040436	–	–	–	–	–	–	–	40	17	2.1	40	16	57	16	54	18
23	6	4	256	–	–	–	–	–	–	–	–	–	40	19	2.6	20	15	77	30	71	26
24	8	3	8	15	32817	29	70983	16	5	1.5	16	4	1.2	5	0.0	1.2	5	1.6	4	1.2	4
25	8	3	27	79	95022	64	81632	240	7	2.2	130	5	2.9	7	3.7	1.8	5	3.3	6	3.3	8
26	8	3	64	140	123451	110	80861	480	7	4.1	250	5	4.1	9	2.9	2.8	6	4.1	8	4.3	10
27	8	3	125	510	259604	370	152197	2600	7	0	2600	6	12	13	4.3	8.6	10	9.5	16	7.4	10
28	8	3	216	1600	713402	1300	341135	–	–	–	–	–	16	17	3.9	7.2	10	24	24	22	20
29	8	3	343	3100	1145120	1800	275315	–	–	–	–	–	14	19	4.8	5.6	11	29	26	16	16
30	8	3	512	–	–	3500	384398	–	–	–	–	–	31	24	4.4	9.9	13	32	26	39	26
31	8	4	16	610	698528	2000	3664085	850	5	0	850	5	9.0	5	0.0	9.1	5	16	4	8.1	4
32	8	4	81	–	–	–	–	–	–	–	–	–	21	6	2.9	13	5	27	5	16	5
33	8	4	256	–	–	–	–	–	–	–	–	–	110	15	4.5	58	11	200	18	140	14
34	8	5	32	–	–	–	–	–	–	–	–	–	41	5	0.0	41	5	67	3	54	4
35	8	5	243	–	–	–	–	–	–	–	–	–	480	8	3.7	160	5	530	5	460	7
36	9	3	8	13	28202	65	213637	40	6	0	40	6	2.1	9	4.1	2.1	9	1.1	4	0.65	4
37	9	3	27	60	79583	100	202190	350	8	1.7	200	7	4.1	10	4.1	2.8	9	3.1	7	2.8	7
38	9	3	64	140	114204	480	609580	960	9	4.8	440	6	8.1	17	1.1	6.4	14	9.6	20	8.7	16
39	9	3	125	340	189756	860	550976	–	–	4.8	1900	6	22	24	3.7	5.7	12	23	28	21	24
40	9	3	216	600	289801	1300	469528	–	–	–	–	–	18	24	3.4	5.1	12	22	30	19	26
41	9	4	16	710	1091206	1600	3387047	700	7	0	700	7	8.0	9	0.0	7.9	9	3.5	4	2.9	4
42	9	4	81	2700	972937	–	–	–	–	–	–	–	27	16	0.98	18	12	27	16	19	11
43	9	4	256	–	–	–	–	–	–	–	–	–	27	13	1.7	20	10	60	18	36	16
44	9	5	32	–	–	–	–	–	–	–	–	–	140	9	4.6	140	8	46	4	41	4
45	10	3	8	6.0	10025	21	51635	7.7	4	0	7.7	4	0.89	5	0.0	0.89	5	1.1	3	0.63	4
46	10	4	16	330	379947	790	1431662	240	4	0	240	4	5.1	5	0.0	5.1	5	9.4	4	4.2	4
47	10	4	81	2700	991063	–	–	–	–	–	–	–	23	15	4.7	18	12	31	16	18	12
48	10	4	256	–	–	–	–	–	–	–	–	–	44	19	5.0	32	14	65	18	47	19
49	10	5	32	–	–	–	–	–	–	–	–	–	97	7	0.0	97	7	37	4	28	4
50	10	5	243	–	–	–	–	–	–	–	–	–	310	11	4.4	310	10	170	5	220	8

Table 2. Experimental Results – Stochastic Capacitated Warehouse Location Problem

any Benders iteration, for this test problem. The complete master model is tried on these 50 instances. However, the observed computational performance is very discouraging. The solution times are on the average 100 times more than "BD with Single-Cuts" solution times, and 2 times more than "SCP".

The results in Table 2 show that monolithic SCP – either with variable ordering or linear relaxation – could not prove optimality in 10 instances under 1 hour. At termination, in only 2 instances the best-so-far feasible solutions were optimal. The results also point out another strong side of BD approach. If an optimality gap which is less than 5% is considered satisfactory, then the solution time is halved on average for the addressed test suite. It should also be noted that we obtained the optimal solutions in 8 out of 50 cases.

A multiple regression at confidence level 0.95 with a constant coefficient 0 gives us an insight to the BD algorithms' exact behavior on SCWL problem. The dependent variable is defined as the ratio of the solution times $st(.)$, $st(CP)/st(BD_{single})$ and independent variables of # warehouses, # stores, # scenarios. The adjusted R^2 is 0.83, which demonstrates that the overall regression model is meaningful. The regression coefficients are 4.76 (warehouses), -3.00 (stores) and 0.33 (scenarios). The corresponding t-statistics are 1.72, -0.49, 14.01, hence only the "scenarios" coefficient is significant. The results indicate that as the number of scenarios increases the performance of BD is more significant.

The two initial-solution heuristics give mixed results. In some instances the overhead of solving an additional CP model to obtain a better starting solution is not worth the effort. However, cheaper heuristics that exploit the unique structure of the problem addressed can be designed and the total solution time performance can be improved.

5.2 Stochastic Template Design Problem

The deterministic **template design problem** (prob002 in CSPLib) is described as follows. Given is a set of variations of a design, with a common shape and size and such that the number of required "pressings" of each variation is known. The problem is to design a set of templates, with a common capacity to which each must be filled, by assigning one or more instances of a variation to each template. A design should be chosen that minimises the total number of "runs" of the templates required to satisfy the number of pressings required for each variation. As an example, the variations might be for cartons for different flavours of cat food, such as fish or chicken, where ten thousand fish cartons and twenty thousand chicken cartons need to be printed. The problem would then be to design a set of templates by assigning a number of fish and/or chicken designs to each template such that a minimal number of runs of the templates is required to print all thirty thousand cartons. In the stochastic version of the problem, the demand for each variation is uncertain.

Proll and Smith address this problem by fixing the number of templates and minimising the total number of pressings [14]. We adopt their model herein, extending it to comply with the stochastic demand assumption. We use the following notation: N, number of variations; T, number of templates (T=2, for all instances of the test suit); S, number of slots on each template; K, number of scenarios (demand for each variation is uncertain); c_h, scrap cost; c_p, shortage cost; $a_{i,j}$, number of slots designated to variation i, on template j; R_j, number of required "runs" of template j; x_i, an auxiliary variable; $d_{i,k}$, demand for variation i in scenario k; $e_{i,k}$, total scrap (variation i in scenario k); $b_{i,k}$, total shortage (variation i in scenario k); p_k, probability of observing scenario k.

The certainty equivalent CP model is:

$$\begin{aligned}
&\min\ \textstyle\sum_{i=1}^{N}\sum_{k=1}^{K} p_k(c_p b_{ik} + c_h e_{ik}) \\
&\text{subject to} \\
&\textstyle\sum_{i=1}^{N} a_{ij} = S,\ \forall j \in \{1, ..., T\}, \\
&\textstyle\sum_{j=1}^{T} a_{ij} R_j = x_i,\ \forall i \in \{1, ..., N\}, \text{ and} \\
&x_i = d_{ik} + e_{ik} - b_{ik},\ \forall i \in \{1, ..., N\},\ \forall k \in \{1, ..., K\}.
\end{aligned}$$

In the above formulation, the first stage (a_{ij}, R_j) and second stage (x_i, e_{ik}, b_{ik}) decision variables are employed to partition the given model. In this case, the master and slave problems for the Stochastic Template Design Problem are defined as

Master Problem

$$\begin{aligned}
&\textstyle\sum_{k=1}^{K} \max\{\sum_{i=1}^{N} p_k(d_{ik} - x_i)\widehat{\lambda}_{ik}\} \\
&\text{subject to} \\
&\textstyle\sum_{i=1}^{N} a_{ij} = S,\ \ \forall j \in \{1, ..., T\} \\
&\textstyle\sum_{j=1}^{T} a_{ij} R_j = x_i,\ \ \forall i \in \{1, ..., N\}
\end{aligned}$$

Slave Problems,

$$\begin{aligned}
&\forall i \in \{1, ..., N\}, \forall k \in \{1, ..., K\} \\
&\max\ (d_{ik} - \overline{x}_i)\lambda_{ik} \\
&\text{subject to} \\
&-c_h \le \lambda_{ik} \le c_p,\ \lambda_{ik} \text{ free}
\end{aligned}$$

				SCP		Complete Benders		BD with Multiple-Cuts					BD with Single-Cuts				
No	N	S	Scen	Time	#choice	Time	#choice	Time	step	gap	Time	step	Time	step	gap	Time	step
1	3	6	10	340	846,584	190	845,660	4,400	13	0	4,400	13	9.5	33	4.1	7.9	29
2	3	9	10	1,400	3,401,506	780	3,400,897	–	–	–	–	–	8.1	31	4.8	7.4	26
3	4	2	10	230	703,957	140	694,949	3,100	27	4.0	2,100	25	62	43	3.7	42	36
4	4	3	10	470	1,153,468	270	1,145,436	5,500	23	0.6	3,100	21	41	39	4.7	26	32
5	4	4	10	1,900	4,165,262	1,100	4,159,062	–	–	–	–	–	190	49	4.3	130	40
6	4	5	10	690	1,630,615	380	1,627,743	–	–	–	–	–	380	41	3.4	380	36
7	4	6	10	–	–	5,700	20,909,785	–	–	–	–	–	670	52	3.5	510	44
8	4	7	10	–	–	–	–	–	–	–	–	–	820	53	4.8	600	40
9	4	8	10	–	–	–	–	–	–	–	–	–	1,800	50	4.4	1,000	37
10	4	9	10	–	–	–	–	–	–	–	–	–	5,900	49	3.3	4,900	43
11	4	6	11	–	–	6,800	21,302,918	–	–	–	–	–	480	51	4.1	410	44
12	4	6	12	–	–	–	–	–	–	–	–	–	1,100	52	4.6	640	41
13	4	6	13	–	–	–	–	–	–	–	–	–	190	58	4.9	160	43
14	4	6	14	–	–	–	–	–	–	–	–	–	1,300	57	3.3	760	42
15	4	6	15	–	–	–	–	–	–	–	–	–	790	53	3.5	780	42
16	4	6	16	–	–	–	–	–	–	–	–	–	1,800	55	4.4	1,500	40
17	4	6	17	–	–	–	–	–	–	–	–	–	540	54	4.8	370	38
18	4	6	18	–	–	–	–	–	–	–	–	–	290	53	3.7	230	40
19	4	6	19	–	–	–	–	–	–	–	–	–	590	61	3.9	500	49
20	4	6	20	–	–	–	–	–	–	–	–	–	400	59	4.2	360	40

Table 3. Experimental Results – Stochastic Template Design Problem

Experiments, summarised in Table 3, were conducted on 20 stochastic template design instances using an Intel Centrino 2GHz computer with 1GB RAM. Allowed solution time was 2 hours. A dash in the table indicates this time was exceeded without finding the optimal solution.

We compared four different solution methods: stochastic constraint programming (solved using Ilog OPLStudio and Solver 6.0), and three versions of Benders Decomposition (solved using Ilog Cplex 9.0 and Solver 6.0). In the first, the complete master problem, which can be solved to optimality without any Benders iterations, was built and solved using Solver 6.0. In the second and third, the multiple cut and single cut versions of BD were used. For the stochastic constraint program and complete Benders formulation, the number of choice points explored to find and prove optimality are shown. For the latter two Benders variations, the "step" column indicates the number of steps (solving a master or a slave problem is counted as a step) BD takes before proving optimality. The "Gap" column gives the optimality gap for the first feasible solution found with an optimality gap of 5% or less.

The results show that monolithic SCP can solve only the smaller instances in the time allowed. The performance of the complete Benders formulation is comparable, giving better times and solving two more instances. The performance of the two BD approaches is at polar opposites. The multiple-cut variant performs relatively poorly, solving fewer instances than SCP and taking much more time. However, the single-cut variant is clearly the strongest approach that we tested, solving all 20 instances.

6 Conclusion

This paper has aimed to enhance the effectiveness of the stochastic constraint programming framework by extending the use of the well-known Benders' decomposition algorithm, which has proved to be useful in mathematical programming, to solve stochastic constraint programs with linear recourse, *SCPwLR*. First and second stage decision variables are used to decompose the stochastic constraint program into master and slave problems. The unique structure of stochastic constraint programs, which is based upon a scenario tree representation, yields independent slave sub-problems, one for each scenario considered in the scenario tree. This natural slave problem decomposition has the obvious benefit of solving a set of small problems, and hence to a degree relieving the difficulty of dealing with a large scenario tree.

In our test problems, it was shown that the slave problem decomposes into trivial scenario problems, each of which is a boundary value problem and can be solved simply by checking the objective function coefficient and deciding whether the single decision variable takes the lower or the upper limit of its domain. Computational experiments confirmed the potential of Benders' decomposition method as an efficient solution algorithm for these problems.

An attractive feature of this algorithm is the availability of upper bounds and lower bounds produced by the slave and master problem solutions, respectively. The upper bound is generated by a sequence of feasible solutions to the problem, so the best of these may be taken as a solution if the procedure is terminated short of optimality. Furthermore, the best-so-far lower bound can be used to produce a metric for the optimality gap.

Although at present we consider only 2-stage *SCPwLR*, the ideas presented here can be generalised to n-stage *SCPwLR* without loss of generality. In this case, Benders' algorithm is applied in a recursive manner. We expect the performance discrepancy between the monolithic model and the decomposed model to be magnified as the number of stages grows. A further extension of this work should consider the case in which the linear recourse assumption is relaxed and recourse actions with complex structures are allowed. Hooker and Ottosson [7] provide a method for generating Benders cuts in such situations by generalising the linear programming dual of a sub-problem to an "inference" dual. We will adopt this approach for general stochastic CSP.

Acknowledgements: S. Armagan Tarim is supported by Science Foundation Ireland under Grant No. 03/CE3/I405 as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR) and Grant No. 00/PI.1/C075. Ian Miguel is supported by a UK-Royal Academy of Engineering/EPSRC Research Fellowship.

References

1. Beasley, J.E.: An algorithm for solving large capacitated warehouse location problems. European Journal of Operational Research, **33** (1988) 314–325
2. Benders, J. F.: Partitioning Procedures for Solving Mixed-Variables Programming Problems. Numerische Mathematik, **4** (1962) 238–252
3. Benoist, T., Gaudin, E., Rottembourg, B.: Constraint Programming Contribution to Benders Decomposition: A Case Study. Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, LNCS **2470** (2002) 603–617
4. Eremin, A., Wallace, M.: Hybrid Benders Decomposition Algorithms in Constraint Logic Programming. Principles and Practice of Constraint Programming, Proceedings of the 7th International Conference, CP 2001, Paphos, Cyprus, LNCS **2239** (2001) 1–15
5. Geoffrion, A. M.: Generalized Benders decomposition. Journal of Optimization Theory and Applications, **10** (1972) 237-260
6. Holmberg, K., Ronnqvist, M., Yuan, D.: An exact algorithm for the capacitated facility location problem with single sourcing. European Journal of Operational Research, **113** (1999) 544–559
7. Hooker, J. N., Ottosson, G.: Logic-based Benders decomposition. Mathematical Programming, **96** (2003) 33–60
8. ILOG Inc.: OPL Studio 3.7, Studio Users Manual, (2003)
9. Jain, V., Grossmann, I. E.: Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems. INFORMS Journal on Computing, **13** (2001) 258–276
10. Khumawala, B. M.: An Efficient Branch-Bound Algorithm for the Warehouse Location Problem. Management Science. **18** (1972) 718–731
11. Krarup, J., Pruzan, P. M.: The simple plant location problem: Survey and synthesis. European Journal of Operational Research. **12** (1983) 36–81
12. Manandhar, S., Tarim, S. A., Walsh. T.: Scenario-Based Stochastic Constraint Programming. Proceedings of IJCAI-2003, Acapulco, Mexico. (2003) 257–262
13. Mirchandani, P. B., Francis, R. L.: Discrete Location Theory. John Wiley and Sons, (1990)
14. Proll, L., Smith, B.M. Integer linear programming and constraint programming approaches to a template design problem. INFORMS Journal on Computing 10(3):265–275, 1998.
15. Tarim, S. A., Manandhar, S., Walsh. T.: Stochastic Constraint Programming. submitted for publication, (2004)
16. Thorsteinsson, E. S.: Branch-and-Check: A Hybrid Framework Integrating Mixed Integer Programming and Constraint Logic Programming. Principles and Practice of Constraint Programming, Proceedings of the 7th International Conference, CP 2001, LNCS **2239** (2001) 16–30
17. Xia, Q., Eremin, A., Wallace. M.: Problem Decomposition for Traffic Diversions. Proceedings of the 1st International Conference, Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004, LNCS **3011** (2004) 348–363

CP-nets and Nash equilibria

Krzysztof R. Apt[1,2,3], Francesca Rossi[4], and Kristen Brent Venable[4]

[1] School of Computing, National University of Singapore
[2] CWI, Amsterdam
[3] University of Amsterdam, the Netherlands
[4] Department of Pure and Applied Mathematics, University of Padova, Italy

Abstract. CP-nets are a natural way to express qualitative and conditional preferences. Here we relate them to a natural extension of the classical notion of a strategic game in which parametrized strict linear orderings are used instead of payoff functions. We show then that the optimal outcomes of a CP-net are exactly the Nash-equilibria of an appropriately defined strategic game in the above sense. This allows us to use the techniques of game theory to search for optimal outcomes of CP-nets and vice-versa, to use techniques developed for CP-nets to search for Nash equilibria of the considered games. We believe this is a first promising step towards a fruitful cross-fertilization between these two research areas, from artificial intelligence and microeconomic theory.

1 Introduction

CP-nets (Conditional Preference nets) are an elegant formalism for representing conditional and qualitative preferences, see [4, 6, 3]. They model such preferences under a *ceteris paribus* (that is, 'all else being equal') assumption. Preference elicitation in such a framework appears to be natural and intuitive.

Research on CP-nets focused on its modeling capabilities and algorithms for solving various natural problems related to their use. Also, computational complexity of these problems was extensively studied. An outcome of a CP-net is an assignment of values to its variables. One of the fundamental problems is that of finding an optimal outcome, i.e., the one that cannot be improved in presence of the adopted preference statements. This is in general a complex problem since it was found that finding optimal outcomes and testing for their existence is NP-hard in general. In contrast, for acyclic CP-nets this is an easy problem which can be solved by a linear time algorithm.

The aim of this paper to show the relationship between CP-nets and game theory, and how game-theoretic techniques developed for the analysis of strategic games can be fruitfully used to study CP-nets. To this end, we introduce a generalization of the customary strategic games (see, e.g., [9],) in which each player has to his disposal a strict preference relation on his set of strategies, parametrized by a joint strategy of his opponents. We call such games *strategic games with parametrized preferences*.

The cornerstone of our approach are two results closely relating CP-nets to such games. They show that the optimal outcomes of a CP-net are exactly the Nash-equilibria of an appropriately defined strategic game with parametrized preferences. This allows us to transfer techniques of game theory to CP-nets. These results are based on the observation that the ceteris-paribus principle, typical of CP-nets, implies that any optimal outcome is worsened if any worsening change to any variable is made. This is exactly the idea of a Nash equilibrium, and thus the results follow easily once this is clear.

Notice that these results do not hold if we consider other ways to express preferences, for example soft constraints [2] rather than CP-nets. Consider for instance fuzzy constraints [7], where the preference of a solution is the minimal preference over all the constraints, and solutions with higher values are better. In a fuzzy constraint problem, we can have a solution s_1 with preference $0.5 = min(0.5, 0.6, 0.7)$ (assuming three constraints) and another one, say s_2, with preference $0.5 = min(0.5, 0.9, 0.7)$. Then, s_1 could be optimal (if no solution has preference higher than 0.5), but is not a Nash equilibrium, since the preference of the second constraint is improved by passing from s_1 to s_2.

To find Nash equilibria in strategic games, many reduction techniques have been studied which reduce the game by eliminating some players' strategies, thus obtaining a smaller game. We introduce a counterpart of one of such game-theoretic technique that allows us to reduce a CP-net while maintaining its optimal outcomes. We also introduce a method of simplifying a CP-net by eliminating so-called redundant variables from the variables parent sets. Both techniques simplify the search for optimal outcomes of a CP-net.

In the other direction, we can use the techniques developed to reason about optimal outcomes of a CP-net in search for Nash equilibria of strategic games with parametrized preferences. These techniques, as recently shown in [5, 11], involve the use of the customary constraint solving techniques. In fact, it has been shown that the optimal outcomes of any CP-nets, even a cyclic one, can be found by just solving a set of hard constraints. Thus hard constraint solving is enough to find also Nash equilibria in strategic games. In particular, when the CP-net corresponding to a given game is acyclic, we know it has a unique optimal outcome that can be found in linear time. This allows us to find easily the unique Nash equlibrium of the given game.

The paper is organized as follows. Section 2 provides the basic definitions of CP-nets. Then, Section 3 introduces our generalized notion of games, Section 4 shows how to pass from a CP-net to a game, and Section 5 handles the opposite direction. Then, Section 6 introduces the concept of reduced CP-nets, and Sections 7 and 8 show how to exploit techniques developed for CP-nets in games and vice-versa. Finally, Section 9 summarizes the main contributions of the paper and gives some hints for future work.

2 CP-nets

CP-nets [4, 3] (for Conditional Preference nets) are a graphical model for compactly representing conditional and qualitative preference relations. They exploit conditional preferential independence by decomposing an agent's preferences via the **ceteris paribus** (cp) assumption. Informally, CP-nets are sets of ***ceteris paribus (cp)*** preference statements. For instance, the statement *"I prefer red wine to white wine if meat is served."* asserts that, given two meals that differ *only* in the kind of wine served *and* both containing meat, the meal with a red wine is preferable to the meal with a white wine. On the other hand, this statement does not order two meals with a different main course. Many users' preferences appear to be of this type.

CP-nets bear some similarity to Bayesian networks. Both utilize directed graphs where each node stands for a domain variable, and assume a set of *features* (variables) $F = \{X_1, \ldots, X_n\}$ with the corresponding finite domains $\mathcal{D}(X_1), \ldots, \mathcal{D}(X_n)$. For each feature X_i, a user specifies a (possibly empty) set of ***parent*** features $Pa(X_i)$ that can affect her preferences over the values of X_i. This defines a ***dependency graph*** in which each node X_i has $Pa(X_i)$ as its immediate predecessors.

Given this structural information, the user explicitly specifies her preference over the values of X_i for *each complete assignment* on $Pa(X_i)$. This preference is assumed to take the form of a linear ordering over $\mathcal{D}(X_i)$ [4, 3]. Each such specification is called below a ***preference statement*** for the variable X_i. These conditional preferences over the values of X_i are captured by a *conditional preference table* which is annotated with the node X_i in the CP-net. An ***outcome*** is an assignment of values to the variables with each value taken from the corresponding domain.

As an example, consider a CP-net whose features are A, B, C and D, with binary domains containing f and $\overline{f}$ if F is the name of the feature, and with the following preference statements:

$d : a \succ \overline{a},\ \overline{d} : a \succ \overline{a},$
$a : b \succ \overline{b},\ \overline{a} : \overline{b} \succ b,$
$b : c \succ \overline{c},\ \overline{b} : \overline{c} \succ c,$
$c : d \succ \overline{d},\ \overline{c} : \overline{d} \succ d.$

Here the preference statement $d : a \succ \overline{a}$ states that $A = a$ is preferred to $A = \overline{a}$, given that $D = d$. From the structure of these preference statements we see that $Pa(A) = \{D\}, Pa(B) = \{A\}, Pa(C) = \{B\}, Pa(D) = \{C\}$ so the dependency graph is cyclic.

An ***acyclic*** CP-net is one in which the dependency graph is acyclic. As an example, consider a CP-net whose features and domains are as above and with the following preference statements:

$a \succ \overline{a},$
$b \succ \overline{b},$
$(a \wedge b) \vee (\overline{a} \wedge \overline{b}) : c \succ \overline{c},\ (a \wedge \overline{b}) \vee (\overline{a} \wedge b) : \overline{c} \succ c,$
$c : d \succ \overline{d},\ \overline{c} : \overline{d} \succ d.$

Here, the preference statement $a \succ \overline{a}$ represents the unconditional preference for $A = a$ over $A = \overline{a}$. Also each preference statement for the variable C is a actually an abbreviated version of two preference statements. In this example we have $Pa(A) = \emptyset, Pa(B) = \emptyset, Pa(C) = \{A, B\}, Pa(D) = \{C\}$.

The semantics of CP-nets depends on the notion of a ***worsening flip***. A worsening flip is a transition between two outcomes that consists of a change in the value of a single variable to one which is less preferred in the unique preference statement for that variable. By analogy we define an ***improving flip***. For example, in the acyclic CP-net above, passing from $abcd$ to $ab\overline{c}d$ is a worsening flip since c is better than $\overline{c}$ given a and b. We say that an outcome α is ***better*** than the outcome β (or, equivalently, β is ***worse*** than α), written as $\alpha \succ \beta$, iff there is a chain of worsening flips from α to β. This definition induces a strict preorder over the outcomes. In the above acyclic CP-net the outcome $\overline{a}b\overline{c}\overline{d}$ is worse than $abcd$.

An ***optimal*** outcome is one for which no better outcome exists. In general, a CP-net does not need to have an optimal outcome. As an example consider two features A and B with the respective domains $\{a, \overline{a}\}$ and $\{b, \overline{b}\}$ and the following preference statements:

$a : b \succ \overline{b},\ \ \overline{a} : \overline{b} \succ b,$

$b : \overline{a} \succ a,\ \ \overline{b} : a \succ \overline{a}.$

It is easy to see that then

$ab \succ a\overline{b} \succ \overline{a}\overline{b} \succ \overline{a}b \succ ab.$

Finding optimal outcomes and testing for optimality is NP-hard. However, in acyclic CP-nets there is a unique optimal outcome and it can be found in linear time [4, 3]. We simply sweep through the CP-net, following the arrows in the dependency graph, assigning at each step the most preferred value in the preference relation. For instance, in the CP-net above, we would choose $A = a$ and $B = b$, then $C = c$ and then $D = d$. The optimal outcome is therefore $abcd$.

Hard constraints are enough to find optimal outcomes of a CP-net and to test whether a CP-net has an optimal outcome. In fact, given a CP-net one can define a set of hard constraints (called ***optimality constraints***) such that their solutions are the optimal outcomes of the CP-net [5, 11].

Indeed, take a CP-net N and consider a linear ordering $\succ$ over the elements of the domain of a variable X used in a preference statement for X. Let φ be the disjunction of the corresponding assignments used in the preference statements that use $\succ$. Then for each of such linear ordering $\succ$ the corresponding optimality constraint is $\varphi \rightarrow X = a_j$, where a_j is the undominated element of $\succ$. The optimality constraints $opt(N)$ corresponding to N consist of the entire set of such optimality constraints, each for one such linear ordering $\succ$.

For example, the preference statements $a \succ \overline{a}$ and $(a \wedge \overline{b}) \vee (\overline{a} \wedge b) : \overline{c} \succ c$ from the above CP-net map to the hard constraints $A = a$ and $(A = a \wedge B = \overline{b}) \vee (A = \overline{a} \wedge B = b) \rightarrow C = \overline{c}$, respectively.

It has been shown that an outcome is optimal in the strict preorder over the outcomes induced by a CP-net N iff it is a satisfying assignment for $opt(N)$.

A CP-net is ***eligible*** iff it has an optimal outcome. Even if the strict preorder induced by a CP-net has cycles, the CP-net may still be useful if it is eligible. All acyclic CP-nets are trivially eligible as they have a unique optimal outcome. We can thus test eligibility of any (even cyclic) CP-net by testing the consistency of the optimality constraints $opt(N)$. That is, a CP-net N is eligible iff $opt(N)$ is consistent.

3 Strategic games with parametrized preferences

In this section we introduce a generalization of the notion of a strategic game used in game theory, see, e.g., [9].

First we need the concept of a ***preference*** on a set A which in this paper denotes a strict linear ordering on A. If $\succ$ is a preference, we denote by $\succeq$ the corresponding ***weak preference*** defined by: $a \succeq b$ iff $a \succ b$ or $a = b$.

Given a sequence of non-empty sets $S_1, \ldots, S_n$ and $s \in S_1 \times \ldots \times S_n$ we denote the ith element of s by s_i and use the following standard notation of game theory, where $I := i_1, \ldots, i_k$ is a subsequence of $1, \ldots, n$:

- $s_{-i} := (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$,
- $s_I := (s_{i_1}, \ldots, s_{i_k})$,
- $(s'_i, s_{-i}) := (s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_n)$, where we assume that $s'_i \in S_i$,
- $S_{-i} := S_1 \times \ldots \times S_{i-1} \times S_{i+1} \times \ldots \times S_n$,
- $S_I := S_{i_1} \times \ldots \times S_{i_k}$.

In game theory it is customary to study strategic games in which the outcomes are numerical values provided by means of the payoff functions. A notable exception is [10] in which instead of payoff functions the linear quasi-orderings on the sets of joint strategies are used.

In our setup we adopt a different approach according to which each player has to his disposal a strict preference relation $\succ(s_{-i})$ on his set of strategies *parametrized* by a joint strategy s_{-i} of his opponents. So in our approach

- for each $i \in [1..n]$ player i has a finite, non-empty, set S_i of strategies available to him,
- for each $i \in [1..n]$ and $s_{-i} \in S_{-i}$ player i has a preference relation $\succ(s_{-i})$ on his set of strategies S_i.

In what follows such a ***strategic game with parametrized preferences*** (in short a ***game with parametrized preferences***, or just a ***game***) for n players is represented by a sequence

$$(S_1, \ldots, S_n, \succ(s_{-1}), \ldots, \succ(s_{-n})),$$

where each s_{-i} ranges over S_{-i}.

It is straightforward to transfer to the case of games with parametrized preferences the basic notions concerning strategic games. The following notions will be of importance for us (for the original definitions see, e.g., [10]), where G is a game with parametrized preferences specified as above.

- A strategy s_i is a ***best response*** for player i to a joint strategy s_{-i} of his opponents if $s_i \succeq(s_{-i})\ s'_i$, for all $s'_i \in S_i$.
- A strategy s_i is ***never a best response*** for player i if it is not a best response to any joint strategy s_{-i} of his opponents.
- A joint strategy s is a ***Nash equilibrium*** of G if each s_i is a best response to s_{-i}. Equivalently, s a Nash equilibrium if for all $i \in [1..n]$ and all $s'_i \in S_i$

$$s_i \succeq(s_{-i})\ s'_i.$$

- A strategy s'_i is ***strictly dominated*** by a strategy s_i if $s_i \succ(s_{-i})\ s'_i$, for all $s_{-i} \in S_{-i}$.

To clarify these definitions consider the classical Prisoner's dilemma strategic game represented by the following bimatrix representing the payoffs to both players:

	C_2	N_2
C_1	$3, 3$	$0, 4$
N_1	$4, 0$	$1, 1$

So each player i has two strategies, C_i (cooperate) and N_i (not cooperate), the payoff to player 1 for the joint strategy (C_1, N_2) is 0, etc. To represent this game as a game with parametrized preferences we simply stipulate that

$\succ(C_2) := N_1 \succ C_1,\ \succ(N_2) := N_1 \succ C_1,$

$\succ(C_1) := N_2 \succ C_2,\ \succ(N_1) := N_2 \succ C_2.$

These orderings reflect the fact that for each strategy of the opponent each player considers his 'not cooperate' strategy better than his 'cooperate' strategy. So for each player i his strategy C_i is strictly dominated by N_i, or (here) equivalently, his strategy C_i is never a best response. Further, (N_1, N_2) is a unique Nash equilibrium of this game with parametrized preferences.

Given a game with parametrized preferences

$$G := (S_1, \ldots, S_n, \succ(s_{-1}), \ldots, \succ(s_{-n})),$$

where each s_{-i} ranges over S_{-i}, and sets of strategies $S'_1, \ldots, S'_n$ such that $S'_i \subseteq S_i$ for $i \in [1..n]$, we say that

$$G' := (S'_1, \ldots, S'_n, \succ(s_{-1}), \ldots, \succ(s_{-n})),$$

where each s_{-i} now ranges over S'_{-i}, is a ***subgame*** of G, and identify in the context of G' each preference relation $\succ(s_{-i})$ with its restriction to S'_i.

We now introduce the following two notions of reduction between a game

$$G := (S_1, \ldots, S_n, \succ(s_{-1}), \ldots, \succ(s_{-n})),$$

where each s_{-i} ranges over S_{-i} and its subgame

$$G' := (S'_1, \ldots, S'_n, \succ(s_{-1}), \ldots, \succ(s_{-n})),$$

where each s_{-i} ranges over S'_{-i}:

- $G \to_{NBR} G'$
  when $G \neq G'$ and for all $i \in [1..n]$ each $s_i \in S_i \setminus S'_i$ is never a best response for player i in G,
- $G \to_S G'$
  when $G \neq G'$ and for all $i \in [1..n]$ each $s'_i \in S_i \setminus S'_i$ is strictly dominated in G by some $s_i \in S_i$.

In the literature it is customary to consider more specific reduction relations in which, respectively, *all* never best responses or *all* strictly dominated strategies are eliminated. The advantage of using the above versions is that we can prove the relevant property of both reductions by just one simple lemma, since by definition a strictly dominated strategy is never a best response and consequently $G \to_S G'$ implies $G \to_{NBR} G'$.

Lemma 1. *Suppose that $G \to_{NBR} G'$. Then s is a Nash equilibrium of G iff it is a Nash equilibrium of G'.*

Proof. ($\Rightarrow$) By definition each s_i is a best response to s_{-i} to G. So no s_i is eliminated in the reduction of G to G'.

($\Leftarrow$) Suppose s is not a Nash equilibrium of G. So some s_i is not a best response to s_{-i} in G. Let s'_i be a best response to s_{-i} in G. (s'_i exists since $\succ(s_{-i})$ is a linear ordering.)

So s'_i is not eliminated in the reduction of G to G' and s'_i is a best response to s_{-i} in G'. But this contradicts the fact that s is a Nash equilibrium of G'. □

Theorem 1. *Suppose that $G \rightarrow^*_{NBR} G'$, i.e., G' is obtained by an iterated elimination of never best responses from the game G.*

(i) Then s is a Nash equilibrium of G iff it is a Nash equilibrium of G'.
(ii) If each player in G' has just one strategy, then the resulting joint strategy is a unique Nash equilibrium of G.

Proof.
(i) By the repeated application of Lemma 1.

(ii) It suffices to note that $(s_1, \ldots s_n)$ is a unique Nash equilibrium of the game in which each player i has just one strategy, s_i. □

The above theorem allows us to reduce a game without affecting its (possibly empty) set of Nash equilibria or even, occasionally, to find its unique Nash equilibrium. In the latter case one says that the original game was ***solved*** by an iterated elimination of never best responses (or of strictly dominated strategies).

As an example let us return to the Prisoner's dilemma game with parametrized preferences defined above. In this game each strategy C_i is strictly dominated by N_i, so the game can be solved by either reducing it in two steps (by removing in each step one C_i strategy) or in one step (by removing both C_i strategies) to a game in which each player i has exactly one strategy, N_i.

Finally, let us mention that [8] and [12] proved that all iterated eliminations of strictly dominated strategies yield the same final outcome. An analogous result for the iterated elimination of never best responses was established in [1].

4 From CP-nets to strategic games

Consider now a CP-net with the set of variables $\{X_1, \ldots, X_n\}$ with the corresponding finite domains $\mathcal{D}(X_1), \ldots, \mathcal{D}(X_n)$. We write each preference statement for the variable X_i as $X_I = a_I : \succ_i$, where for the subsequence $I = i_1, \ldots, i_k$ of $1, \ldots, n$:

- $Pa(X_i) = \{X_{i_1}, \ldots, X_{i_k}\}$,
- $X_I = a_I$ is an abbreviation for $X_{i_1} = a_{i_1} \wedge \ldots \wedge X_{i_k} = a_{i_k}$,
- $\succ_i$ is a preference over $\mathcal{D}(X_i)$.

We also abbreviate $\mathcal{D}(X_{i_1}) \times \ldots \times \mathcal{D}(X_{i_k})$ to $\mathcal{D}(X_I)$.

By definition, the preference statements for a variable X_i are exactly all statements of the form $X_I = a_I : \succ(a_I)$, where a_I ranges over $\mathcal{D}(X_I)$ and $\succ(a_I)$ is a preference on $\mathcal{D}(X_i)$ that depends on a_I.

We now associate with each CP-net N a game $\mathcal{G}(N)$ with parametrized preferences as follows:

- each variable X_i corresponds to a player i,
- the strategies of player i are the elements of the domain $\mathcal{D}(X_i)$ of X_i.

To define the parametrized preferences, consider a player i. Suppose $Pa(X_i) = \{X_{i_1}, \ldots, X_{i_k}\}$ and let $I := i_1, \ldots, i_k$. So I is a subsequence of $1, \ldots, i-1, i+1, \ldots, n$. Given a joint strategy a_{-i} of the opponents of player i, we associate with it the preference relation $\succ(a_I)$ on $\mathcal{D}(X_i)$ where $X_I = a_I : \succ(a_I)$ is the unique preference statement for X_i determined by a_I.

In words, the preference of a player i over his strategies, given the strategies chosen by its opponents, say a_{-i}, coincides with the preference given by the CP-net over the domain of X_i given the assignment to his parents a_I which must coincide with the projection of a_{-i} over I.

This completes the definition of $\mathcal{G}(N)$.

As an example consider the first CP-net of Section 2. The corresponding game has four players A, B, C, D, each with two strategies indicated with f, $\bar{f}$ for player F. The preference of each player on his strategies will depend only on the strategies chosen by the players which correspond to his parents in the CP-net. Consider for example player B. His preference over his strategies b and $\bar{b}$, given the joint strategy of his opponents $s_{-B} = dac$, is $b \succ \bar{b}$. Notice that, for example, the same ordering holds for the opponents joint strategy $s_{-B} = \bar{d}a\bar{c}$, since the strategy chosen by the only player corresponding to his parent, A, has not changed.

We have then the following result.

Theorem 2. *An outcome of a CP-net N is optimal iff it is a Nash equilibrium of the game $\mathcal{G}(N)$.*

Proof. ($\Rightarrow$) Take an optimal outcome o of N. Consider a player i in the game $\mathcal{G}(N)$ and the corresponding variable X_i of N. Suppose $Pa(X_i) = \{X_{i_1}, \ldots, X_{i_k}\}$. Let $I := i_1, \ldots, i_k$, and let $X_I = o_I : \succ(o_I)$ be the corresponding preference statement for X_i. By definition there is no improving flip from o to another outcome, so o_i is the maximal element in the ordering $\succ(o_I)$.

By the construction of the game $\mathcal{G}(N)$, each outcome in N is a joint strategy in $\mathcal{G}(N)$. Also, two outcomes are one flip away iff the corresponding joint strategies differ only in a strategy of one player. Given the joint strategy o considered above, we thus have that, if we modify the strategy of player i, while leaving the strategies of the other players unchanged, this change is worsening in $\succ(o_{-i})$, since $\succ(o_{-i})$ coincides with $\succ(o_I)$. So by definition o is a Nash equilibrium of $\mathcal{G}(N)$.

($\Leftarrow$) Take a Nash equilibrium s of the game $\mathcal{G}(N)$. Consider a variable X_i of N. Suppose $Pa(X_i) = \{X_{i_1}, \ldots, X_{i_k}\}$. Let $I := i_1, \ldots, i_k$, and let $X_I = s_I : \succ(s_I)$ be the corresponding preference statement for X_i.

By definition for every strategy $s_i' \neq s_i$ of player i, we have $s_i \succ(s_{-i})\ s_i'$, so $s_i \succ(s_I)\ s_i'$ since $\succ(s_{-i})$ coincides with $\succ(s_I)$. So by definition s is an optimal outcome for N. $\square$

5 From strategic games to CP-nets

We now associate with each game G with parametrized preferences a CP-net $\mathcal{N}(G)$ as follows:

- each player i corresponds to a variable X_i,
- the domain $\mathcal{D}(X_i)$ of the variable X_i consists of the set of strategies of player i,
- we stipulate that $Pa(X_i) = \{X_1, X_{i-1}, \ldots, X_{i+1}, \ldots, X_n\}$, where n is the number of players in G.

Next, for each joint strategy s_{-i} of the opponents of player i we take the preference statement $X_{-i} = s_{-i} : \succ(s_{-i})$, where $\succ(s_{-i})$ is the preference relation on the set of strategies of player i associated with s_{-i}.

This completes the definition of $\mathcal{N}(G)$. As an example of this construction let us return to the Prisoner's dilemma game with parametrized preferences from Section 3. In the corresponding CP-net we have then two variables X_1 and X_2 corresponding to players 1 and 2, with the respective domains $\{C_1, N_1\}$ and $\{C_2, N_2\}$. To explain how each parametrized preference translates to a preference statement take for example $\succ(C_2) := N_1 \succ C_1$. It translates to $X_2 = C_2 : N_1 \succ C_1$.

We have now the following counterpart of Theorem 2.

Theorem 3. *A joint strategy is a Nash equilibrium of the game G iff it is an optimal outcome of the CP-net $\mathcal{N}(G)$.*

Proof. ($\Rightarrow$) Assume G has a Nash equilibrium s. Thus, for every player i, joint strategy s_{-i}, and strategy $s'_i \neq s_i$ for player i, we have $s_i \succ(s_{-i})\ s'_i$. This means that, if we only change the strategy of any player i, this change is worsening for that player. In the CP-net $\mathcal{N}(G)$, s is an outcome, and the ordering in the conditional preference table of variable X_i coincides with $' \succ(s_{-i})$. Thus all the flips from s are worsening. Thus s is optimal for $\mathcal{N}(G)$.

($\Leftarrow$) Assume $\mathcal{N}(G)$ is eligible, and take an optimal outcome o of $\mathcal{N}(G)$. By definition of optimal outcome, for every other outcome o', $o' \not\succ o$, which means that there is no sequence of improving flips from o to any other outcome. Thus there is no improving flip from o to any other outcome. Therefore every flip modifying variable X_i in o is worsening in the preference statement for the variable X_i.

Given the construction from game G to CP-net $\mathcal{N}(G)$, an outcome in $\mathcal{N}(G)$ is a joint strategy in G. Also, two outcomes one flip away in $\mathcal{N}(G)$ are two joint strategies of G which differ only for the strategy of one player. Given the joint strategy o, we thus have that, if we modify the strategy of player i, while leaving the strategies of the other players unchanged, this change is worsening in $\succ_i(o_{-i})$, since the preference of variable X_i given o_{-i} coincides with $\succ_i(o_{-i})$.

$\square$

6 Reduced CP-nets

The disadvantage of the above construction of the CP-net $\mathcal{N}(G)$ from a game G is that it always produces a CP-net in which all sets of parent features are of size $n - 1$ where

n is the number of features of the CP-net. This can be rectified by reducing each set of parent features to a minimal one as follows.

Given a CP-net N, consider a variable X_i with the parents $Pa(X_i)$, and take a variable $Y \in Pa(X_i)$. Suppose that for all assignments a to $Pa(X) - \{Y\}$ and any two values $y_1, y_2 \in \mathcal{D}(Y)$, the orderings $\succ(a, y_1)$ and $\succ(a, y_2)$ on $\mathcal{D}(X_i)$ coincide.

We say then that Y is *redundant* in the set of parents of X_i. It is easy to see that by removing all redundant variables from the set of parents of X_i and by modifying the corresponding preference statements for X_i accordingly, the strict preorder $\succ$ over the outcomes of the CP-nets is not changed.

Given a CP-net, if for all its variable X_i the set $Pa(X_i)$ does not contain any redundant variable, we say that the CP-net is ***reduced***.

By iterating the above construction every CP-net can be transformed to a reduced CP-net. As an example consider a CP-net with three features, X, Y and Z, with the respective domains $\{a_1, a_2\}, \{b_1, b_2\}$ and $\{c_1, c_2\}$. Suppose now that $Pa(X) = Pa(Y) = \emptyset, Pa(Z) = \{X, Y\}$ and that

$\succ(a_1, b_1) = \succ(a_2, b_1),\ \succ(a_1, b_2) = \succ(a_2, b_2),$

$\succ(a_1, b_1) = \succ(a_1, b_2),\ \succ(a_2, b_1) = \succ(a_2, b_2).$

Then both X and Y are redundant, so we can reduce the CP-net by reducing $Pa(Z)$ to $\emptyset$. Z becomes an independent variable in the reduced CP-net with an ordering over its domain which coincides with the unique one given in the original CP-net in terms of the assignments to its parents.

In what follows for a CP-net N we denote by $r(N)$ the corresponding reduced CP-net. The following result summarizes the relevant properties of $r(N)$ and relates it to the constructions of $\mathcal{G}(N)$ and $\mathcal{N}(G)$.

Theorem 4.
- *(i) Each CP-net N and its reduced form $N' = r(N)$ have the same ordering $\succ$ over the outcomes.*
- *(ii) For each CP-net N and its reduced form $N' = r(N)$ we have $\mathcal{G}(N) = \mathcal{G}(N')$.*
- *(iii) Each reduced CP-net N is a reduced CP-net corresponding to the game $\mathcal{G}(N)$. Formally: $N = r(\mathcal{N}(\mathcal{G}(N)))$.*

Proof.
(i) Consider N and $r(N)$, an outcome o and an improving flip in N from o to o' which modifies the value of variable X_i. Then this change is improving in the conditional preference table of X_i in N. Let us now consider the conditional preference table of X_i in $r(N)$. In this table there could be a subsequence of parents but with the same corresponding preference orderings. Thus the change is improving in this conditional preference table as well. Thus any chain of improving flips in N remain a chain of improving flips also in $r(N)$, and therefore the orderings over the outcomes of N and $r(N)$ are the same.

(ii) It follows directly from the construction of the game corresponding to a CP net, since the preference of player i over its strategies depends on the strategies of all the other opponents, even if variable X_i has just a few parents in N.

(iii) Given a reduced CP-net N, consider the CP-net $\mathcal{N}(\mathcal{G}(N))$. For each variable X_i, $Pa(X_i)$ in N is a subset of $Pa(X_i)$ in $\mathcal{N}(\mathcal{G}(N))$, which is the set of all variables except

X_i. However, by the construction of the game corresponding to a CP-net and of the CP-net corresponding to a game, in each conditional preference table, if the assignments to the common parents are the same, the preference orderings over X_i are the same.

Let us now reduce $\mathcal{N}(\mathcal{G}(N))$ to obtain $N' = r(\mathcal{N}(\mathcal{G}(N)))$. Then $Pa(X_i)$ in N' coincides with $Pa(X_i)$ in N. In fact, assume there is a parent of X_i in N which is not in N'. Since N is reduced, such a parent cannot be redundant in N. Thus the reduction, when applied to $\mathcal{N}(\mathcal{G}(N))$, cannot remove it since the orderings in the conditional preference tables of N and $\mathcal{N}(\mathcal{G}(N))$ are the same. On the other hand, assume there is a parent of X_i in N' which is not in N. Since N' is reduced, such a parent cannot be redundant in N'. Thus it is not redundant in $\mathcal{N}(\mathcal{G}(N))$ as well. By construction of $\mathcal{N}(\mathcal{G}(N))$, it cannot be redundant in N neither. $\square$

Part (i) states that the reduction procedure preserves the ordering over the outcomes. Part (ii) states that the construction of a game corresponding to a CP-net does not depend on the redundancy of the given CP-net. Finally, part (iii) states that the reduced CP-net N can be obtained 'back' from the game $\mathcal{G}(N)$.

7 Game theoretical techniques in CP-nets

Given the correspondence between CP-nets and games and its properties presented in the previous sections, we can now use them to transfer standard techniques of game theory, used to find Nash equilibria, to CP-nets to find their optimal outcomes, and vice-versa.

To be more specific, given a CP-net N, consider the game $\mathcal{G}(N)$. Let us now modify $\mathcal{G}(N)$ by an iterated elimination of never best responses, obtaining a game G'. By Theorems 1 and 2, an outcome of N is optimal iff it is a Nash equilibrium of G'. Now modify N by

- reducing the variable domains to the corresponding sets of strategies in G',
- removing all preference statements that refer to a removed element,

and call the resulting CP-net N'.

By Theorem 1, the Nash equilibria of G and G' coincide. Also, by Theorem 3, a joint strategy is a Nash equilibrium of G iff it is an optimal outcome of N, and a joint strategy is a Nash equilibrium of G' iff it is an optimal outcome of N'. Thus N and N' have the same set of optimal outcomes.

It is useful to note that the elimination of never best responses, and consequently also the elimination of strictly dominated elements, can be carried out directly on a CP-net by introducing the following notions. Consider a CP-net N.

- We say that an element d_i from the domain $\mathcal{D}(X_i)$ of the variable X_i is a ***best response*** to a preference statement

  $$X_I = a_I : \succ_i$$

  for X_i if $d_i \succeq_i d'_i$ for all $d'_i \in \mathcal{D}(X_i)$.

– We say that an element d_i from the domain of the variable X_i is a ***never a best response*** if it is not a best response to any preference statement for X_i.
– Given two elements d_i, d'_i from the domain $\mathcal{D}(X_i)$ of the variable X_i we say that d'_i is ***strictly dominated*** by d_i if for all preference statements $X_I = a_I : \succ_i$ for X_i we have

$$d_i \succ_i d'_i.$$

By a ***subnet*** of a CP-net N we mean a CP-net obtained from N by removing some elements from some variable domains followed by the removal of all preference statements that refer to a removed element.

Then we introduce the following relation between a CP-net N and its subnet N':

$$N \rightarrow_{NBR} N'$$

when $N \neq N'$ and for each variable X_i each removed element from the domain of X_i is never a best response in N, and introduce an analogous relation $N \rightarrow_S N'$ for the case of strictly dominated elements.

By the same argument as in the case of Theorem 1 we get the following result. Part (iii) can be established by repeating the argument of [1].

Theorem 5. *Suppose that $N \rightarrow^*_{NBR} N'$, i.e., the CP-net N' is obtained by an iterated elimination of never best responses from the CP-net N.*

(i) Then s is an optimal outcome of N iff it is an optimal outcome of N'.
(ii) If each variable in N' has a singleton domain, then the resulting outcome is a unique optimal outcome of N.
(iii) All iterated eliminations of never best responses from the CP-net N yield the same final outcome.

To illustrate the use of this theorem reconsider the first CP-net from Section 2, i.e., the one with the preference statements
$d : a \succ \overline{a},\ \overline{d} : a \succ \overline{a},$
$a : b \succ \overline{b},\ \overline{a} : \overline{b} \succ b,$
$b : c \succ \overline{c},\ \overline{b} : \overline{c} \succ c,$
$c : d \succ \overline{d},\ \overline{c} : \overline{d} \succ d.$
Denote it by N.

We can reason about it using the iterated elimination of strictly dominated strategies (which coincides here with the iterated elimination of never best responses, since each domain has exactly two elements).

We have the following chain of reductions:

$$N \rightarrow_S N_1 \rightarrow_S N_2 \rightarrow_S N_3 \rightarrow_S N_4,$$

where

– N_1 results from N by removing $\overline{a}$ (from the domain of A) and the preference statements $d : a \succ \overline{a},\ \overline{d} : a \succ \overline{a},\ \overline{a} : \overline{b} \succ b,$

– N_2 results from N_1 by removing $\overline{b}$ and the preference statements $a : b \succ \overline{b}$, $\overline{b} : \overline{c} \succ c$,
– N_3 results from N_2 by removing $\overline{c}$ and the preference statements $b : c \succ \overline{c}$ $\overline{c} : \overline{d} \succ d$,
– N_4 results from N_3 by removing $\overline{d}$ from the domain of D and the preference statement $c : d \succ \overline{d}$.

Indeed, in each step the removed element is strictly dominated in the considered CP-net. So using the iterated elimination of strictly dominated elements we reduced the original CP-net to one in which each variable has a singleton domain and consequently found a unique optimal outcome of the original CP-net N.

Finally, the following result shows that the introduced reduction relation on CP-nets is complete for acyclic CP-nets.

Theorem 6. *For each acyclic CP-net N a unique subnet N' with the singleton domains exists such that $N \rightarrow^*_{NBR} N'$.*

Proof. First note that if N is an acyclic CP-net with some non-singleton domain, then $N \rightarrow_{NBR} N'$ for some subnet N' of N. Indeed, suppose N is such a CP-net. Then a variable X exists with a non-singleton domain with no parent variable that has a non-singleton domain. So there exists in N exactly one preference statement for X, say $X_I = a_I : \succ_i$, where X_I is the sequence of parent variables of X. Reduce the domain of X to the maximal element in $\succ_i$. Then for the resulting subnet N' we have $N \rightarrow_{NBR} N'$.

Uniqueness of the outcome is a consequence of Theorem 5(iii). □

8 CP-net techniques in strategic games

The established relationship between CP-nets and strategic games with prioritized preferences allows us also to exploit the techniques developed for the CP-nets when reasoning about such games. In particular, to find the Nash equilibria of such a game we can proceed as follows. Take a game G. Apply to it an iterated elimination of never best responses. This yields a subgame G'. Now consider the corresponding CP-net $\mathcal{N}(G')$. Reduce it by eliminating redundant parents as described in Section 6, obtaining a CP-net $N' = r(\mathcal{N}(G))$. Next, apply to it an iterated elimination of never best responses. This yields a CP-net N''. By the theorems established in this paper the Nash equilibria of G coincide with the optimal outcomes of N''.

If each variable in N'' has a singleton domain, then we found a unique Nash equilibrium of G. Otherwise we can construct the optimality constraints $opt(N'')$ discussed in Section 2 and use the fact that $opt(N'')$ is consistent iff an optimal outcome of N'' exists. So we can now use hard constraint solving techniques to search for Nash equilibria of the original game G, by focusing on solving the constraints in $opt(N'')$.

As an example consider the following strategic game G, in which there are two players, Player 1 with the strategies *Top* and *Bottom* and Player 2 with the strategies *Left* and *Right*.

	L	R
T	$2,2$	$1,1$
B	$1,1$	$2,2$

Note that in this game each strategy is a best response, so it cannot be reduced using the elimination of never best responses.

The corresponding CP-net, $\mathcal{N}(G)$, has two variables, each corresponding to a player: X_1 and X_2, with the domains $\mathcal{D}(X_1) = \{T, B\}$ and $\mathcal{D}(X_2) = \{L, R\}$. Moreover $Pa(X_1) = \{X_2\}$ and $Pa(X_2) = \{X_1\}$. The preference statements are the following:

$X_1 = T : L \succ R$,
$X_1 = B : R \succ L$,
$X_2 = L : T \succ B$,
$X_2 = R : B \succ T$.

Notice that $\mathcal{N}(G)$ is already reduced. In fact, X_1 is not redundant as a parent of X_2 and vice-versa.

The optimality constraints corresponding to this CP-net are:

$X_1 = T \rightarrow X_2 = L,\ X_1 = B \rightarrow X_2 = R,$
$X_2 = L \rightarrow X_1 = T,\ X_2 = R \rightarrow X_1 = B.$

These constraints have two solutions: $(X_1 = T, X_2 = L)$ and $(X_1 = B, X_2 = R)$, which are also the only two Nash equilibria of the initial game.

This approach can be used also to discover quickly that a CP-net has a unique Nash equilibrium, and to find it. Given a game G, we construct the reduced CP-net $N' = r(\mathcal{N}(G))$. If this CP-net is acyclic, we know that it has a unique optimal outcome which can be found in linear time. By the Theorems of the previous sections we also know that the Nash equilibria of game G coincide with the optimal outcome of this CP-net. This means that G has a unique Nash equilibrium that can be found in linear time by the usual CP-net techniques applied to N'.

Games G such that the CP-net $N' = r(\mathcal{N}(G))$ is acyclic are not uncommon. In fact, they naturally represent multi-agent scenarios where agents (that is, players of the game) can be partitioned into levels $1, 2, \ldots, n$, such that agents at level i can express their preferences (that is, payoff function) without looking at what players at higher levels do. Informally, agents at level i are more important than agents at level j is $j > i$. In particular, agents at level 1 can decide their preferences without looking at the behavior of any other agent.

9 Conclusions and future work

We showed that optimal outcomes in CP-nets are Nash equilibria in strategic games, and we exploited this to inherit useful techniques from strategic games to CP-nets and vice-versa.

In this paper we assume that payoff functions give a linear order over the strategies of a player. It could be useful to see whether our results can be generalized to games in which players' strategies can be incomparable or indifferent to each other, thus using partial orderings with ties. We are currently studying this scenario.

This paper is just a first step towards what we think is a fruitful cross-fertilization between preferences, constraint solving, and game theory. CP-nets appear to be very amenable for being studied using the existing game theoretical techniques, and also to provide strategic games with new and hopefully more efficient approaches to find Nash equilibria or to solve other game-theoretical tasks. In addition, CP-nets are closely related to constraint solving and also to other preference formalisms like soft constraints, so we envision future possibilities for relating also these other research areas to game theory.

References

1. K. R. Apt. Rationalizability and order independence. In *Proc. 10th Conference on Theoretical Aspects of Reasoning about Knowledge (TARK)*. National University of Singapore, 2005. To appear.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, mar 1997.
3. C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res. (JAIR)*, 21:135–191, 2004.
4. C. Boutilier, R. I. Brafman, H. H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *UAI '99*, pages 71–80. Morgan Kaufmann, 1999.
5. R. Brafman and Y. Dimopoulos. Extended semantics and optimization algorithms for cp-networks. *Computational Intelligence*, 20, 2:218–245, 2004.
6. C. Domshlak and R. I. Brafman. CP-nets: Reasoning and consistency testing. In *KR-02*, pages 121–132. Morgan Kaufmann, 2002.
7. H. Fargier D. Dubois and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *IEEE International Conference on Fuzzy Systems*, 1993.
8. I. Gilboa, E. Kalai, and E. Zemel. On the order of eliminating dominated strategies. *Operation Research Letters*, 9:85–89, 1990.
9. R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard Univ Press, Cambridge, Massachusetts, 1991.
10. M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, Cambridge, Massachusetts, 1994.
11. S. Prestwich, F. Rossi, K. B. Venable, and T. Walsh. Constraint-based preferential optimization. In *Proc. of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*. Morgan Kaufmann, 2005.
12. M. Stegeman. Deleting strictly eliminating dominated strategies. Working Paper 1990/6, Department of Economics, University of North Carolina, 1990.

Disjunctive Temporal Problems with Uncertainty

Kristen Brent Venable[1] and Neil Yorke-Smith[2]

[1] University of Padova, Italy. kvenable@math.unipd.it
[2] Artificial Intelligence Center, SRI International, USA. nysmith@ai.sri.com

Abstract Driven by planning and scheduling problems which feature both disjunctive constraints and contingent events, we define the Disjunctive Temporal Problem with Uncertainty (DTPU). The DTPU is an extension of the Disjunctive Temporal Problem that accounts for contingent events not under the control of the executing agent. Generalizing existing work on Simple Temporal Problems with Uncertainty, we divide the time-points into controllable and uncontrollable classes, and propose varying notions of controllability to replace the notion of consistency. Algorithms are introduced to determine which levels of controllability hold for a DTPU, and initial complexity results are given.

1 Introduction

For modeling and solving real-world planning and scheduling problems, quantitative temporal constraint networks in the form of the Simple Temporal Problem (STP) [2] are widely known. The STP has been independently extended in two orthogonal directions.

First, since the STP formalism is unable to express concepts such as "event A occurs before or after event B", the Disjunctive Temporal Problem (DTP) [11] admits more than one disjunct in a temporal constraint. The increased expressiveness comes with increased complexity: the worst-case time complexity of solving a DTP is **NP**-hard whereas the STP is polynomial. In practice, effi cient solving techniques have been developed [13,1].

Second, since the STP formalism assumes all time-points are under the complete control of the execution agent, the *Simple Temporal Problem with Uncertainty* (STPU) [16,3] divides the time-points into two classes, *executable* and *contingent*. Contingent time-points are controlled by exogenous factors, often referred to as 'Nature', and consistency of an STP is replaced by varying notions of *controllability*.

Both disjunctive constraints and contingent events are intrinsic features of many real-world planning and scheduling problems. Consider, for instance, planning the schedule of an elderly patient [7]. On the one hand, her schedule includes events, such as exercising and a relative's visit, that are not totally ordered by the problem requirements: either event may occur before the other. On the other hand, while the patient has some idea when the relative will arrive, the precise time is not under her control.

Neither a DTP or an STPU is adequate on its own to represent such temporal problems and reason upon them. Driven by planning and scheduling problems such as these, in this paper we propose the *Disjunctive Temporal Problem with Uncertainty* (DTPU). The DTPU extends the Disjunctive Temporal Problem to accounts for contingent events, following the concepts pioneered in the STPU. We propose varying notions of controllability to replace the notion of consistency of a DTP, and introduce algorithms to determine which levels of controllability hold.

In the next section we recall the temporal constraint frameworks on which the DTPU builds, providing the necessary background for the definition of the DTPU itself (Section 3). We then introduce different levels of controllability for DTPUs (Section 4), and present algorithms to check which of them hold, together with an initial complexity analysis (Section 5). A discussion of related work concludes (Section 6).

2 Background

Simple Temporal Problem In a *Temporal Constraint Satisfaction Problem* (TCSP) [2], variables X_i denote time-points and constraints represent the possible temporal relations between them. Events are represented by a variable each for the start and end time-points. The constraints of a TCSP are quantitative, describing restrictions either on durations of events or on distances (interleaving times) between events, in terms of intervals over the timeline.[1] Thus TCSP constraints are unary, constraining the domain of a variable (which may be continuous or discrete), or binary. A binary TCSP constraint has the form:

$$X_j - X_i \in \{[a_{ij1}, b_{ij1}], \ldots, [a_{ij\ell}, b_{ij\ell}]\} \tag{1}$$

In general such problems are **NP**-complete. However, if each temporal constraint has just one interval — hence the constraints have form $X_j - X_i \in [a_{ij}, b_{ij}]$, or equivalently $a_{ij} \leq X_j - X_i \leq b_{ij}$ — then we have a *Simple Temporal Problem* (STP). A distinguished time-point, denoted *TR*, marks the start of time. Unary domain constraints thus can be modelled as binary relations to *TR*.

To conveniently distinguish variables, for convenience we will often write an STP constraint as $Y - X \in [a, b]$ when the identity of the variables and constants is clear.

Solving an STP equates to deciding consistency and deriving the minimal network of time-point domains. An STP (and more generally a TCSP) is consistent if it has a *solution*: an assignment to all time-points such that all constraints are satisfied. The *minimal network* is the tightest representation of the constraints that includes all solutions to the problem. Solving an STP can be achieved by enforcing path consistency, for which an All-Pairs Shortest Path algorithm such as Floyd-Warshall is sufficient. The computational complexity of solving an STP is thus cubic in the number of time-points, i.e. $\mathcal{O}(n^3)$ for n time-points [2].

Disjunctive Temporal Problem The *Disjunctive Temporal Problem* (DTP) [11] generalizes the TCSP by allowing non-binary temporal relations. This allows us to express constraints such as: event A (exercising) occurs before or after event B (relative's visit), which cannot be expressed with an STP or TCSP. DTP constraints are disjuncts:

$$X_{j1} - X_{i1} \in [a_{i1j1}, b_{i1j1}] \vee X_{j2} - X_{i2} \in [a_{i2j2}, b_{i2j2}] \vee \cdots \vee X_{j\ell} - X_{i\ell} \in [a_{i\ell j\ell}, b_{i\ell j\ell}] \tag{2}$$

Note that a variable may appear in more than one disjunct. A TCSP is an instance of a DTP with $X_{j1} = X_{j2} = \cdots = X_{j\ell}$ and $X_{i1} = X_{i2} = \cdots = X_{i\ell}$.

While the worst-case complexity of solving a DTP is **NP**-hard [11], in practice efficient solving techniques have been developed (e.g. [13,1]), and tractability results

[1] For simplicity, we assume closed intervals; however the same applies to semi-open intervals.

are known for some classes of DTPs [9]. The use of DTPs in decision-theoretic planning is discussed by [10].

A simple way to solve a DTP is to consider the *component* STPs obtained by selecting one disjunct from each constraint. A DTP is consistent iff it contains a consistent component STP. Search through the space of component STPs is the heart of most DTP solvers [11,4,13]. The complexity of a straightforward search through this space depends on the number of component STPs, which depends on the number of DTP constraints and the number of disjuncts in each. In the worst case, for m constraints with a maximum of k disjuncts each, there are $\mathcal{O}(k^m)$ component STPs. A straightforward search therefore has complexity $\mathcal{O}(n^3 k^m)$ if there is no consistent component STP [6].

Simple Temporal Problem with Uncertainty The STP and DTP formalisms assume all time-points are under the complete control of the execution agent. Recognizing that this assumption is often not valid, the *Simple Temporal Problem with Uncertainty* (STPU) [16] divides the time-points into two classes, *executable* and *contingent*. In contrast to executable time-points, which are controlled by the execution agent (e.g. the patient in our earlier example), contingent time-points are controlled by exogenous factors, often referred to as 'Nature'; they are observed but cannot be controlled by the agent. The only information known prior to observation is that Nature will respect the interval on the duration. Durations of contingent links are assumed independent.

Controllability of an STPU is the analogue of consistency of an STP. Controllable implies the agent has a means to execute the time-points under its control, subject to all constraints. Three notions have been proposed in [16]. Informally, Strong Controllability (SC) guarantees a single solution to the STPU, consistent whatever the *realisation* of the contingent time-points. Weak Controllability (WC), on the other hand, guarantees only the existence of some solution for each realisation. The two notions correspond to conformant and conditional planning respectively. Dynamic Controllability (DC) guarantees the existence of an online policy that can construct a solution step-by-step as contingent time-points are observed. It is assumed that a time-point is observed immediately after it occurs. The three notions are ordered by their strength: Strong $\implies$ Dynamic $\implies$ Weak. The time complexity of determining strong and weak controllability is in classes **P** and co-**NP** respectively [16]; determining dynamic controllability is **P** [3].[2]

3 Extending the DTP with Contingent Events

Following the STPU formalism [16], we divide the time-point variables into two classes: the controllable *decision variables* V_d and the uncontrollable *parameters* V_u. In an STPU, this induces a partition of the temporal relation constraints C into the two types described above. A constraint $Y - X \in [a, b]$, where $Y \in V_u$, is a *contingent* link. Intuitively, contingent links model tasks with uncertain durations. If a constraint is not contingent, it is an *executable* link; thus all other constraints (i.e. where $Y \in V_d$) are executable links.

In a DTPU, each constraint is a disjunction of links of one or both of these types:

[2] In fact, the algorithm of [3] is pseudo-polynomial in the size of the STPU representation.

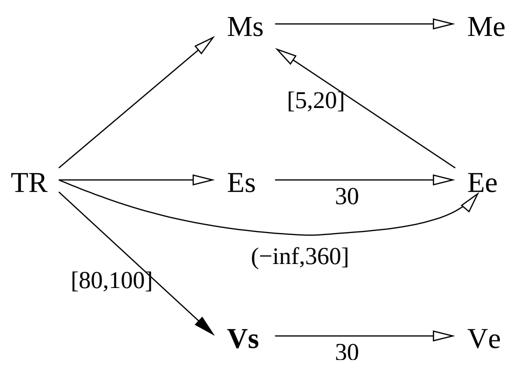


Figure1. Part of DTPU representing an elderly patient's schedule. The three tasks are taking medicine (M), exercising (E), and a relative's visit (V). The start time of the relative's visit, V_s, is a uncontrollable parameter. Not shown with arrows is the disjunctive constraint (3), which has two STP disjuncts: between the exercise end and visit start, and the visit end and exercise start.

Definition 1 (DTPU). *A* Disjunctive Temporal Problem with Uncertainty *is a tuple* $\langle V_d, V_u, R_d, R_u, C\rangle$*, where* V_d*,* V_u *are as above,* R_d *is the set of all executable links, and* R_u *is the set of all contingent links, and* C *is a finite set of disjunctive temporal constraints.* □

Thus in a DTPU, three types of constraints arise according to the contingent status of the included disjuncts. First, only R_d disjuncts; analogously with STPUs, the constraint is said to be *executable*. Second, only R_u disjuncts; analogously with STPUs, the constraint is said to be *contingent*. Third, both R_d and R_u disjuncts. Here, a situation which has no analogue in an STPU, the constraint can be satisfi ed by the satisfaction of either some executable link (a possibility under the agent's control), or of some contingent link (which is not); we say that it is *semi-executable*.

Example 1. Consider further the example of an elderly patient's schedule, from the AUTOMINDER domain [7]. The time-points in a DTP model of the schedule include E_s and E_e, and V_s and V_e, which describe the start and end of exercising and a visit respectively. A fragment of the schedule is shown in Figure 1. *TR* denotes the temporal reference time-point, which in this problem is set at 3 p.m.

The problem and its DTP formulation are fully described in [7,6]. The constraints describe the durations and temporal ordering between the activities. One of them is:

$$V_s - E_e \in [5, \infty) \vee E_s - V_e \in [0, \infty) \tag{3}$$

stating that "Exercise must fi nish at least 5 minutes before the visitor arrives *or* start anytime after the visitor leaves."

While the visitor has a scheduled arrival time of 4:30 p.m., his precise arrival is not under the control of the patient; thus V_s is a parameter. Its domain represents the patent's expectations of the occurrence of the contingent event. This means that the fi rst disjunct of (3) is a contingent link. The second disjunct is an executable link, and so constraint (3) is semi-executable. □

A constraint is satisfied if at least one of its disjuncts is satisfied. A solution to a DTPU is a complete assignment to all variables $V = V_d \cup V_u$ that satisfies all the constraints in C. A solution s can be partitioned into $s_d \cup s_u$, where s_d and s_u are the set of temporal values assigned to elements of V_d and V_u respectively. s_d is called a *decision*, and s_u is a *realisation* (or possible world, or scenario). We say that the DTP obtained when the parameters of a DTPU take values under a realisation s_u is the *realised DTP* corresponding to that realisation.

We say a *component* STPU of a DTPU is the problem obtained by selecting one disjunct from each constraint. Note the variables and parameters of a component STPU are all those of the parent DTPU, including those not contained in the chosen disjuncts:

Definition 2 (Component STPU). *Let $P = \langle V_d, V_u, R_d, R_u, C\rangle$ be a DTPU. A* component STPU*, P' of P is an STPU with variables V_d and parameters V_u, and executable links R'_d and contingent links R'_u selected as follows. For each $c \in C$, where $c = r_u^1 \vee \cdots \vee r_u^k \vee r_d^{k+1} \vee \cdots \vee r_d^\ell$, for $r_d^i \in R_d$ and $r_u^j \in R_u$, there exists $i \in [1, \ell]$ s.t. $r_d^i \in R'_d$ (if $i \leq k$) or $r_u^i \in R_u$ (if $i > k$).* □

Example 2. Let P be the DTPU with decision variables $V_d = \{X, Y\}$ and parameter $V_u = \{Z\}$. Let all domains be integers, with $X \in [5]$, $Y \in [6]$, and $Z \in [5, 6]$. Suppose the sole constraint is $c_1 : Z-X = 0 \vee Z-Y = 0$. There are two component STPUs, one for each disjunct of c_1. Component STPU P_1 has constraint $Z - X = 0$ and component STPU P_2 has constraint $Z - Y = 0$; both have variables $V_d = \{X, Y\}$ and parameter $V_u = \{Z\}$ with the same domains as P. □

In an STPU, it is prohibited for two contingent links to 'meet' at the same timepoint, and contingent links between two parameters are also prohibited. Thus each parameter Y is involved in exactly one contingent link. We inherit these technical restrictions for DTPUs. They are without loss of generality as they can be overcome by inserting zero-length executable links [3].

4 Controllability

Having introduced the DTPU, we now extend the notions of controllability to DTPUs. In the presence of uncertainty, determining controllability is the key temporal reasoning problem [16]. The important new possibility is of a semi-executable DTPU constraint.

4.1 Strong Controllability

A DTPU is *Strongly Controllable* (SC) if there exists a decision that, combined with any realisation, satisfies the constraints. In other words, the agent has a way to assign values to the decision variables such that, given any values for the parameters, at least one disjunct on each constraint is satisfied: $\exists s_d : \forall s_u, C(s_d \cup s_u)$. Note this means that a DTPU is SC if some component STPU is SC. The converse is false, however, since in a SC decision, $s_d \cup s_u$ may satisfy different disjuncts in different realisations.

Example 3. Consider the DTPU P of Example 2. P is SC with the (only possible) strategy, i.e. $X = 5$ and $Y = 6$. It is easy to see that neither of the two component STPUs P_1 and P_2 is SC. □

We can define a stronger notion, not found in STPUs: a DTPU is *Absolutely Strongly Controllable* (ASC) if it is SC for a decision s_d where the same disjuncts of each constraint are satisfied in all realisations. In other words, there is a component STPU which is SC not only w.r.t. its own set of realisations but also w.r.t. the set of realisations of the whole DTPU. Clearly, ASC implies SC, and moreover a DTPU is ASC iff some component STPU is SC. Summarizing:

Theorem 1. *A DTPU is strongly controllable if it contains at least one strongly controllable STPU. The converse holds iff the DTPU is absolutely strongly controllable.*

4.2 Dynamic Controllability

A DTPU is *Dynamically Controllable* (DC) if there exists an incremental, online decision that, given knowledge of the values of all parameters that have occurred until now, can be dynamically extended to a decision in the future, whatever the values of the parameters that have not yet occurred.

Note this means that a DTPU is DC if some component STPU is DC, provided the decision for the component STPU can be consistently extended to a decision to all decision variables in the complete DTPU. Again, the converse is false in general.

Example 4. Let P be the DTPU with decision variables $V_d = \{X, Y\}$ and parameter $V_u = \{Z\}$. Let the domains be $X \in [4, 5]$, $Y \in [6, 7]$, and $Z \in [4, 6]$. Suppose the sole constraint $c_1 : Z - X = 0 \vee Z - Y = 0$. Now P is DC with the following strategy: at time 4 set $Y = 6$; at time 5 if Z has been observed (i.e. if $Z = 4$) set $X = 4$, otherwise set $X = 5$. There are two component STPUs, one for each disjunct of c_1, and it can be checked that neither is DC. □

We define that a DTPU is *Absolutely Dynamically Controllable* (ADC) if it is DC and, at any point in time, the strategy satisfies the same disjuncts in all future realisations. ADC implies DC, and a DTPU is ADC iff some component STPU is DC.

Theorem 2. *A DTPU is dynamically controllable if it contains at least one dynamically controllable STPU. The converse holds iff the DTPU is ADC.* □

4.3 Weak Controllability

A DTPU is *Weakly Controllable* (WC) if for any realisation there exists a decision that, combined with that realisation, satisfies the constraints: $\forall s_u\ \exists s_d : C(s_d \cup s_u)$. In other words, every realisation can be consistently extended to a decision. Once more, a DTPU is WC if some component STPU is WC, but the converse is false.

Example 5. Let P be the DTPU with decision variable $V_d = \{X\}$ and parameter $V_u = \{Z\}$. Let the domains be $X \in \{4, 6\}$, and $Z \in [5, 6]$. Suppose the sole constraint $c_1 : Z - X = 0 \vee Z - X = 1$. Now P is WC with the following strategy: if $Z = 5$ set $X = 4$, while if $Z = 6$ set $X = 6$. There are again two component STPUs, one for each disjunct of c_1, and neither is WC. □

For completeness we define that a DTPU is *Absolutely Weakly Controllable* (AWC) if there exist decisions such that the same disjuncts are satisfied by each of these decisions. AWC implies WC, and a DTPU is AWC iff some component STPU is WC.

Theorem 3. *A DTPU is weakly controllable if it contains at least one weakly controllable STPU. The converse holds iff the DTPU is absolutely weakly controllable.* □

4.4 Trivial Controllability

The situation where every constraint is semi-executable brings the possibility that the agent could satisfy the DTPU without regard to the contingent links, by satisfying at least one executable link in each constraint. This situation, which cannot occur for an STPU, introduces a new form of controllability which has no STPU analogue.

Definition 3. *A component STPU of a DTPU is a* component STP *if it contains only executable links. A DTPU is* Trivially Controllable *(TC) if one or more of its component STPUs is a consistent component STP.*

Consistency of a component STP is required to prove the following. Note that even if some component STPUs of a DTPU are STPs, if none of these component STPs are consistent then the DTPU need not be TC, nor even weakly controllable.

Theorem 4. *A trivially controllable DTPU is absolutely strongly controllable.* □

Since SC $\implies$ DC $\implies$ WC (inherited from the STPU case), by testing whether a DTPU is trivially controllable, we have a sufficient but not necessary condition for all other types of controllability. Testing for this condition requires, in the worst case, checking the consistency of every component STP. While there may be exponentially-many such STPs to check, the consistency of each can be determined relatively quickly compared to other inference tasks for DTPUs. Further, determining whether trivially controllable can occur at all is simple: it cannot occur if the DTPU contains any contingent constraint.

Example 6. The other constraints of Example 1, not shown here, are all single-disjunct executable constraints [7]. Thus, since the component STP arising with the second disjunct of (3) is consistent, the whole problem is trivially controllable. While so for this particularly simple example of a DTPU, more interesting DTPUs are not TC in general — such as even Examples 3–5. □

4.5 Relation Between Controllability Notions

As with STPUs, the controllability notions are ordered by logical implication from strong to weak. In addition, the absolute form of each controllability notion implies the non-absolute form: ASC $\implies$ SC, etc. Finally, ASC $\implies$ ADC $\implies$ AWC. The lattice of logical implications among the notions is shown in Figure 2.

5 Determining the Controllability of a DTPU

We now consider how to check which levels of controllability hold for a DTPU. As noted earlier, TC is a relatively easy condition to check and is a sufficient but not necessary condition for the other levels of controllability. The simple method to check TC,

$$\begin{tikzcd}
\mathrm{TC} \arrow[r, Rightarrow] & \mathrm{ASC} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathrm{SC} \arrow[d, Rightarrow] \\
& \mathrm{ADC} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathrm{DC} \arrow[d, Rightarrow] \\
& \mathrm{AWC} \arrow[r, Rightarrow] & \mathrm{WC}
\end{tikzcd}$$

Figure2. Logical relationship between consistency notions

by generate-and-test over component STPs, need consider only component STPs when testing for TC. That is, in searching the space of component STPUs, we generate only STPs: contingent disjuncts may be ignored. Since there are up to $\mathcal{O}(k^m)$ component STPs, and checking the consistency of each is polynomial in time and space, the worst-case complexity class is **NP**.

In the methods that follow for testing for the other levels of controllability, we can test TC as an initial first step. Checking whether the DTPU has any possibility of being TC has cost linear in the number of disjuncts ($\mathcal{O}(km)$), which is negligible. If TC is possible, the cost of testing for it by the search is still relatively modest compared to the methods described below.

5.1 Strong Controllability

Strong Controllability corresponds to the existence of a decision that satisfies the constraints in all realisations. A sound and complete algorithm for testing SC is to directly search for such a decision. For each possible s_d, consider all realisations s_u. Test whether s_d is a solution to each realised DTP in turn.

This simple approach suffers from a doubly-exponential search space. There are exponentially-many decisions, and we must consider them all until either we find one that satisfies all realisations, or until we prove that none such exists. For each candidate s_d, there are exponentially-many realisations to check; the check is to test whether $s_d \cup s_u$ is a solution to the realised DTP, which has worst-case time complexity $\mathcal{O}(km)$

An algorithm for determining the SC of an STPU is given by [17]. It operates in polynomial time by transforming the STPU into an STP such that the latter problem is consistent iff the STPU is SC. It does this by forming a 'worst case' constraint for each constraint in the problem; the transformation is linear in the number of constraints.

We can apply this transformation to each disjunct of a DTPU. This results in a DTP, which we can check for consistency by classical DTP algorithms [13]. The complexity class is **NP**, which is a significant improvement on the simple search for a SC decision. While this method provides only a sufficient condition for SC (due to Theorem 1), it does give a necessary and sufficient for testing for ASC. The complexity is thus the same as solving a classical DTP, i.e. **NP**, the best we can expect to achieve for a DTPU.

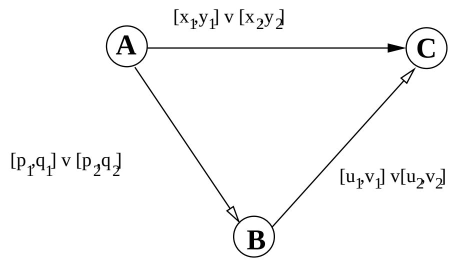


Figure3. Triangular DTPU of the single reduction case

5.2 Algorithm D-SC

We now describe a sound and complete algorithm, D-SC, to test whether a DTPU is SC by reducing the problem to that of testing the consistency of a DTP. The proposed algorithm relies on reductions performed on specific DTPUs. We consider first a triangular TCSPU (that is, a DTPU where each disjunctive temporal constraint is defined on only a single pair of variables) of the form ABC, where A and B are two decision variables time-points and C is a contingent time-point. This is the *single reduction* case. We will then generalize to arbitrary DTPUs, beginning with a single contingent link.

Single Reduction The first type of problem we consider is depicted in Figure 3. Before addressing the disjunctive case, let us recall how the non-disjunctive STPU case is handled in order to enforce SC. In the STPU case there is only one disjunct (e.g. those with index 1 in Figure 3 on each constraint). In [3] it is shown that a triangular STPU constraint network is strongly controllable iff the intersection of existing interval $[p_1, q_1]$ and the new one induced by the SC requirement, $[y_1 - v_1, x_1 - u_1]$, on constraint AB is not empty. In particular the new interval will contain durations corresponding to assignments to A and B which will be consistent with any possible assignment of C.

We now extend this reasoning to the case of disjunctive links. For simplicity, we consider only two disjuncts per link. However, what follows can be easily extended to an arbitrary number of disjuncts on each link. The links of the TCSPU which we consider are the following:

1. $C - A \in [x_1, y_1] \vee [x_2, y_2]$
2. $B - A \in [p_1, q_1] \vee [p_2, q_2]$
3. $C - B \in [u_1, v_1] \vee [u_2, v_2]$.

Consider the disjunct on AC. C can happen either between x_1 and y_1 or between x_2 and y_2 units of time after A. 'Nature' will decide which of the two disjuncts will be satisfied. For SC, the agent must choose a value of B which will be a priori consistent in both cases. In terms of constraints, the above TCSPU is SC iff the following is:

1. $C - A \in [x_1, y_1]$
2. $C - A \in [x_2, y_2]$
3. $B - A \in [p_1, q_1] \vee [p_2, q_2]$
4. $C - B \in [u_1, v_1] \vee [u_2, v_2]$

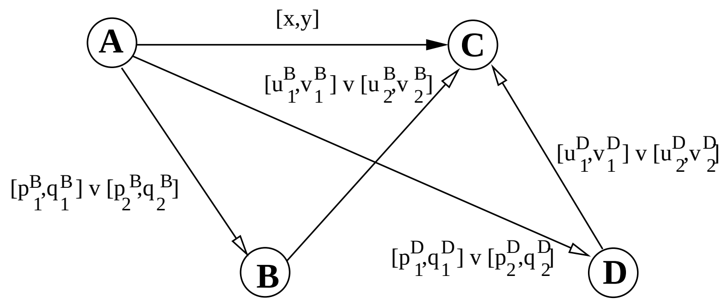


Figure4. Multiple reduction case

In fact, the difference between the two TCSPUs is that in the second the two contingent links are no longer in a disjunction. Intuitively, this corresponds to forcing the agent to control, at the same time, situations for which C occurs in $[x_1, y_1]$ and those for which it occurs in $[x_2, y_2]$. If this last TCSPU is not strongly controllable, then there is no unique assignment for B simultaneously consistent with all realisations in $[x_1, y_1]$ and all in $[x_2, y_2]$. This means that whatever value is chosen for B there will always be the possibility that 'Nature' will choose a value which there will be an inconsistency.

Extending the reasoning of the STPU case, the last TCSPU is SC is iff at least one of the following conditions holds for $i = 1$ and for $i = 2$:

- $[y_i - v_1, x_i - u_1] \cap [p_1, q_1] \neq \emptyset$
- $[y_i - v_1, x_i - u_1] \cap [p_2, q_2] \neq \emptyset$
- $[y_i - v_2, x_i - u_2] \cap [p_1, q_1] \neq \emptyset$
- $[y_i - v_2, x_i - u_2] \cap [p_2, q_2] \neq \emptyset$.

For example, if $[y_1 - v_1, x_1 - u_1] \cap [p_1, q_1] = [p'_1, q'_1] \neq \emptyset$ and $[y_2 - v_2, x_2 - u_2] \cap [p_1, q_1] = [p'_2, q'_2] \neq \emptyset$, then the DTPU will be controllable iff the new constraints on AB, i.e. intervals $[p'_1, q'_1]$ and $[p'_2, q'_2]$ are satisfied at the same time. This implies that we have reduced the problem of testing the SC of the TCSPU to that of checking the consistency of an STP defined on only the decision variables A and B. We call this a *single reduction*, since the constraints relate only one decision variable (other than A) to the execution of C — here, to the variable B.

Multiple reduction The next case we consider is when there is disjunction in the same link relating two or more decision variables to C. Suppose the DTPU at hand has four or more variables, only one of which (i.e. C) is a parameter, as depicted in Figure 4. The constraints of the DTPU shown are the following:

1. $C - A \in [x, y]$
2. $B - A \in [p_1^B, q_1^B] \vee [p_2^B, q_2^B] \vee D - A \in [p_1^D, q_1^D] \vee [p_2^D, q_2^D]$
3. $C - B \in [u_1^B, v_1^B] \vee [u_2^B, v_2^B] \vee C - D \in [u_1^D, v_1^D] \vee [u_2^D, v_2^D]$.

For simplicity, this DTPU supposes only: two decision variables (other than A) related to C in a constraint, one link on the contingent constraint, and two links on the other constraints. However, the reasoning can be directly extended to an arbitrary number of decision variables related to C and an arbitrary number of links.

Let us briefly describe the scenario in words. We know that C will happen sometime between x and y after A. We must assign values to B and D in order to guarantee that whenever C happens at least one link on each of constraints 2 and 3 above will be satisfied. This can happen in one of the following situations:

– **Case 1:** there is a way to assign a value to B consistent with any value of C. In this case the assignment to D becomes redundant, since choosing a good value for B is sufficient to satisfy the links;
– **Case 2:** as above with B and D exchanging role;
– **Case 3:** both Case 1 and 2 hold at the same time;
– **Case 4:** there is a value for B which is consistent with a subset of the assignments to C and a value for D which is consistent with the remaining assignments to C.

Notice that Case 4 is the one which the DTPU differs radically from the STPU case. When we consider such a triangular DTPU, we check which of the above conditions are satisfied and we replace the DTPU with a set of executable constraints on AB and AD as follows. We call this the *multiple reduction*.

– **Case 1:** the problem is reduced to a single reduction case. Thus we replace the DTPU constraint network with the new link on AB: $[p_1'^B, q_1'^B] \cap [p_2'^B, q_2'^B]$.
– **Case 2:** as above, except that the new link is on AD: $[p_1'^D, q_1'^D] \cap [p_2'^D, q_2'^D]$.
– **Case 3:** the network is replaced by the following disjunctive constraint $B - A \in [p_1'^B, q_1'^B] \cap [p_2'^B, q_2'^B] \vee D - A \in [p_1'^D, q_1'^D] \cap [p_2'^D, q_2'^D]$.
– **Case 4:** this case holds iff for some $i, j, h, k = 1, 2$, $[x, y] \subseteq [q_i^B + u_j^B, p_i^B + v_j^B] \cup [q_h^D + u_k^D, p_h^D + v_k^D]$. The DTPU is replaced by two constraints: $B - A \in [p_i^B, q_i^B]$ and $D_A \in [p_i^D, q_i^D]$.

D-SC for a Generic DTPU We are now in a position to address a generic DTPU P and any of its contingent or semi-executable constraints containing two or more contingent links defined on different parameters. Consider replacing each such constraint with as many constraints as the contained contingent links. For example, consider the following semi-executable link T defined on executable time-points A,B and D and contingent time-points C and G: $C - A \in [x_1, y_1] \vee G - A \in [x_2, y_2] \vee D - B \in [p, q]$. We replace T by two semi-executable constraints $G - A \in [x_2, y_2] \vee D - B \in [p, q]$ and $C - A \in [x_1, y_1] \vee D - B \in [p, q]$. Denote by $Expand(P)$ the DTPU obtained from P by replacing each contingent and semi-executable link in this way.

An intuitive explanation is that a disjunction of contingent links 'contains' more uncertainty than the each of the single links. For example, constraint $C - A \in [x_1, y_1]$ means parameter C will occur at some (unpredictable) time between x_1 and y_1 after A. Instead, constraint $C - A \in [x_1, y_1] \vee G - A \in [x_2, y_2]$ means: *either* parameter C will occur at some (unpredictable) time between x_1 and y_1 after A *or* parameter G will occur at some (unpredictable) time between x_2 and y_2 after A *or* both.

Now consider any contingent or semi-executable link T of DTPU $Q = Expand(P)$. By definition it can contain only contingent links defined on the same parameter, say C. Consider now any other link of Q, T' with at least a constraint involving C. Then the sub-problem consisting of links T and T' is either a single or a multiple reduction case. Let $r_C(T, T')$ be the set of executable constraints produced by the reduction. Let us denote with T_C and T_{-C} the set of links of T resp. involving and not

Algorithm 1 Testing for strong controllability of a DTPU

Input: DTPU P

if P is trivially controllable **then**
    return true
DTPU Q= $Expand(P)$
Initialize DTP P': $var(P') \leftarrow Executables(P)$, $Links(P') \leftarrow \emptyset$;
for every constraint T in P **do**
    if T has no contingent link **then**
        $Links(P') \leftarrow Links(P') \cup T$
    else
        for every other constraint T' with links involving C **do**
            Compute $r_C(T,T'), T_{new} \leftarrow r_C(T,T') \vee T_{-C}, T'_{new} \leftarrow r_C(T,T') \vee T'_{-C}$
            $Links(P') \leftarrow Links(P') \cup T_{new}$
            if T'_{new} executable **then**
                $Links(P') \leftarrow Links(P') \cup T'_{new}$
if DTP P' consistent **then**
    return true
return false

involving variable C. Then we can generate constraints $T_{new} = r_C(T,T') \vee T_{-C}$ and $T'_{new} = r_C(T,T') \vee T'_{-C}$. Notice that T_{new} is now an executable constraint; T'_{new} by contrast may contain contingent links on some other parameter G.

Algorithm D-SC checks whether a given DTPU is SC by applying the above reductions to all the sub-networks of the DTPU given in input, obtaining a DTP. The pseudo-code is given in Figure 1. D-SC first checks if the DTPU P, given as input, is trivially controllable. If not, it proceeds by *expanding* the DTPU to give another DTPU Q, in which each constraint has at most one contingent link, and that is equivalent to P in terms of SC. Then for each constraint containing at least a contingent link it considers all the other constraints which involve variable C. For each such link it performs one of the two reductions and adds the generated constraint to a new DTP P', defined on the executable time-points of P. Finally, when each contingent link in Q has been considered, it tests whether P' is consistent.

Theorem 5. *DTPU P is SC iff DTP P' produced by D-SC is consistent.* □

Theorem 5 allows us to conclude that D-SC is a sound and complete algorithm for testing the SC of a DTPU. Moreover, it also implies the following complexity result.

Theorem 6. *The complexity of algorithm D-SC is $O(n^3k^m)$, where n is the number of variables, m the number of constraints and k the number of links in each constraint.*

Proof. Obtaining the expanded DTPU Q takes linear time in the number of links: $O(km)$. DTP P' is obtained from Q in time $O(m^2)$. Thus the dominating complexity is that of testing if P' is consistent, i.e. $O(n^3k^m)$. □

5.3 Dynamic Controllability

Dynamic controllability of an STPU can be determined with the same complexity as strong controllability [3]. This result, that checking whether an STPU is DC has poly-

Algorithm 2 Testing for weak controllability of a DTPU

Input: DTPU P

1: **if** P is trivially controllable **then**
2: **return** true
3: Choose and remove a Y_i from V_u
4: Replace the interval on Y_i with its lower bound $[l_i, l_i]$
5: **if** $WeaklyControllable(P)$ **then**
6: Replace the interval on Y_i with its upper bound $[u_i, u_i]$
7: **return** $WeaklyControllable(P)$

nomial time complexity, was a surprise when discovered, overturning a previous conjecture [16]. The interesting challenge is the corresponding: does determining whether a DTPU is DC have the same complexity as determining whether it is SC?

While this question is presently open, we can describe an algorithm for testing DC that again provides a suffi cient but not necessary check. Enumerate the component STPUs and apply the polynomial-time DC-checking algorithm of [3] to each, until one is found DC or the search space of component STPUs is exhausted. If one component STPU is found to be DC, then the DTPU is DC. If all are proved not DC, then we can conclude that the DTPU is not ADC; by Theorem 2 we cannot conclude that it is not DC. The complexity class is **NP**, since there are exponentially-many such component STPUs, and the test we apply to each is in **P**.

5.4 Weak Controllability

Given a realisation s_u, a DTPU reduces to a classical DTP. Thus weak controllability can be determined by enumerating the possible realisations and solving one DTP for each. In fact, we need consider only the 'end-point' realisations:

Theorem 7. *Let P be a DTPU with parameters $V_u = \{Y_1, \ldots, Y_p\}$ where Y_i has domain $Y_i \in [l_i, u_i]$. P is weakly controllable iff for every realisation s_u in $\prod_i \{l_i, u_i\}$, a decision exists that satisfies the constraints under s_u.*

Proof. If P is WC then under every realisation (not just the end-points) a decision exists satisfying the constraints. Conversely, the proof relies on the monotonicity of the disjuncts of a DTP; it exactly parallels the same theorem for STPUs: Properties 4.1 and 4.2 in [16]. □

The method is given as Algorithm 2. Although end-point realisations suffi ce, there are still exponentially-many to consider. While naive, Algorithm 2 is the direct analogue of determining WC for STPUs. The co-problem of checking WC is fi nding a realisation such that the realised DTP is inconsistent. Since checking the inconsistency of a DTPU is in **NP**, this co-problem belongs to Σ_2^p.[3]

Testing for absolute weak controllability is in fact harder than testing for WC. The proof that a DTPU is AWC — a decision $s_d^{(u)}$ for every realisation s_u, such that, for

[3] The complexity class Σ_2^p (or $\mathbf{NP}^{\mathbf{NP}}$) is the class of decision problems that can be solved in polynomial time on a non-deterministic Turing machine using **NP**-oracles [5].

Table1. Complexity of the controllability notions

	STPU	DTPU
no uncertainty	**P**	**NP**-hard
trivial controllability	—	**NP**-hard
absolute strong controllability	—	**NP**-hard
strong controllability	**P**	**NP**-hard
absolute dynamic controllability	—	?
dynamic controllability	**P**	?
absolute weak controllability	—	$P^{\Sigma_2^p}$
weak controllability	co-**NP** complete	co-Σ_2^p complete

every realisation, the corresponding solutions $s_d^{(u)} \cup s_u$ satisfies the same disjuncts — requires us to prove not only the existence of a solution for every realisation, but demonstrate that these decisions collectively share the stated property.

5.5 Complexity of Controllability Notions

The algorithms described in this section are initial approaches to the task of reasoning over DTPUs. As by-and-large crude methods, D-SC excepted, they have at heart search over one or more of the space of decisions, realisations, or component STPUs, combined with application of algorithms for STPs, DTPs, and STPUs.

Table 1 compares the complexity of STPU and DTPU solving for the different levels of controllability. The results for DTPUs are tentative; they are upper bounds based on the methods described. For instance, it is an open question whether dynamic controllability is as hard as weak controllability.

In developing more sophisticated approaches to DTPU reasoning problems, we must exploit both reasoning reuse and solution reuse [15]. The path is indicated by DTP solvers that operate on a search space of component STPs; efficient solvers prune the space by search-theoretic and DTP-semantic techniques [13,6].

6 Discussion and Future Work

As a step towards modelling planning and scheduling problems with both disjunctive temporal constraints and contingent events, we have proposed the Disjunctive Temporal Problem with Uncertainty. The DTPU extends the DTP to account for contingent time-points not under the control of the execution agent. We introduced differing levels of controllability and proposed first methods to determine whether strong, weak, and trivial controllability hold, together with an initial complexity analysis.

While the DTPU extends previous work on the STPU, much future work is needed. On the theory side, our main task is to formulate and prove further properties of DTPUs with the aim of resolving the complexity of testing dynamic controllability. On the practical side, our task is to implement the algorithms described in Section 5 and test them on sample problems.

Beyond these aspects, we are interested to combine the DTPU framework with user preferences [6], in a similar manner to [8]. Another interesting direction will be that of extending the framework to allow different levels of uncertainty on the parameters, drawing from [12,14].

Acknowledgments. We thank M. Pollack and T. Uribe for useful discussions. This material is based upon work supported by the Defense Advanced Research Projects Agency, through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

References

1. A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based decision procedure for the boolean combination of difference constraints. In *Proc. of SAT'04*, 2004.
2. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1–3):61–95, 1991.
3. P. Morris, N. Muscettola, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proc. of IJCAI'01*, pages 494–502, Seattle, WA, 2001.
4. A. Oddi and A. Cesta. Incremental forward checking for the disjunctive temporal problem. In *Proc. of ECAI-00*, pages 108–112, Berlin, Germany, Aug. 2000.
5. C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, Reading, MA, 1994.
6. B. Peintner and M. E. Pollack. Low-cost addition of preferences to DTPs and TCSPs. In *Proc. of AAAI-2004*, pages 723–728, San Jose, CA, 2004.
7. M. E. Pollack, C. E. McCarthy, S. Ramakrishnan, I. Tsamardinos, L. Brown, S. Carrion, D. Colbry, C. Orosz, and B. Peintner. Autominder: A planning, monitoring, and reminding assistive agent. In *Proc. of ICIAS'02*, 2002.
8. F. Rossi, K. B. Venable, and N. Yorke-Smith. Controllability of soft temporal constraint problems. In *Proc. of CP'04*, LNCS 3258, pages 588–603, Toronto, Canada, 2004.
9. T. K. Satish Kumar. On the tractability of restricted disjunctive temporal problems. In *Proc. of ICAPS'05*, pages 110–119, Monterey, CA, June 2005. To appear.
10. P. Schwartz and M. E. Pollack. Planning with disjunctive temporal constraints. In *Proc. of ICAPS'04 Workshop on Integrating Planning into Scheduling*, Whistler, Canada, June 2004.
11. K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.
12. I. Tsamardinos. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence, Second Hellenic Conf. on AI (SETN 2002)*, LNCS 2308, pages 97–108. Springer, 2002.
13. I. Tsamardinos and M. E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1–2):43–89, 2003.
14. I. Tsamardinos, M. E. Pollack, and S. Ramakrishnan. Assessing the probability of legal execution of plans with temporal uncertainty. In *Proc. of ICAPS'03 Workshop on Planning Under Uncertainty and Incomplete Information*, Trento, Italy, June 2003.
15. G. Verfaillie and N. Jussien. Commented bibliography on dynamic constraint solving. `www.emn.fr/x-info/jussien/CP03tutorial/`, Oct. 2003.
16. T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: From consistency to controllabilities. *JETAI*, 11(1):23–45, 1999.
17. T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. of ECAI-96*, pages 48–52, Budapest, Hungary, Aug. 1996.

Uncertainty in soft constraint problems

Maria Silvia Pini, Francesca Rossi and Brent Venable

Department of Pure and Applied Mathematics, University of Padova, Italy
E-mail: {mpini,frossi,kvenable}@math.unipd.it

Abstract Preferences and uncertainty occur in many real-life problems. The theory of possibility is one way of dealing with uncertainty, which allows for easy integration with fuzzy preferences. In this paper we consider an existing technique to perform such an integration and, while following the same basic idea, we propose to generalize it to other classes of preferences and to probabilistic uncertainty while maintaining certain desirable properties.

1 Introduction

Preferences and uncertainty occur in many real-life problems. In this paper we are concerned with the coexistence of such concepts in the same problem. In particular, we consider uncertainty that comes from lack of data or imprecise knowledge.

The theory of possibility [6, 10] is one non-probabilistic way of dealing with uncertainty, which allows for easy integration with fuzzy preferences [3]. In fact, both possibilities and fuzzy preferences are values between 0 and 1 associated to events and express the level of plausibility that the event will occur, or its preference.

In our context, we will describe a real-life problem as set of variables with finite domains and a set of soft constraints among subsets of the variables. A variable will be said to be uncertain if we cannot decide its value. In this case, we will associate a possibility degree to each value in its domain, which will tell how plausible it is that the variable will get that value.

Soft constraints [2] allow to express preferences over the instantiations of the variables of the constraints. In particular, fuzzy preferences are values between 0 and 1, which are combined using the *min* operator, and are ordered in such a way that higher values denote better preferences. Probabilistic preferences are similar to fuzzy ones, except that they are combined via multiplication: the goal is to maximize the product of the preferences. Weighted constraints use instead preferences representing costs which are combined by summing them: the goal is to minimize the sum of the weights.

In this paper we consider existing techniques to integrate fuzzy preferences and uncertainty, which use possibility theory [3, 9]. In particular, the approach in [9] allows one to handle uncertainty within a fuzzy optimization engine, and at the same time to observe separately the preference level and the robustness of the complete instantiations. This approach has certain desirable properties, that we describe formally.

We then generalize the approach, and its properties, in order to use it also for other classes of soft constraints, not necessarily fuzzy, by pointing out a sufficient condition for the properties to hold. This allows us to handle the coexistence of preferences and uncertainty in a more general setting. We also consider uncertainty expressed by a probabilistic distribution rather than a possibilistic one, and we show that in this setting the same properties hold.

2 Soft constraints

Soft constraints [2] are a very general formalism to describe quantitative preferences. In general, a soft constraint is just pair $\langle def, con\rangle$, where *con* is the set of variables of the constraint (that is, its scope), and def is a function from the Cartesian product of the domains of the variables in *con* to a preference set, say A. Therefore def defines the constraint, by associating a level of preference from A to each assignment of values to the variables of the constraint.

Set A can be totally or partially ordered, and its ordering, denoted by $\leq$, can be used to order the assignments of values to variables: assignments corresponding to higher preferences are more preferred. Moreover, a combination operation $\times$ should be defined over A, to combine different constraints and generate the preference level of an assignment of values to variables which range over the scopes of several constraints. More precisely, A should have properties similar to a semiring. We will therefore say that a soft constraint is defined *over semiring* A. For more details on semiring-based soft constraints, see [2].

A soft constraint problem is usually denoted by a tuple $\langle S, V, C\rangle$ where S is a semiring, V is a set of variables, and C is a set of soft constraints over S whose scopes are subsets of these variables. An optimal solution of a soft constraint problem is an assignment of its variables which is optimal according to the ordering associated to the semiring.

This general description of soft constraints instantiates to several classes of concrete constraints:

- *Fuzzy constraints*: when $A = [0, 1]$, $\leq$ is derived by the *max* operator, and the combination operator is *min*. This means that a fuzzy constraint associates an element between 0 and 1 to each instantiation of its variables, that values closer to 1 denote a higher preference, and that the preferences of two or more constraints are combined by taking their minimum value.
- *Hard constraints*: they can also be described by this framework, by just choosing $A = \{true, false\}$, $\leq$ derived by logical *or* (thus 1 is better than 0), and combination is logical *and*.
- *Weighted constraints*: they are soft constraints where each assignment of values to variables has a weight, and the goal is to minimize the sum of the weights: this can be cast by choosing A as the set of possible weights, by deriving the ordering by the *min* operator, and by using the *sum* as the combination operator.

– *Probabilistic constraints*: they are soft constraints where each assignment is associated to a probability, which informally represents the chance for the assignment to satify the constraint in the real problem. Constraints are then combined by multiplying the associated probabilities, and *max* is used to induce the ordering over preferences: the best solutions have the highest probability.

The concept of fuzzy constraint, as defined above, was originally based on the notion of fuzzy set [5, 7, 10]. A fuzzy set A is a subset of a referential set U whose boundaries are gradual. More formally: the *membership function* μ_A of a fuzzy set A assigns to each element $u \in U$ its degree of membership $\mu_A(u)$ usually taking values in $[0, 1]$. If $\mu_A(u) = 1$, it means that u belongs to A, while $\mu_A(u) = 0$ means that u does not belong to A. If $\mu_A(u)$ is between 0 and 1, then it means that $u \in A$ with degree $\mu_A(u)$.

Fuzzy constraints use the notion of fuzzy sets to describe the level of preference of a certain assignment of values to variables. More precisely, a *soft fuzzy constraint* [3] C on variables $\{x_1, \ldots, x_n\}$ is associated with a $fuzzy$ relation R, i.e. a fuzzy subset of $D_1 \times \cdots \times D_n$ of values that more or less satisfy C. A membership function μ_R is associated with relation R and specifies for each tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ the level of satisfaction $\mu_R(d_1, \ldots, d_n)$ in a set L, which is totally ordered (e.g. [0,1]). In particular, $\mu_R(d_1, \ldots, d_n) = 1$ if tuple $(d_1, \ldots, d_n)$ totally satisfies C, $\mu_R(d_1, \ldots, d_n) = 0$ if it totally violates C, and $0 < \mu_R(d_1, \ldots, d_n) < 1$ if it partially satisfies C. Moreover, $\mu_R(d_1, \ldots, d_n) > \mu_R(d'_1, \ldots, d'_n)$ means that tuple $(d_1, \ldots, d_n)$ is better than tuple $(d'_1, \ldots, d'_n)$.

In the following we will use two operations on fuzzy constraints [3]: projection and combination. The *projection* of a fuzzy constraint, represented by fuzzy relation R on variables $\{x_1, \ldots, x_k\} \subseteq V(R) = \{x_1, \ldots, x_n\}$, is a fuzzy relation $R^{\downarrow\{x_1,\ldots,x_k\}}$ defined on $\{x_1, \ldots, x_k\}$ such that: $\mu_{R^{\downarrow\{x_1,\ldots,x_k\}}}(d_1, \ldots, d_k) = sup_{\{d=(d_1,\ldots,d_n)|d^{\downarrow\{x_1,\ldots,x_k\}}=(d_1,\ldots,d_k)\}}\ \mu_R(d)$. The *conjunctive combination* of two fuzzy constraints, represented by fuzzy relations R_i and R_j, is a fuzzy relation $R_i \otimes R_j$ defined on variables $V(R_i) \cup V(R_j)$ such that: $\mu_{R_i \otimes R_j}(d_1, \ldots, d_k) = min(\mu_{R_i}(d_1, \ldots, d_k)^{\downarrow V(R_i)}, \mu_{R_j}(d_1, \ldots, d_k)^{\downarrow V(R_j)})$ where $\mu_{R_i \otimes R_j}(d_1, \ldots, d_k)$ evaluates to what extent $(d_1, \ldots, d_k)$ satisfies both C_i and C_j.

3 Possibility theory

A *possibility distribution* [10] is the membership function of a fuzzy set A attached to a single-valued variable x. It is denoted $\pi_x = \mu_A$ and represents the set of more or less plausible, mutually exclusive values of x. A possibility distribution is similar to a probability density. However, $\pi_x(u) = 1$ only means that $x = u$ is a plausible situation, which cannot be excluded. Thus, a degree of possibility can be viewed as an upper bound of a degree of probability.

Possibility theory encodes incomplete knowledge while probability accounts for random and accurately observed phenomena. In particular, the complete ignorance about x is expressed by $\pi_x(u) = 1$, for all $u \in U$, since in this case all values u are plausible for x and so it is impossible to exclude any of them. Whereas, $\pi_x(\bar{u}) = 1$ for a specific value $\bar{u}$ and $\pi_x(u) = 0$ otherwise, expresses the complete knowledge about x, because in this case only the value $\bar{u}$ is plausible for x.

The *possibility* of an event "$x \in E$", $E \subseteq U$, denoted by $\Pi(x \in E)$, is formally $\Pi(x \in E) = sup_u min(\pi_x(u), \mu_E(u)) = sup_{u \in E} \pi_x(u)$.

If an event has possibility equal to 1, it means that it is totally possible. However, it could also not happen. Therefore it means that we are completely ignorant about its occurrence. On the contrary, having a possibility equal to 0 means that the event will not happen.

The dual measure of *necessity* of "$x \in E$", denoted by $N(x \in E)$, evaluates the extent to which "$x \in E$" is *certainly* true, i.e. to what extent the proposition "$x \in E$" is implied by the item of information "$x \in A$": $N(x \in E) = inf_u max(c(\pi_x(u)), \mu_E(u)) = inf_{u \notin E}(c(\pi_x(u))) = 1 - \Pi(x \in E^C)$, where c is the order reversing map such that $c(p) = 1 - p$ and E^C is the complement of E in U.

$N(x \in E) = 1$ when it is certain that $x \in E$. On the contrary, having necessity equal to 0 means that the event is not necessary at all, although it may happen. In fact, $N(x \in E) = 0$ iff $\Pi(x \in E^C) = 1$.

4 Uncertainty in soft constraints

Whereas in usual soft constraint problems all the variables are assumed to be controllable, that is, their value can be decided according to the constraints which relate them to other variables, in many real-world problems uncertain parameters must be used. Such parameters are associated with variables which are not under the user's direct control and thus cannot be assigned. Only Nature will assign them.

Formally, we can define an *uncertain soft constraint problem* as a tuple $\langle S, V_c, V_u, C \rangle$, where S is a semiring, V_c is the set of controllable variables, V_u is the set of uncontrollable variables, and C is the set of soft constraints. The soft constraints in C may involve any subset of variables of $V_c \cup V_u$.

While in a classical soft constraint problem we can decide how to assign the variables to make the assignment optimal, in the presence of uncertain parameters we must assign values to the controllable variables guessing what Nature will do with the uncontrollable variables. So, in this paper an optimal solution for an uncertain soft constraint problem is an assignment of values to the variables in V_c such that, *whatever* Nature will decide for the variables in V_u, the overall assignment will be optimal. This is a pessimistic view and other definitions of solutions can be considered [1].

Moreover, the uncontrollable variables can be equipped with additional information on the likelyhood of their values. Such information can be given in several ways, depending on the amount and precision of knowledge we have. In this paper we will consider two ways of expressing such information: possibilities and probabilities. This information can be used to infer new soft constraints over the controllable variables, expressing the compatibility of the controllable parts of the problem with the uncertain parameters, and can be used to change the notion of optimal solution.

The next section describes two existing approaches [3,9] for integrating fuzzy constraints and uncertainty given by possibilities. In both, the original problem is replaced by another one without uncontrollable variables and with new soft constraints depending on the possibilistic distributions. In [9] the two sets of constraints are kept separate, thus allowing for a fine discrimination between preferences and robustness to uncertainty.

5 Unifying fuzzy preferences and uncertainty via possibility theory

In [3] it is shown how it is possible to replace a fuzzy constraint involving at least one uncontrollable variable with a fuzzy constraint over controllable variables only. Consider a fuzzy constraint C, represented by the fuzzy relation R, which relates a set of controllable variables $X = \{x_1, \ldots, x_n\}$ to a set of uncertain parameters $Z = \{z_1, \ldots, z_k\}$ with domains $A_1, \ldots, A_k$. The knowledge of the uncertain parameters is modeled with the possibility distribution π_Z defined on $A_Z = A_1 \times \cdots \times A_k$. The constraint C is considered satisfied by the assignment $d = (d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ if, *whatever the values of* Z, $z = (z_1, \ldots, z_k)$, *d is compatible with z*, i.e., the set of possible values for z is included in $T = (R \otimes \{(d_1, \ldots, d_n)\})^{\downarrow Z}$. Therefore $\mu_T(a) = \mu_R(a, d)$ and $\mu'(d) = \mu'_R(d) = N(d\, satisfies\, C) = N(z \in T) = inf_{a \in A_Z} max(\mu_T(a), c(\pi_Z(a)))$ $= c(sup_{a \in A_Z} min(c(\mu_T(a)), \pi_Z(a)))$. If C is a hard constraint, then the formula above still applies, and becomes the following one: $N(d\, satisfies\, C) = inf_{a \notin T = (R \cap \{d\})^{\downarrow D_Z}} c(\pi_Z(a))$.

The method proposed in [3], which we call *Algorithm DFP* (by the name of the authors), for managing uncertainty in a fuzzy CSP, is the following: It starts from an uncertain fuzzy CSP, say P. P is then reduced to a fuzzy constraint problem P': all the constraints which link uncertain parameters to decision variables are replaced by fuzzy constraints only among the decision variables. The new preference levels of the decision variables in such new constraints are computed by applying the specific procedure defining μ'. P' has only fuzzy constraints, therefore it can be solved by applying the usual method for solving fuzzy CSPs, i.e. using the *min* operator to combine the constraints and choosing the complete assignments with the highest preference.

In [3] the following property is given:

Property 1. $\mu'(d) \geq \alpha$ iff, when $\pi_Z(a) > c(\alpha)$ then $\mu_R(d, a) \geq \alpha$, where a is the actual value of Z.

Moreover, from the definition of μ', the following two properties can also be proven [9].

Property 2. Given the possibilities of uncertain parameters, defined by π_Z, an assignment d to the decision variables $X_1, \ldots, X_k$, and two preference functions μ_1 and μ_2 such that $\mu_1(d,a) \leq \mu_2(d,a)$ for all a assignments to Z, then $\mu'_1(d) \leq \mu'_2(d)$.

Property 3. Given a preference function μ, an assignment d to the decision variables $X_1, \ldots, X_k$, and two possibility distributions π_1 and π_2 on Z, such that $\pi_1(a) \geq \pi_2(a)$ for all a, then $\mu'_1(d) \leq \mu'_2(d)$, where μ'_i is the preference function obtained considering π_i, i=1,2.

By using algorithm DFP, the preference of a complete assignment is the minimum value among all the preferences of the constraints, both the original fuzzy constraints and those obtained via the transformation which eliminates the uncontrollable variables. In other words, the overall preference for a solution is $min(F,U)$, where F is the minimum of the preferences in the initially given fuzzy constraints only on decision variables, and U is the minimum of the preferences of the new fuzzy constraints. This means that a low overall preference may be caused from a low preference in some of the new fuzzy constraints (when U is less than F), that is, a low compatibility with the uncertain events, or also from a low preference on some fuzzy constraint initially given only on decision variables (when F is less than U).

In [9] these two components (F and U) are kept separate, rather than combined with min. This is done by performing, for each constraint c involving both decision and uncertain variables (X and Z), a projection over the decision variables. This will create a new constraint c'' over X where, for each assignment of values to its variables, the preference is computed by assuming the best in the uncertain parameters. Since preferences are combined via the min operator, this new constraint will force the overall preference to be no higher than its best preference. Given an assignment to decision variables, we denote with P the minimum preference over these new *projection constraints*. Such a value P, combined with preference F given by the initial constraints, defines the new preference F_P.

The algorithm presented in [9], called *algorithm* **SP** (from *separation* and *projection*), is the following:

1. It starts from an uncertain fuzzy CSP with fuzzy constraints C.
2. All the constraints which link uncertain parameters to decision variables are replaced by fuzzy constraints only among the decision variables. Let us call C_u such new constraints.
3. It computes the projection constraints, say C_p.
4. For each complete assignment, it computes its overall preference as a pair $\langle F_P, U\rangle$, where $F_P = min(F,P)$ and F, P, and U are, respectively, the minimum preference over C, C_p, and C_u.

Let us consider the following example, where we have a complete assignment d with $F = 0.3$, $P = 0.9$, and $U = 0.9$, and another one d' with $F = 0.9$, $P = 0.9$, and $U = 0.3$. According to algorithm DFP, d is considered equally preferred to d' since d and d' have the same preference $min(F,U) = 0.3$. However, d and d' differ on both preference and robustness.

The approach based on algorithm **SP** preserves such a difference, by defining various semantics which exploit both elements of the pair $\langle F_P = min(F, P), U\rangle$ to deduce a solution ordering.

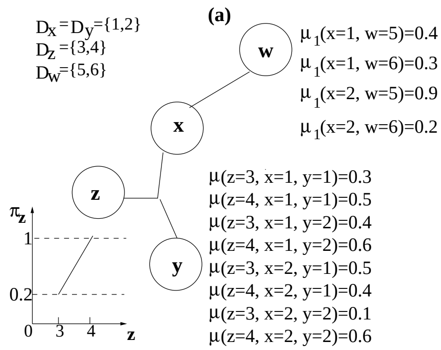


Figure1. An uncertain soft CSP.

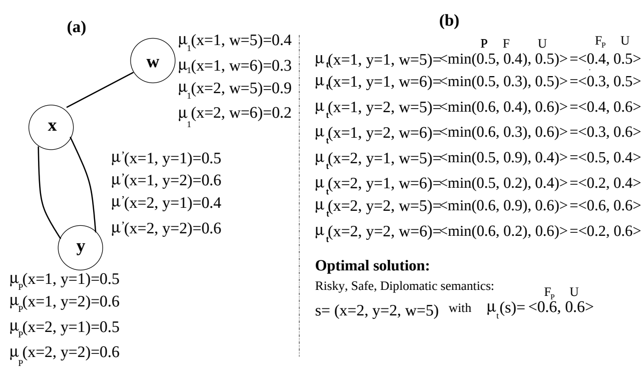


Figure2. Result of algorithm **SP** applied to the uncertain soft CSP in Figure 1, seen as an uncertain fuzzy CSP.

Figure 1 shows a soft CSP with uncertainty. There are three decision variables (X, Y, W), one uncertain variable (Z), and two constraints: C_{XYZ} with function μ and C_{XW} with function μ_1. The possibility distribution π_Z describes the plausibility of Z.

Figure 2 (a) shows the fuzzy CSP obtained from the soft CSP in Figure 1 seen as an uncertain fuzzy CSP, after applying step 2 and step 2 of algorithm **SP**. In Figure 2 (b) there are all the complete assignments to decision variables of the fuzzy CSP, with the overall preference μ_t.

Given an assignment d to the decision variables and the pair $\langle F_P, U\rangle$ computed as described above, F_P tells us how much d is preferred by the constraints, while U represents to what extent it is impossible to have a possible value of the uncertain parameters violating the constraints involving them. This means that $1 - U$ gives an idea of the risk of hitting a value of uncertain parameter that is inconsistent with d, hence U can be seen as a measure of the *certainty* (or *robustness*) of d.

U is computed as in [3], so Properties 1, 2 and 3 still hold. We recall that these properties state that U can increase in the following two cases: when the possibilities of the uncertain parameters remain fixed and the preferences of the constraints involving them increase, or when preferences are fixed but possibilities decrease.

Consider two solutions d and d' and the corresponding pairs of values $\langle F_P(d), U(d)\rangle = \langle a_1, b_1\rangle$ and $\langle F_P(d'), U(d')\rangle = \langle a_2, b_2\rangle$. The first semantics proposed in [9], called *Risky*, can be seen as a Lex ordering on pairs $\langle a_i, b_i\rangle$, with the first component as the most important feature. Informally, the idea is to give more importance to the preference level that can be reached in the best case (a higher a_i) considering less important a high risk of being inconsistent (a low certainty b_i).

The second semantics, called *Safe*, follows the opposite attitude with the respect to the previous one: it can be seen as a Lex ordering on pairs $\langle a_i, b_i\rangle$, with the second component as most important feature. Informally, the idea is to give more importance to the certainty level that can be reached (a higher b_i) considering less important having a high preference (a high a_i).

The third semantics, called *Diplomatic*, aims at giving the same importance to the two aspects of a solution: preference and certainty. This is obtained via the Pareto ordering on pairs $\langle a_i, b_i\rangle$, where both components have the same importance. The idea is that a pair is to be preferred to another only if it wins both on preference and certainty, leaving incomparable all the pairs that have one component higher and the other lower. Contrarily to the diplomatic semantics, the other two semantics produce a total order over the solutions.

In the example in Figure 2, solution $(x = 2, y = 2, w = 5)$ is optimal for all three semantics. In general, the comparison among the orders induced by the three semantics of [9] and the one in [3] can be seen in Table 1.

Table1. Comparison between the ordering in [3] and in [9].

[3]	Risky	Safe	Dipl.
$=$	$<,>,=$	$<,>,=$	$<,>,=,\bowtie$
$>$	$<,>$	$<,>$	$>,\bowtie$
$<$	$<,>$	$<,>$	$<,\bowtie$

6 A generalized approach

The methods described in the previous sections can handle only uncertainty in fuzzy constraints. In the following of the paper we will extend the method in [9] to other classes of soft constraints and to probabilistic uncertainty. In particular, we will consider the combinations of fuzzy, probabilistic, and weighted soft constraints with either possibilistic or probabilistic uncertainty.

To do this, we now redefine **SP** for generic soft constraints and generic uncertainty. Then we will define the extensions above as instances of this general framework.

We recall that in Fuzzy CSPs with uncertainty the preferences of the new constraints obtained removing the uncertain parameters are computed via the formula $\mu'(d) = inf_{a \in A_Z} max(\mu_R(d,a), c(\pi_Z(a)))$, where c is the order reversing map such that $c(p) = 1 - p$ and π_Z is the possibilistic distribution.

Let us consider any semiring $S = \{A, +, \times, 0, 1\}$, where $\leq_S$ is the semiring ordering on A (we denote incomparability with $\bowtie_S$). The formula above can be generalized to deal with any semiring as follows:

$$\mu'(d) = inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a)))$$

where

- $+$ refer to the one of the semiring operations, and inf is one of the bottom elements of A_Z, i.e. an element such that $\forall a' \in A_Z$ with $a' \neq a$ then $a' >_S a$ or $a' \bowtie_S a$.
- $[0,1] \subseteq A$;
- c is an bijection from [0,1] to [0,1] such that, for each $a_1, a_2 \in [0,1]$, $a_1 \leq a_2$ if and only if $c(a_1) \geq_S c(a_2)$; moreover, $c(c(a)) = a$ for all a. We will say that c is an order-reversing map w.r.t. semiring S;
- π_Z is a possibilistic or probabilistic distribution.

Notice that, by generalizing the formula, we do not change the set of values for the possibilities, which remains $[0,1]$. When working with other classes of soft constraints rather than fuzzy CSPs, functions μ_R associates a preference from set A to an assignment. In particular, μ_R is the preference function of the soft constraint R.

The algorithm **SP** presented in [9], can be generalized as follows:

1. It starts from an uncertain soft CSP $\langle S, V_c, V_u, C\rangle$;
2. All the constraints which link variables in V_u to variables in V_c are replaced by soft constraints defined by μ', described above, only among variables in V_c. Let us call C_u such new constraints.
3. All the constraints which link variables in V_u to variables in V_c are used to compute their projection over variables in V_c. The constraints obtained in this way are called C_p.
4. For each assignment of the variables in V_c, it computes its overall preference as the pair $\langle F_P, U\rangle$, where $F_P = F \times P$ and F, P, and U are, respectively, the preference of the assignment over the constraints in C, C_p, and C_u.

We recall that, given a soft constraint R, defined on variables X and Z its projection over X is defined by $\mu_P(d) = \sum_{a \in A_Z} \mu_R(d,a)$. In the following we show that properties $1, 2, 3$ hold for generic soft constraints where the set A of the semiring is *totally ordered*.

General Property 1: $\mu'(d) \geq_S \alpha$ if and only if, when $\pi_Z(a) > c(\alpha)$, then $\mu_R(d,a) \geq_S \alpha$.

Proof. Let us recall that $\mu'(d) \geq_S \alpha$ iff, by definition of μ', $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$.

($\Rightarrow$) Assume that $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$. Since $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \leq_S (\mu_R(d,a) + c(\pi_Z(a)))$ for all $a \in A_Z$, then $(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$ for all $a \in A_Z$. By hypothesis $\pi_Z(a) > c(\alpha)$. Since c is an order-reversing map such that $c(c(p)) = p$, we have $c(\pi_Z(a)) <_S \alpha$. Since A is totally ordered, for any two elements of the semiring we have $a + b = a$ or b, then $\mu_R(d,a) = (\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$.

($\Leftarrow$) Assume that, for every a such that $\pi_Z(a) > c(\alpha)$, we have $\mu_R(d,a) \geq_S \alpha$. Then, since $c(\pi_Z(a)) <_S \alpha$ and $\mu_R(d,a) \geq_S \alpha$ for such a, then $(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$. For all a such that $\pi_Z(a) \leq c(\alpha)$, we have $c(\pi_Z(a)) \geq_S \alpha$ and so $(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$. Thus for all a, $(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$. Therefore, since the inf among the elements of the semiring is one of these elements $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$, i.e., $\mu'(d) \geq_S \alpha$. $\square$

Notice that, if A is *partially ordered*, this property doesn't hold. However, two slightly weaker properties can be proved.

Weak general Property 1 ($\not\leq$): $\mu'(d) \not\leq_S \alpha$ if and only if, when $\pi_Z(a) \geq c(\alpha)$, then $\mu_R(d,a) \not\leq_S \alpha$.

Proof. Let us recall that $\mu'(d) \not\leq_S \alpha$ iff, by definition of μ', $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$.

($\Rightarrow$) Assume that $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$, where $\not\leq_S$ means that can be $>_S$ or incomparable $\bowtie_S$, then $(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$ for all $a \in A_Z$. In fact, if $(\mu_R(d,a) + c(\pi_Z(a))) \leq_S \alpha$ for some a then $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \leq_S \alpha$, that is a contradiction. By hypothesis $\pi_Z(a) \geq c(\alpha)$. Since c is an order-reversing map such that $c(c(p)) = p$, we have $c(\pi_Z(a)) \leq_S \alpha$ and so, since $(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$ for all $a \in A_Z$, $\mu_R(d,a) \not\leq_S \alpha$. In fact, suppose $\mu_R(d,a) \leq_S \alpha$. Then $(\mu_R(d,a) + c(\pi_Z(a))) \leq_S \alpha + \alpha$ for monotonicity of $+$, and since, for idempotency of $+$, $\alpha + \alpha = \alpha$, and so $(\mu_R(d,a) + c(\pi_Z(a))) \leq_S \alpha$, that is a contradiction since for hypothesis $\mu_R(d,a) \not\leq_S \alpha$.

($\Leftarrow$) Assume that, for every a such that $\pi_Z(a) \geq c(\alpha)$, we have $\mu_R(d,a) \not\leq_S \alpha$. Then, since $c(\pi_Z(a)) \leq_S \alpha$ and $\mu_R(d,a) \not\leq_S \alpha$ for such a, then $(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$. In fact, if $(\mu_R(d,a) + c(\pi_Z(a))) \leq_S \alpha$, then we have for monotonicity and idempotency of $+$, $\mu_R(d,a) \leq_S (\mu_R(d,a) + c(\pi_Z(a))) \leq_S \alpha + \alpha \leq_S \alpha$, that is a contradiction since $\mu_R(d,a) \not\leq_S \alpha$. Moreover, for all a such that $\pi_Z(a) < c(\alpha)$, we have $c(\pi_Z(a)) >_S \alpha$ and so $(\mu_R(d,a) + c(\pi_Z(a))) >_S \alpha$. Thus for all a, $(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$. Therefore, since every inf among the elements of the semiring is one of these elements, $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \not\leq_S \alpha$, i.e., $\mu'(d) \not\leq_S \alpha$. $\square$

Weak general Property 1 ($\not<$): $\mu'(d) \not<_S \alpha$ if and only if, when $\pi_Z(a) > c(\alpha)$, then $\mu_R(d,a) \not<_S \alpha$.

Proof. Very similar to the proof above.

Let us recall that $\mu'(d) \not<_S \alpha$ iff, by definition of μ', $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$.

($\Rightarrow$) Assume that $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$, where $\not<_S$ means that can be $\geq_S$ or incomparable $\bowtie_S$, then $(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$ for all $a \in A_Z$. In fact, if $(\mu_R(d,a) + c(\pi_Z(a))) <_S \alpha$ for some a then $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) <_S \alpha$, that is a contradiction. By hypothesis $\pi_Z(a) > c(\alpha)$. Since c is an order-reversing map such that $c(c(p)) = p$, we have $c(\pi_Z(a)) <_S \alpha$ and so, since $(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$, $\mu_R(d,a) \not<_S \alpha$. In fact, suppose $\mu_R(d,a) <_S \alpha$. Then $(\mu_R(d,a) + c(\pi_Z(a))) <_S \alpha + \alpha$ for monotonicity of $+$, and for idempotency of $+$, $\alpha + \alpha = \alpha$ and so $(\mu_R(d,a) + c(\pi_Z(a))) <_S \alpha$, that is a contradiction since for hypothesis $\mu_R(d,a) \not<_S \alpha$.

($\Leftarrow$) Assume that, for every a such that $\pi_Z(a) > c(\alpha)$, we have $\mu_R(d,a) \not<_S \alpha$. Then, since $c(\pi_Z(a)) <_S \alpha$ and $\mu_R(d,a) \not<_S \alpha$ for such a, then $(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$. In fact, if $(\mu_R(d,a) + c(\pi_Z(a))) <_S \alpha$, then we have for monotonicity and idempotency of $+$, $\mu_R(d,a) <_S (\mu_R(d,a) + c(\pi_Z(a))) <_S \alpha + \alpha = \alpha$, that is a contradiction since $\mu_R(d,a) \not<_S \alpha$. Moreover, for all a such that $\pi_Z(a) \leq c(\alpha)$, we have $c(\pi_Z(a)) \geq_S \alpha$ and so $(\mu_R(d,a) + c(\pi_Z(a))) \geq_S \alpha$. Thus for all a, $(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$. Therefore, since every inf among the elements of the semiring is one of these elements, $inf_{a \in A_Z}(\mu_R(d,a) + c(\pi_Z(a))) \not<_S \alpha$, i.e., $\mu'(d) \not<_S \alpha$. □

In the following we will show that properties $2, 3$ hold for problems with generic soft constraints and uncertainty described by π_Z that can be a possibilistic or a probabilistic distribution.

General Property 2: Given π_Z and two preference functions μ_1 and μ_2 such that $\mu_1(d,a) \leq_S \mu_2(d,a)$ for all a assignments to Z, then $\mu'_1(d) \leq_S \mu'_2(d)$, where μ'_i is the preference function obtained considering μ_i, i=1,2.

Proof. We recall that $\mu'_1(d) = inf_{a \in A_z}(\mu_1(d,a) + c(\pi_Z(a)))$ and $\mu'_2(d) = inf_{a \in A_z}(\mu_2(d,a) + c(\pi_Z(a)))$. Since $\mu_1(d,a) \leq_S \mu_2(d,a)$, $(\mu_1(d,a) + c(\pi_Z(a))) \leq_S (\mu_2(d,a) + c(\pi_Z(a)))$ by monotonicity of $+$. Then we have $inf_{a \in A_z}(\mu_1(d,a) + c(\pi_Z(a))) \leq_S inf_{a \in A_z}(\mu_2(d,a) + c(\pi_Z(a)))$, i.e. $D \leq_S E$ if we call $D = inf_{a \in A_z} D_a$, where $D_a = (\mu_1(d,a) + c(\pi_Z(a)))$ and $E = inf_{a \in A_z} E_a$, where $E_a = (\mu_2(d,a) + c(\pi_Z(a)))$. If we suppose that $D >_S E$, then since $E \geq_S D_a$ for some a, $D >_S E \geq_S D_a$ for some a, that is a contradiction since $D \leq_S D_a$, for all a since it is the inf. Thus $\mu'_1(d) \leq_S \mu'_2(d)$. □

General Property 3: Given a preference function μ and two distributions π_1 and π_2 on Z, such that $\pi_1(a) \geq \pi_2(a)$ for all a, then $\mu'_1(d) \leq_S \mu'_2(d)$, where μ'_i is the preference function obtained considering π_i, i=1,2.

Proof. We recall that $\mu'_1(d) = inf_{a \in A_Z}(\mu(d,a) + c(\pi_1(a)))$, $\mu'_2(d) = inf_{a \in A_Z}(\mu(d,a) + c(\pi_2(a)))$ and that c is an order-reversing map, that is, if $\pi_1(a) \geq \pi_2(a)$, then if $c(\pi_1(a)) \leq_S c(\pi_2(a))$. By monotonicity of $+$, we can conclude that $(\mu(d,a) + c(\pi_1(a))) \leq_S (\mu(d,a) + c(\pi_2(a)))$. For the same reasoning explained in the proof above $inf_{a \in A_Z}(\mu(d,a) + c(\pi_1(a))) \leq_S inf_{a \in A_Z}(\mu(d,a) + c(\pi_2(a)))$. Thus $\mu'_1(d) \leq_S \mu'_2(d)$. $\Box$

Notice that, if A is *partially ordered*, general property 2 and 3 assume a slightly weaker form. The only change in the statement is that $\mu'_1(d) \leq_S \mu'_2(d)$ is replaced with $\mu'_1(d) \not>_S \mu'_2(d)$. The proofs of these properties are equal to the corresponding ones in the total order case, where the only difference is that you have replaced $inf_{a \in A_Z}(\mu(d,a) + c(\pi_1(a))) \leq_S inf_{a \in A_Z}(\mu(d,a) + c(\pi_2(a)))$ with $inf_{a \in A_Z}(\mu(d,a) + c(\pi_1(a))) \not>_S inf_{a \in A_Z}(\mu(d,a) + c(\pi_2(a)))$.

Since these properties hold in general, then we can safely and effectively handle problems with many kinds of soft constraints as well as uncertain possibilistic or probabilistic variables.

The generic semantics are defined like those ones in [9], except that $\leq$ is replaced by $\leq_S$.

7 Uncertain Probabilistic CSPs

In several real-life scenarios, fuzzy constraints are not the ideal setting. In fact, they suffer for the well-known drawning effect which makes solutions with the same minimum preference but very different higher preferences not distinguished.

To avoid this problem, it can be useful to combine preferences by multiplying them rather than taking their minimum value. This is what happens in probabilistic CSPs (PCSPs) [8]. In a PCSP, variable assignments have associated preferences, and the goal is to maximize the product of all such preferences. Therefore, the semiring to be used is $S = \{[0,1], max, \times, 0, 1\}$.

To make sure that the desired properties 1, 2, and 3 hold in this setting, we just need to check whether the assumptions we made are met: $[0,1] \subseteq A$ and c is an bijection from [0,1] to [0,1] such that, for each $a_1, a_2 \in [0,1]$, $a_1 \leq a_2$ if and only if $c(a_1) \geq_S c(a_2)$; moreover, $c(c(a)) = a$ for all a.

The first assumption is trivially true since $A = [0,1]$. As for the second one, we consider $c(x) = 1 - x$ for all x, which satisfies the order-reversing property. In fact, given $a_1, a_2 \in [0,1]$, with $a_1 \leq a_2$ we have $1 - a_1 \geq_S 1 - a_2$, since $max(1 - a_1, 1 - a_2) = 1 - a_1$.

In this setting, we have

$$\mu'(d) = min_{a \in A_Z} max(\mu_R(d,a), 1 - \pi_Z(a)).$$

Figure 3 shows the result of applying **SP** to the problem in Figure 1, seen as an uncertain PCSP. In this case, the algorithm is instantiated with $\times_S = \times$, $+_S = max$, and $c(p) = 1 - p$. In particular, Figure 3 (a) shows the resulting probabilistic CSP obtained by the algorithm, while Figure 3 (b) shows all complete assignments, together with the associated pair.

8 Uncertain Weighted CSPs

In several situations where neither fuzzy nor probabilistic constraints are ideal, weighted constraints can be useful to model preferences. For example, this is usually done when dealing with costs which are naturally combined by a sum.

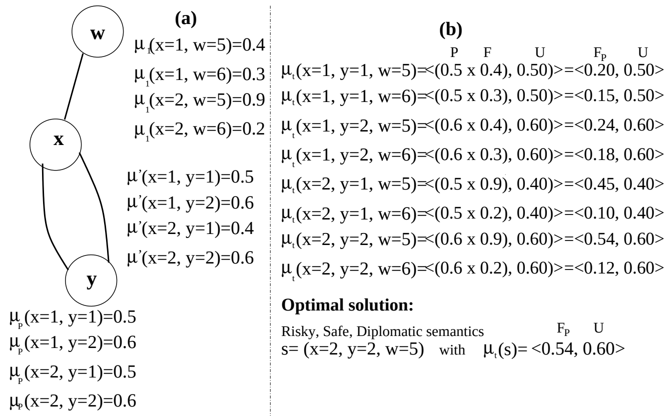


Figure3. Result of algorithm **SP** for the uncertain soft CSP in Figure 1, seen as an uncertain PCSP.

In this setting, preferences are penalties (or costs) to be added, and the best solutions are those with the smallest preference. Thus operators $+$ and min are the instantiation of the operators $\times$ and $+$ of the general case. Therefore, the semiring to be used is $S = \{\mathcal{R}^+, min, +, +\infty, 0\}$.

The three desired property hold if we choose c as the identity map. In fact, $[0, 1] \subseteq \mathcal{R}^+$. Moreover, c is an order-reversing map w.r.t semiring S. In fact, given $a_1, a_2 \in [0, 1]$ such that $a_1 \leq a_2$, we have $a_1 \geq_S a_2$ since $min(a_1, a_2) = a_1$.

The instantiated formula for μ' is then

$$\mu'(d) = max_{a \in A_z} min(\mu_R(d, a), \pi_Z(a))).$$

Figure 4 shows how algorithm **SP** works on weighted CSP with possibilitic uncertainty, as the one in Figure 1 (where preferences are interpreted as costs). Figure 3 (a) shows the resulting weighted CSP, while Figure 3 (b) shows all complete assignments, together with the associated pair. In this problem the optimal solution obtained using the *Risky* semantics is different from the one obtained using the *Safe* semantics, whereas *Diplomatic* considers them both optimal.

9 Probabilistic or possibilistic uncertainty

In the previous sections we have considered different classes of soft constraints with uncertainty described via a possibilitic or a probabilistic distribution. Here we consider the case where a possibilistic distribution is available only for some uncertain variables, while for others we have a probabilistic distribution.

If there are no constraints connecting both possibilistic and probabilistic variables, then we can apply the same algorithm as above, and the three properties hold. However, consider the situation where a constraint involves uncertain variables over which a possibility distribution is available, and also uncertain

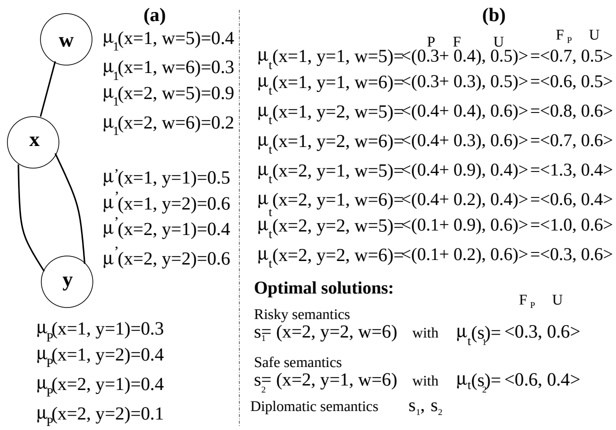


Figure4. Result of algorithm **SP** for the uncertain soft CSP in Figure 1, seen as an uncertain weighted CSP.

variables over which we have a probabilistic distribution. In this case, we could get to the usual setting by replacing possibilities with probabilities, or vice versa. In [4] it is presented a way to do this. Thus we could use such a method to obtain only one kind of distribution and then use our approach.

However, transforming a probability into a possibility distribution we loose information, and solutions have a lower robustness. In fact, using property 3, it is possible to see that, if we use possibilities, which are higher than probabilities, we get a smaller robustness value. Thus we can say that the robustness value obtained in this way is a lower bound to the certainty that the values of the decision variables are compatible with the uncertain variables. On the other hand, if we transform possibilities into probabilities, we get smaller values, and thus by property 3 a higher robustness value, which can be seen as an upper bound to the certainty degree of a solution.

Thus, in the presence of constraints involving both possibilistic and probabilistic uncertainty, we can still use the same approach except that we work with upper and lower bounds for the robustness of the solutions.

To be able to use the same three semantics defined above, we need to define an ordering over intervals, since now the robustness is described by an interval. A possible ordering defines two intervals incomparable if one is strictly contained in the other one, and both lower and upper bounds are different. For example, $\langle 0.2, 0.5\rangle$ is incomparable with $\langle 0.1, 0.6\rangle$ but it is not incomparable with $\langle 0.1, 0.4\rangle$, nor with $\langle 0.2, 0.6\rangle$. In all other cases, the two intervals are ordered. More precisely, $\langle l_1, u_1\rangle$ is better than $\langle l_2, u_2\rangle$ if they are different and $l_1 \geq l_2$ and $u_1 \geq u_2$.

Notice that this ordering is partial over the robustness values, while before we had a total order. This yields more incomparability in the ordering over solutions induced by each of the semantics. Thus also the Risky and Safe semantics will induce partial orders.

10 Future work

We plan to develop a solver that can handle problems with several classes of soft constraints, together with uncertainty expressed via possibility or probability distributions. The solver will be able to generate orderings according the three semantics proposed in this paper as well as others that we will define by following different optimistic or pessimistic approaches.

References

1. L. Amgoud, H. Prade. Using arguments for making decisions: a possibilistic logic approach. Proc. Uncertainty in Artificial Intelligence, 10-17, 2004.
2. S. Bistarelli, U. Montanari and F. Rossi. Semiring-based Constraint Solving and Optimization. Journal of the ACM, Vol.44, n.2, March 1997.
3. D. Dubois, H. Fargier, H. Prade. Possibility theory in constraint satisfaction problems: handling priority, preference and uncertainty. Applied Intelligence, 6, 287-309, 1996.
4. D. Dubois, H. Prade. Possibility theory: qualitative and quantitative aspects. Handbook of Defeasible Reasoning and Uncertainty Management Systems, Vol.1, 169-226, Kluwer, 1998.
5. D. Dubois, H. Prade. Fuzzy sets and Systems - Theory and Applications. Academic Press, 1980.
6. D. Dubois, H. Prade. Possibility theory. Plenum Press, 1988.
7. D. Dubois, H. Prade. Fundamentals of Fuzzy Sets. Handbooks of Fuzzy Sets Series, Kluwer, 2000.
8. H. Fargier, J. Lang, T. Schiex. Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. Proc. AAAI'96, The AAAI Press, 1996.
9. M.S. Pini, F. Rossi, B. Venable. Possibility theory for reasoning about uncertain soft constraints. Proc. of ECSQARU 2005, LNCS Springer, 2005.
10. L.A. Zadeh. Fuzzy sets as a basis for the theory of possibility. Fuzzy Sets and Systems, 13-28, 1978.

Graph Invariants as Necessary Conditions for Global Constraints

Nicolas Beldiceanu[1], Mats Carlsson[2], Jean-Xavier Rampon[3], and Charlotte Truchet[3]

[1] LINA FRE CNRS 2729, École des Mines de Nantes, 44307 Nantes Cedex 3, France.
Nicolas.Beldiceanu@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden.
Mats.Carlsson@sics.se
[3] LINA FRE CNRS 2729, 2 rue de la Houssinière, BP-92208, 44322 Nantes Cedex 3, France.
{Jean-Xavier.Rampon,Charlotte.Truchet}@lina.univ-nantes.fr

Abstract. This article presents a database of about 200 graph invariants for deriving systematically necessary conditions from the graph properties based representation of global constraints. This scheme is based on invariants on the graph characteristics used in the description of a global constraint. A SICStus Prolog implementation based on arithmetic and logical constraints as well as on indexicals is available.

1 Introduction

Adding necessary conditions to a constraint program has been recognized in the early time of constraint programming [1] as a key point in order to enhance efficiency. However this was usually done manually after a careful analysis of the problem under consideration or by identifying typical constraints patterns [2]. Beldiceanu presented in [3] a systematic description of global constraints in terms of graph properties: among the 227 constraints of the catalog of global constraints [3], about 200 constraints are described as a conjunction of graph properties where each graph property has the form $P\ op\ V$, where P is a graph characteristic, *op* is a comparison operator in $\{\leq, \geq, =, \neq\}$, and V a variable that ranges over a finite set of integers (a *domain variable*). Within this context, this article presents a database of graph invariants: given a specification of a constraint C in terms of graph properties, we can automatically extract, from that database, graph invariants that mention the graph characteristics used in the specification of C, and post these invariants as necessary conditions for the feasibility of C.

Example 1. Consider the `nvalue(`$N, \{x_1, ..., x_m\}$`)` constraint [4], where $N, x_1, ..., x_m$ are domain variables. The `nvalue` constraint holds iff the number of distinct values assigned to the variables in $\mathcal{X} = \{x_1, ..., x_m\}$ is equal to N. It can been seen as enforcing the following graph property: the number of strongly connected components of the *intersection graph* $G(\mathcal{X}, E)$, where $E = \{x_i \in \mathcal{X}, x_j \in \mathcal{X} : x_i = x_j\}$, is equal to N. From Bessière et al. [5] we have the necessary condition $\mathbf{NSCC} \geq \left\lceil \frac{\mathbf{NVERTEX}^2}{\mathbf{NARC}} \right\rceil$ (see Turán [6]) relating the number of arcs $\mathbf{NARC}$, the number of vertices $\mathbf{NVERTEX}$ and the number of strongly connected components $\mathbf{NSCC}$ of the *intersection graph*.

Using graph invariants is especially useful when a global constraint mentions more than one graph property in its description. In this context, these graph properties involve several graph characteristics that cannot vary independently.

Example 2. Consider again the `nvalue` constraint introduced in Example 1, and assume we want to put a restriction on the minimum and the maximum number of occurrences (respectively denoted by $\underline{occ}$ and by $\overline{occ}$) of each value that is effectively used. In terms of the intersection graph, this can be interpreted as putting a restriction on the number of vertices of its strongly connected components. Let $\mathbf{MIN\_NSCC}$ and $\mathbf{MAX\_NSCC}$ respectively denote the number of vertices of the smallest and the largest strongly connected components of the intersection graph. Our initial constraint on the minimum and maximum number of occurrences is now expressed by $\mathbf{MIN\_NSCC} \geq \underline{occ}$ and $\mathbf{MAX\_NSCC} \leq \overline{occ}$. We have recast our original balanced assignment problem to the search of a digraph on which we restrict its number of vertices $\mathbf{NVERTEX}$[1], its number of strongly connected components $\mathbf{NSCC}$, and the sizes $\mathbf{MIN\_NSCC}$ and $\mathbf{MAX\_NSCC}$ of its smallest and largest strongly connected components. By querying our database of invariants in order to extract those graph invariants that only mention the four graph characteristics $\mathbf{NVERTEX}$, $\mathbf{NSCC}$, $\mathbf{MIN\_NSCC}$ and $\mathbf{MAX\_NSCC}$ we get the following invariants $\mathbf{NVERTEX} \leq \max(0, \mathbf{NSCC} - 1) \cdot \mathbf{MAX\_NSCC} + \mathbf{MIN\_NSCC}$ and $\mathbf{NVERTEX} \geq \max(0, \mathbf{NSCC} - 1) \cdot \mathbf{MIN\_NSCC} + \mathbf{MAX\_NSCC}$, which are necessary conditions for the balanced assignment constraint.

Section 2 recalls the graph-based representation of global constraints. Section 3 introduces graph invariants, while Section 4 presents the database of graph invariants. The database and its 200 graph invariants and their corresponding proofs is available in Chapter 3 of [3]. Finally, Section 4 provides an evaluation of the approach on two constraints, which mention various graph characteristics.

2 Graph-Based Representation of Global Constraint

This section summarizes the representation of global constraints as graph properties in [3] and illustrates this framework on the `group` [7] and the `change_continuity` [3] constraints, which will be used throughout this article. They both correspond to timetabling constraints which allow for expressing conditions on sliding sequences of consecutive working days of a given person.

The graph-based representation. A global constraint C is represented as an initial digraph $G_i = (\mathcal{X}_i, E_i)$: to each vertex in $\mathcal{X}_i$ corresponds a variable involved in C, while to each arc e in E_i corresponds a binary constraint involving the variables at both extremities of e. To generate G_i from the parameters of C, the set of arc generators described in [3] is used. Figure 1 illustrates the most commonly used arc generators by depicting the initial digraph generated from a sequence of four vertices. When all variables of C are fixed, we remove from G_i all binary constraints that do not hold as well as isolated vertices, i.e., vertices that are not extremities of an arc. This final digraph is denoted by G_f. C is equivalent to a conjunction of graph properties which should be satisfied by

[1] In fact, $\mathbf{NVERTEX}$ is fixed to the number of variables of the `nvalue` constraint.

G_f. Within the global constraint catalog [3], commonly used graph characteristics on the final digraph G_f are:

– $\mathbf{NARC}$ and $\mathbf{NVERTEX}$ denote the number of arcs and vertices,
– $\mathbf{NCC}$ and $\mathbf{NSCC}$ denote the number of connected and strongly connected components,
– $\mathbf{MIN\_NCC}$ and $\mathbf{MAX\_NCC}$ (resp. $\mathbf{MIN\_NSCC}$ and $\mathbf{MAX\_NSCC}$) respectively denote the number of vertices of the smallest and the largest connected components (resp. the strongly connected components).

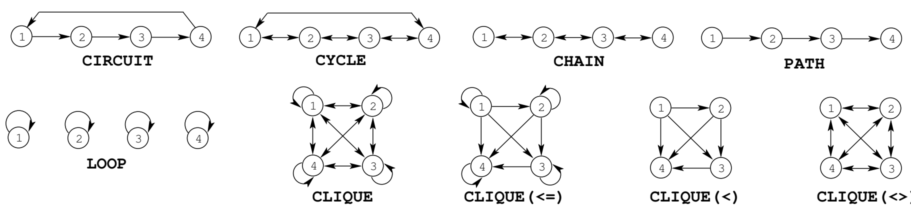


Fig. 1. Examples of arc generators

Illustrative examples of the graph-based representation. We now define the `group` and the `change_continuity` constraints and present their links with the graph-based description. Since they respectively use 6 and 8 graph characteristics these constraints can potentially benefit from the use of graph invariants.

Example 3. The first six parameters of the `group(NGROUP,MIN_SIZE,MAX_SIZE, MIN_DIST,MAX_DIST,NVAL,VARIABLES,VALUES)` constraint are domain variables, while `VARIABLES` is a sequence of domain variables and `VALUES` a finite set of integers. Let n denote the number of variables of the sequence `VARIABLES`. Let $X_i, X_{i+1}, \ldots, X_j$ $(1 \leq i \leq j \leq n)$ be consecutive variables of the sequence `VARIABLES` such that all the following conditions simultaneously apply: (1) All variables $X_i, \ldots, X_j$ take their value in the set of values `VALUES`, (2) $i = 1$ or X_{i-1} does not take a value in `VALUES`, (3) $j = n$ or X_{j+1} does not take a value in `VALUES`. We call such a set of variables a *group*. The constraint `group` is fulfilled if all the following conditions hold:

– There are exactly `NGROUP` groups of variables,
– `MIN_SIZE` and `MAX_SIZE` are the number of variables of the smallest and largest group,
– `MIN_DIST` and `MAX_DIST` are the minimum and maximum number of variables between two consecutive groups or between one border and one group,
– `NVAL` is the number of variables that take their value in the set of values `VALUES`.

group$(2, 2, 4, 1, 2, 6, \langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle, \{1, 2, 3\})$ holds since the sequence $\langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle$ contains 2 groups $\langle 1, 3\rangle$ and $\langle 2, 2, 2, 3\rangle$ of non-zero values of size 2 and 4, 2 groups $\langle 0, 0\rangle$ and $\langle 0\rangle$ of zeros, and 6 non-zero values. The graph-based description of the `group` constraint uses two graph constraints which respectively mention the graph properties $\mathbf{NCC}$ = `NGROUP`, $\mathbf{MIN\_NCC}$ = `MIN_SIZE`, $\mathbf{MAX\_NCC}$ = `MAX_SIZE`, $\mathbf{NVERTEX}$ = `NVAL` and $\mathbf{MIN\_NCC}$ = `MIN_DIST`, $\mathbf{MAX\_NCC}$ = `MAX_DIST`. The leftmost part of Figure 2 depicts the initial graph of well as the two final graphs associated to the two graph constraints of the example given for the `group` constraint.

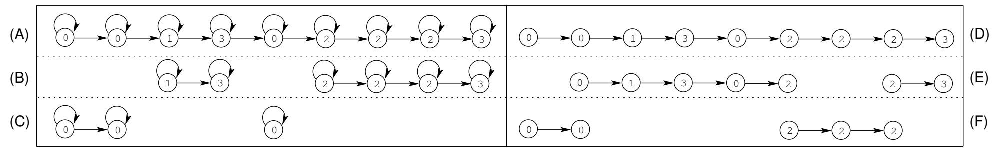


Fig. 2. Initial (A) and final graphs (B,C) of $\texttt{group}(2, 2, 4, 1, 2, 6, \langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle, \{1, 2, 3\})$. Initial (D) and final graphs (E,F) of $\texttt{change\_continuity}(2, 2, 2, 5, 2, 3, 5, 3, \langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle, \neq)$.

Example 4. The first eight parameters of the change_continuity(NB_PERIOD_CHANGE, NB_PERIOD_CONTINUITY, MIN_SIZE_CHANGE, MAX_SIZE_CHANGE, MIN_SIZE_CONTINUITY, MAX_SIZE_CONTINUITY, NB_CHANGE, NB_CONTINUITY, VARIABLES, CTR) constraint are domain variables, while VARIABLES is a sequence of domain variables and CTR a binary constraint in $\{=, \neq, \leq, >, \geq, <\}$. A *change* (resp. *continuity*) is defined by the fact that constraint CTR holds (resp. does not hold) between two consecutive variables of the sequence VARIABLES. Let n denote the number of variables of the sequence VARIABLES, and let $X_i, X_{i+1}, \ldots, X_j (1 \leq i < j \leq n)$ be consecutive variables of the sequence VARIABLES. $X_i, X_{i+1}, \ldots, X_j$ corresponds to a *period of change* if X_k CTR X_{k+1} holds for all $k \in [i, j-1]$, and if $i = 1$ or X_{i-1} CTR X_i does not hold, and if $j = n-1$ or X_j CTR X_{j+1} does not hold. A *period of continuity* is defined in a similar way by considering the negation of CTR . The constraint change_continuity holds if and only if:

- NB_PERIOD_CHANGE and NB_PERIOD_CONTINUITY are respectively equal to the number of periods of change and of continuity,
- MIN_SIZE_CHANGE and MAX_SIZE_CHANGE are respectively equal to the number of variables of the smallest and largest period of change,
- MIN_SIZE_CONTINUITY and MAX_SIZE_CONTINUITY are respectively equal to the number of variables of the smallest and largest period of continuity,
- NB_CHANGE and NB_CONTINUITY are respectively equal to the total number of changes and continuities.

$\texttt{change\_continuity}(2, 2, 2, 5, 2, 3, 5, 3, \langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle, \neq)$ holds since the sequence $\langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle$ contains 2 periods of changes $\langle 0, 1, 3, 0, 2\rangle$ and $\langle 2, 3\rangle$ of minimum and maximum size 2 and 5, 2 periods of continuities $\langle 0, 0\rangle$ and $\langle 2, 2, 2\rangle$ of minimum and maximum size 2 and 3. Finally, the total number of changes and continuities are respectively equal to 5 and 3. The graph-based description of the change_continuity(NB_PERIOD_CHANGE, NB_PERIOD_CONTINUITY, MIN_SIZE_CHANGE, MAX_SIZE_CHANGE, MIN_SIZE_CONTINUITY, MAX_SIZE_CONTINUITY, NB_CHANGE, NB_CONTINUITY, VARIABLES, CTR) constraint uses two graph constraints which respectively mention the graph properties **NCC** = NB_PERIOD_CHANGE, **MIN_NCC** = MIN_SIZE_CHANGE, **MAX_NCC** = MAX_SIZE_CHANGE, **NARC** = NB_CHANGE and **NCC** = NB_PERIOD_CONTINUITY, **MIN_NCC** = MIN_SIZE_CONTINUITY, **MAX_NCC** = MAX_SIZE_CONTINUITY, **NARC** = NB_CONTINUITY. The rightmost part of Figure 2 depicts the initial graph of well as the two final graphs associated to the two graph constraints of the example given for the change_continuity constraint.

3 Graph Invariants

Within the scope of the graph-based description this section introduces implied constraints which are systematically linked to the description of a global constraint:

- We then describe the different contexts where graph invariants can be used.
- Finally, we show how to get sharper graph invariants by taking advantage of the structure of the global constraint under consideration.

Since no final digraph contains isolated vertices, the database of graph invariants considers digraphs for which each vertex has at least one arc.

Context for Using Graph Invariants. They can be used in the following contexts:

- Quite often, it happens that one wants the final digraph to satisfy more than one graph property. This was illustrated by the balanced assignment constraint (see Example 2) as well as by the `group` and `change_continuity` constraints. In this context, these graph properties involve several graph characteristics which cannot vary independently.
- Even if the description of a global constraint involves one single graph characteristic $\mathbf{C}$, we can introduce the number of vertices, $\mathbf{NVERTEX}$, and the number of arcs, $\mathbf{NARC}$, of the final digraph. In this context, we can take advantage of graph invariants linking $\mathbf{C}$, $\mathbf{NARC}$ and $\mathbf{NVERTEX}$. This is in fact what was done for the `nvalue` constraint in Example 1.
- It also happens that we enforce two graph constraints $\mathcal{GC}_1$ and $\mathcal{GC}_2$, which have the same initial digraph $\mathcal{G}$. In this context we consider the following situations:
  - Each arc of $\mathcal{G}$ belongs to one of the final digraphs associated to $\mathcal{GC}_1$ or to $\mathcal{GC}_2$ (but not to both). An example of such global constraints is the `change_continuity` constraint depicted by Example 4.
  - Each vertex of $\mathcal{G}$ belongs to one of the final digraphs associated to $\mathcal{GC}_1$ or to $\mathcal{GC}_2$ (but not to both). An example of such global constraint is the `group` constraint depicted by Example 3.

  In these situations the graph properties associated to the two graph constraints are not independent. This will be illustrated by Example 12.

Graph Classes. By definition, a graph invariant has to hold for any final digraph. For instance, we have the graph invariant $\mathbf{NARC} \leq \mathbf{NVERTEX}^2$, which relates the number of arcs and the number of vertices of any digraph. This invariant is sharp since the equality is reached for a clique. However, by considering the structure of a final digraph, we can get sharper invariants. For instance, if our final digraph is a subset of an elementary path (e.g. we use the $PATH$ arc generator depicted by Figure 1) we have that $\mathbf{NARC} \leq \mathbf{NVERTEX} - 1$, which is a tighter bound of the maximum number of arcs since $\mathbf{NVERTEX} - 1 < \mathbf{NVERTEX}^2$. For this reason, we consider recurring graph classes that show up for different global constraints. For a given global constraint, a graph class specifies a general property which holds on all its final digraphs. In addition, we also consider graph constraints such that their final digraph is a subset of the digraph generated by the arc generators depicted by Figure 1.

Example 5. We provide typical examples of graph classes and, for each of them, we point to some global constraints that fit in that class:

- **`acyclic`**: graph constraint for which the final digraph doesn't have any circuit (e.g. `change` [7], `change_continuity` [3], `common` [3]).

- **apartition**: constraint defined by two graph constraints having the same initial digraph, where each arc of the initial digraph belongs to one of the final digraphs (but not to both) (e.g. `change_continuity` [3]).
- **bipartite**: graph constraint for which the final digraph is bipartite (e.g. `alldifferent_on_intersection` [3], `common` [3]).
- **consecutive_loops_are_connected**: denotes the fact that the graph constraints of a global constraint use only the $PATH$ and the $LOOP$ arc generators and that their final digraphs do not contain consecutive vertices which have a loop and which are not connected together by an arc (e.g. `group` [3]).
- **equivalence**: graph constraint for which the final digraph is reflexive, symmetric and transitive (e.g. `balance` [3], `nvalue` [5]).
- **no_loop**: graph constraint for which the final digraph doesn't have any loop (e.g. `change_continuity` [3], `common` [3]).
- **one_succ**: graph constraint for which all the vertices of the initial digraph belong to the final digraph and for which all vertices of the final digraph have exactly one successor (e.g. `alldifferent` [8], `cycle` [9], `tree` [10]).
- **symmetric**: graph constraint for which the final digraph is symmetric (e.g. `connect_points` [3]).
- **vpartition**: constraint defined by two graph constraints having the same initial digraph, where each vertex of the initial digraph belongs to one of the final digraphs (but not to both) (e.g. `group` [3]).

4 The Database of Graph Invariants

This section introduces the database of graph invariants we have built so far. It first provides a taxonomy of graph invariants and discusses their implementation. It then presents the organisation of the database. Finally, it explains how to use the database in order to automatically extract the relevant invariants for a given global constraint.

Taxonomy of Graph Invariants. Within the database of graph invariants we currently have seven categories of graph invariants. These categories steem from the structure of the formulae associated to the invariants.

I1. Invariants involving *one single* graph characteristics C, restricting the initial set of possible values of C.

Example 6. When the final digraph does not contain any loops, we have that $2 \cdot \mathbf{NCC} \leq \mathbf{NVERTEX}_{\mathtt{INITIAL}}$, where $\mathbf{NVERTEX}_{\mathtt{INITIAL}}$ is the number of vertices of the initial digraph and where $\mathbf{NCC}$ is the number of connected components of the final digraph. This invariant restricts the initial domain of $\mathbf{NCC}$ to $[0, \lfloor \frac{\mathbf{NVERTEX}_{\mathtt{INITIAL}}}{2} \rfloor]$.

I2. Invariants characterizing the *lower bound* (resp. *upper bound*) of a given graph characteristics C in terms of other graph characteristics $C_1, \ldots, C_n (n > 1, C_i \neq C)$. They are defined as an inequality of the form $C \geq f(C_1, \ldots, C_n)$ (resp. $C \leq f(C_1, \ldots, C_n)$), where $f(C_1, \ldots, C_n)$ is a formula involving the graph characteristics $C_1, \ldots, C_n$.

Example 7. As illustrated by Figure 3, the invariant $\mathbf{NARC} \geq \mathbf{NVERTEX} - \left\lfloor \frac{\mathbf{NSCC}-1}{2} \right\rfloor$ can be interpreted as the minimum number of arcs $\mathbf{NARC}$ of a digraph according to a fixed number of vertices $\mathbf{NVERTEX}$ and a fixed number of strongly connected components $\mathbf{NSCC}$.

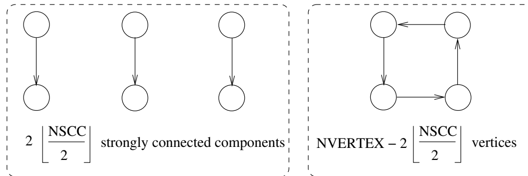


Fig. 3. A digraph which achieves the minimum number of arcs according to a fixed number of strongly connected components as well as to a fixed number of vertices ($\mathbf{NSCC} = 7, \mathbf{NVERTEX} = 10, \mathbf{NARC} = 10 - \left\lfloor \frac{7}{2} \right\rfloor = 7$)

I3. Invariants defining, for a given graph characteristics C, a *forbidden interval* of values of the form $[f_1(C_1, \ldots, C_n), f_2(C_{n+1}, \ldots, C_m)]$, where $f_1(C_1, \ldots, C_n)$ and $f_2(C_{n+1}, \ldots, C_m)$ are formulae involving graph characteristics distinct from C. These invariants usually come from a disjunction of the form $C \leq f_1(C_1, \ldots, C_n) - 1 \vee C \geq f_2(C_{n+1}, \ldots, C_m) + 1$.

Example 8. Consider the invariant $\mathbf{MIN\_NCC} \notin \left[\left\lfloor \frac{\mathbf{NVERTEX}}{2} \right\rfloor + 1, \mathbf{NVERTEX} - 1\right]$, which specifies that the number of vertices $\mathbf{MIN\_NCC}$ of the smallest connected component of a digraph does not belong to an interval defined according to the number of vertices $\mathbf{NVERTEX}$. This invariant stems from the following disjunction:

– On the one hand, if the digraph contains no more than one connected component, we have that $\mathbf{MIN\_NCC} \geq \mathbf{NVERTEX}$,
– On the other hand, if the digraph contains at least two connected components, we have that $\mathbf{MIN\_NCC} + \mathbf{MIN\_NCC} \leq \mathbf{NVERTEX}$.

I4. Invariants of the form $C \leq \max(f_1(C_1, \ldots, C_n), f_2(C_{n+1}, \ldots, C_m))$, where C is a graph characteristics and $f_1(C_1, \ldots, C_n)$ and $f_2(C_{n+1}, \ldots, C_m)$ are formulae involving graph characteristics distinct from C. These invariants usually come from a disjunction of two invariants $C \leq f_1(C_1, \ldots, C_n) \vee C \leq f_2(C_{n+1}, \ldots, C_m)$.

Example 9. Consider the invariant $\mathbf{MAX\_NCC} \leq \max(\mathbf{NVERTEX} - \mathbf{MIN\_NCC}, \mathbf{MIN\_NCC})$, which restricts the maximum number of vertices $\mathbf{MAX\_NCC}$ of the largest connected component according to the number of vertices in the smallest connected component and to the number of vertices $\mathbf{NVERTEX}$. This invariant stems from the following disjunction:

– On the one hand, if the digraph contains no more than one connected component, we have that $\mathbf{MAX\_NCC} \leq \mathbf{MIN\_NCC}$,
– On the other hand, if the digraph contains at least two connected components, we have that $\mathbf{NVERTEX} \geq \mathbf{MIN\_NCC} + \mathbf{MAX\_NCC}$ (i.e. $\mathbf{MAX\_NCC} \leq \mathbf{NVERTEX} - \mathbf{MIN\_NCC}$).

I5. Invariants described by an implication between two conditions. These invariants have the form $\mathcal{C}ond_1 \Rightarrow \mathcal{C}ond_2$ where $\mathcal{C}ond_1$ is a condition involving one or two graph characteristics, and where $\mathcal{C}ond_2$ is either a condition involving one or two graph characteristics, either an invariant of type **I2** or **I3**.

Example 10. As an example, consider the invariant $\mathbf{MIN\_NCC} \neq \mathbf{MAX\_NCC} \Rightarrow \mathbf{NCC} \geq 2$, which depicts the fact that, if the number of vertices of the smallest connected component is not equal to the size of the largest connected component, the number of connected components is at least 2.

I6. Invariants depicted by an equivalence between two given conditions where each condition involves one single graph characteristics.

Example 11. $\mathbf{MAX\_NCC} = 0 \Leftrightarrow \mathbf{MIN\_NCC} = 0$ is an instance of such invariant.

I7. Invariants involving graph characteristics coming from more than one graph constraint.

Example 12. `apartition` $\wedge$ $\mathbf{arc\_gen} = PATH : |\mathbf{NCC}_1 - \mathbf{NCC}_2| \leq 1$ [2] is an invariant which can be applied when:

- As specified by `apartition`, a global constraint is defined by two graph constraints having the same initial digraph, where each arc of the initial digraph belongs to one of the final digraphs (but not to both),
- All the graph constraints of a global constraint use only the arc generator $PATH$.

This is in fact the situation of the `change_continuity` constraint introduced in Example 4: in this context, this invariant enforces the number of groups of changes `NB_CHANGE` and the number of groups of continuities `NB_CONTINUITY` to differ by at most 1.

Each graph invariant has a precondition which defines its applicability. The precondition consists of an, possibly empty, conjunction of elementary conditions which characterize the graph class for which it can be applied. An elementary condition is either one of the keywords `acyclic`, `bipartite`, `no_loop`, `one_succ`, `symmetric`, `equivalence`, `apartition`, `vpartition`, `consecutive_loops_are_connected` characterizing a specific graph class which was previously introduced, either an expression of the form `arc_gen` $=$ *arc generator*, where *arc generator* is an arc generator used for generating the arcs of the initial digraph.

Example 13. Consider the graph invariants $\mathbf{NARC} \leq \mathbf{NVERTEX}^2$ and `arc_gen` $= PATH : \mathbf{NARC} \leq \mathbf{NVERTEX} - 1$ of type **I3** which both relate the number of arcs and the number of vertices of a digraph. The first one has no precondition and therefore holds on any digraph, while the second one applies only on those digraphs that are a subset of an elementary path.

[2] $\mathbf{NCC}_1$ and $\mathbf{NCC}_2$ respectively denote the number of connected components of the final digraph of a first graph constraint and the number of connected components of the final digraph of a second graph constraint.

Implementing Graph Invariants. Most graph invariants are usually directly implemented as constraints which directly reduce the domains of the graph characteristics they involve. For this purpose we use:

- The arithmetic constraints of SICStus, which include constraints over non linear expressions [11, page 501],
- Propositional formulae over arithmetic constraints [11, page 461].

Finally, we also use indexicals [12, 13] for implementing some graph invariants. An *indexical* is a reactive function rule of the form X in R, where X is a domain variable and R is a set valued range expression.

Indexicals are used for encoding invariants that define a forbidden interval of values for a given graph characteristics (e.g. category **I3**) and for explicitly implementing the propagation of some non-linear arithmetic constraints for which the existing constraint propagation is too weak. Invariants of category **I3** have the form $C \notin [f_1(C_1,\ldots,C_n), f_2(C_{n+1},\ldots,C_m)]$, where $f_1(C_1,\ldots,C_n)$ and $f_2(C_{n+1},\ldots,C_m)$ are formulae involving the graph characteristics $C_1,\ldots,C_m$ distinct from C. The idea is to evaluate the maximum value, U, of $f_1(C_1,\ldots,C_n)$ as well as the minimum value, L, of $f_2(C_{n+1},\ldots,C_m)$ and to remove from C all values in $[U,L]$ when $U \leq L$. For this purpose we write range expressions for defining L and U.

Example 14. As an illustrative example of how to encode invariants defining a forbidden interval of values, consider the constraint $X \leq L \vee X \geq R$, which comes in handy for invariants such as $\mathbf{MIN\_NCC} \notin \left[\left\lfloor\frac{\mathbf{NVERTEX}}{2}\right\rfloor + 1, \mathbf{NVERTEX} - 1\right]$. This constraint can be encoded by three indexicals maintaining bounds consistency as follows:

```
not_strictly_between(X, L, U) +:
  X in (inf..max(L)) \/ (min(U)..sup),
  L in ((min(U)..max(X)) ? (inf   ..sup   )) \/ (min(X)..sup   ),
  U in ((min(X)..max(L)) ? (inf   ..sup   )) \/ (inf   ..max(X)).
```

Database Organisation. As we previously saw, we have graph invariants that hold for any digraph as well as tighter graph invariants for specific graph classes. As a consequence, we partition the database into groups of graph invariants. A *group of graph invariants* corresponds to several invariants such that all invariants relate to the same subset of graph characteristics and are variations of the first invariant of the group taking into accounts the graph class. Thus, the first invariant of a group has no precondition, while all other invariants have a non-empty precondition that characterizes the graph class for which they hold.

Example 15. As a first example, consider the following group of invariants, which relate the number of arcs $\mathbf{NARC}$ to the number of vertices of the smallest and largest connected component (i.e. $\mathbf{MIN\_NCC}$ and $\mathbf{MAX\_NCC}$) of a digraph:

- $\mathbf{MIN\_NCC} \neq \mathbf{MAX\_NCC} \Rightarrow \mathbf{NARC} \geq \mathbf{MIN\_NCC} + \mathbf{MAX\_NCC} - 2 + (\mathbf{MIN\_NCC} = 1)$[3],
- `equivalence` : $\mathbf{MIN\_NCC} \neq \mathbf{MAX\_NCC} \Rightarrow \mathbf{NARC} \geq \mathbf{MIN\_NCC}^2 + \mathbf{MAX\_NCC}^2$.

[3] The expression $(\mathbf{MIN\_NCC} = 1)$ is equal to 1 if $\mathbf{MIN\_NCC} = 1$ and 0 otherwise.

On the one hand, since the first invariant has no precondition, it can be applied to any digraph. On the other hand, the second invariant specifies a tighter condition (since $\mathbf{MIN\_NCC}^2 + \mathbf{MAX\_NCC}^2 \geq \mathbf{MIN\_NCC} + \mathbf{MAX\_NCC} - 2 + (\mathbf{MIN\_NCC} = 1)$) which only holds for a digraph that is reflexive, symmetric and transitive.

Example 16. As a second example, consider the following group of invariants, which relate the number of arcs $\mathbf{NARC}$ to the number of vertices $\mathbf{NVERTEX}$ according to the arc generator (see Figure 1) used for generating the initial digraph:

- $\mathbf{NARC} \leq \mathbf{NVERTEX}^2$,
- $\mathbf{arc\_gen} = \mathit{CIRCUIT} : \mathbf{NARC} \leq \mathbf{NVERTEX}$,
- $\mathbf{arc\_gen} = \mathit{CHAIN} : \mathbf{NARC} \leq 2 \cdot \mathbf{NVERTEX} - 2$,
- $\mathbf{arc\_gen} = \mathit{CLIQUE}(\leq) : \mathbf{NARC} \leq \frac{\mathbf{NVERTEX} \cdot (\mathbf{NVERTEX}+1)}{2}$,
- $\mathbf{arc\_gen} = \mathit{CLIQUE}(<) : \mathbf{NARC} \leq \frac{\mathbf{NVERTEX} \cdot (\mathbf{NVERTEX}-1)}{2}$,
- $\mathbf{arc\_gen} = \mathit{CLIQUE}(\neq) : \mathbf{NARC} \leq \mathbf{NVERTEX}^2 - \mathbf{NVERTEX}$,
- $\mathbf{arc\_gen} = \mathit{CYCLE} : \mathbf{NARC} \leq 2 \cdot \mathbf{NVERTEX}$,
- $\mathbf{arc\_gen} = \mathit{PATH} : \mathbf{NARC} \leq \mathbf{NVERTEX} - 1$.

The database currently contains 13, 50, 34, 12, 2 groups of invariants respectively mentioning 1, 2, 3, 4 and 5 graph characteristics. It also contains groups of invariants relating the graph characteristics of two digraphs. It contains 8, 6, 4, 10, 2 groups respectively mentioning 2, 3, 4, 5, 6 graph characteristics.

Extracting the Relevant Invariants. Once we have the graph invariants we can use them systematically by applying the following steps:

- For a given graph constraint we extract all the graph characteristics occurring in its description. This can be done automatically by scanning the corresponding graph properties. Let $\mathcal{GC}$ denote this subset of graph characteristics. For each graph characteristic gc of $\mathcal{GC}$ we check if we have a graph property of the form $gc = var$ where var is a domain variable. If this is the case we record the pair (gc, var); if not, we create a new domain variable var and also record the pair (gc, var).
- We then search for all groups of graph invariants involving a subset of the previous graph characteristics $\mathcal{GC}$. For each selected group we filter out those graph invariants for which the preconditions are not compatible with the graph class of the graph constraint under consideration. In each group we finally keep those invariants that have the maximum number of preconditions (i.e. the most specialized graph invariants).
- Finally we state all the previously collected graph invariants as implied constraints. This is achieved by using the variables associated to each graph characteristic.

5 Experimental Results

This section illustrates the approach on the `group` as well as on the `change_continuity` global constraints, which were previously introduced. We have compared the following approaches:

– In a first approach each graph characteristic was handled independently. This was concretely done by constructing an automaton for each graph characteristic and by reformulating that automaton as a conjunction of constraints as described in [14].
– The second approach reuses the first one but, in addition, also exploits the database of graph invariants in order to generate invariants which link the graph characteristics used in the description of `group` and of `change_continuity`.

We first detail the automata used for the `group` constraint. Since it is very similar to the `group` constraint, we then shortly discuss the implementation of the `change_continuity` constraint. Finally, we present the computational results obtained for the first and second approaches on the `group` as well as on the `change_continuity` constraints.

Implementing the `group` Constraint. Parts (A), (B), (C) and (D) of Figure 4 respectively depict the automata associated to the graph characteristics **NCC**, **MIN_NCC**, **MAX_NCC** and **NVERTEX** of the first graph constraint. Each automaton is applied to the sequence of variables corresponding to the VARIABLES parameter. A transition with a standard line depicts the fact that a variable takes its value within the set VALUES, while a thick line denotes the fact that a variable does not take its value within VALUES. Finally, a transition with a dashed line indicates the end of the sequence of variables. Since all the four automata use counters, we indicate how these counters are initialized in the initial state `s`, how a counter is unified to an argument of the `group` constraint in the final state `t`, and how they are possibly updated on a given transition. When there are several transitions between a given pair of states, we indicate with a dotted line or a standard line its type (see for instance the two transitions between `s` and `s` of the automaton depicted by part (C)).

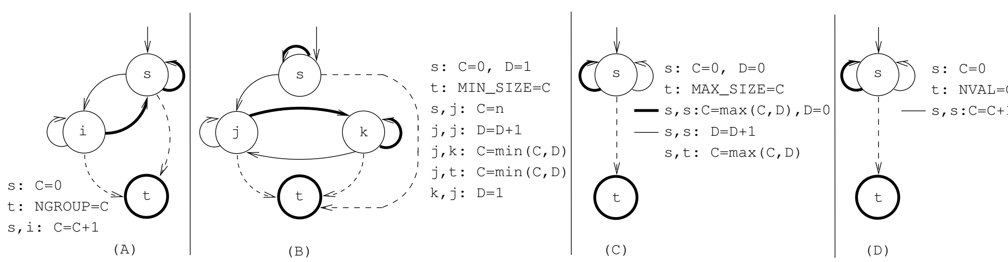


Fig. 4. Automata associated to the graph characteristics of the `group` constraint

The automata associated to **MIN_NCC** = MIN_DIST and to **MAX_NCC** = MAX_DIST are similar to the automata depicted by part (B) and (C), except that we change a thick line to a standard line and vice versa. The first approach for implementing the `group` constraint uses these six automata we just depicted. In the second approach we reuse the six automata and, in addition, extract a set of 51 graph invariants from the database of invariants.

Implementing the `change_continuity` Constraint. As for the `group` constraint, we came up with one automaton for each graph property. Parts (A), (B), (C) and (D) of

Figure 5 respectively depict the automata associated to the graph characteristics **NCC**, **MIN_NCC**, **MAX_NCC** and **NARC** of the first graph constraint. Each automaton is applied to the sequence of pairs of consecutive variables of the `VARIABLES` parameter. A transition with a standard (resp. thick) line depicts the fact that $\mathtt{VAR_i}$ CTR $\mathtt{VAR_{i+1}}$ holds (resp. does not hold). Finally, a transition with a dashed line indicates the end of the sequence of pairs of variables. Since all four automata use counters, we indicate how these counters are initialized in the initial state `s`, how a counter is unified to an argument of the `change_continuity` constraint in the final state `t`, and how they are possibly updated on a given transition. Since the second graph constraint of `change_continuity` is similar to the first graph constraint we don't give the four corresponding automata.

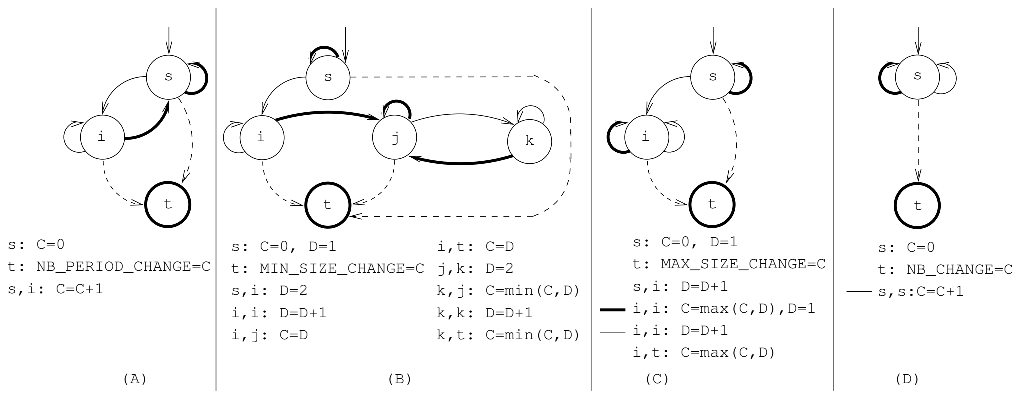


Fig. 5. Automata associated to the graph characteristics of the `change_continuity` constraint

The first approach for implementing the `change_continuity` constraint uses these eight automata. In the second approach we reuse the eight automata and, in addition, extract a set of 32 graph invariants from the database of invariants.

Performance. In order to evaluate the efficiency gained by adding graph invariants, we performed three experiments, generating random instances of the `group` and `change_continuity` constraints. `VARIABLES` was chosen as a sequence of N domains variables ranging over $[0, 1]$, `VALUES` as the singleton set $\{1\}$, and `CTR` as $=$. A constraint instance was generated by setting the initial domain of each domain variable to a randomly chosen interval.

In the first experiment, we computed the total domain size of the domain variables after posting, without invariants vs. with invariants, discarding infeasible instances, for $N = 8$. In the second experiment, we computed the time for posting the constraint instance and searching for all solutions, without invariants vs. with invariants, for $N = 8$. In the third experiment, we computed the time for posting the constraint instance and looking for the first solutions[4], without invariants vs. with invariants, for $N = 100$. Furthermore, with 10% probability, the variables in `VARIABLES` were fixed.

The results are presented in six scatter plots in Figure 6, one row per experiment. Each point represents a random instance, its X coordinate corresponding to excluding the invariants, and its Y coordinate corresponding to including them. The $X = Y$ line is shown in each graph. In the second and third rows, feasible and infeasible instances are denoted differently. Runtimes are in milliseconds.

From these experiments, we observe that the invariants significantly improve the domain reduction including detecting infeasible instances, but that they do not pay off for the purpose of just finding all solutions of feasible instances. However, in a more realistic setting, the improved domain reduction may well lead to savings in search effort that outweigh the overhead of the invariants.

6 Conclusion

The database of graph invariants introduced in this article can be seen as a way to automatically generate necessary conditions for global constraints that can be described in terms of graph properties. In fact, it complements the computation of lower and upper bounds for the graph characteristics presented in [15]. The key advantages of the approach are:

- Instead of developing a specific code for a given global constraint, we come up with graph invariants that can be applied to all global constraints sharing a given graph property.
- The database of graph invariants can be enriched incrementally and systematic experiments can point out missing graph invariants.

Finally, as demonstrated by our experiments on the `group` and the `change_continuity` constraints, it also clearly shows that the graph-based representation and the automaton-based representation of global constraints are not competing approaches for representing the meaning of a global constraint. In fact, when for a given global constraint, both representations are available[5] we can, without developing any specific code, get a filtering algorithm that takes advantage of both representations.

References

1. M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *8th European Conference on Artificial Intelligence, ECAI-88*, pages 290–295, Munich, Germany, August 1988. Pitmann Publishing, London.
2. A. Frisch, I. Miguel, and T. Walsh. Extensions to proof planning for generating implied constraints. In *Proceedings of Calculemus*, 2001.

[4] Each constraint instance was run with a 10 seconds time limit.

[5] Out of the 227 constraints of the catalog of global constraints [3], more than 100 global constraints use both representations.

3. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science, 2005.
4. N. Beldiceanu. Pruning for the *minimum* constraint family and for the *number of distinct values* constraint family. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 211–224. Springer-Verlag, 2001. Preprint available as SICS Tech Report T2000-10.
5. C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. Filtering algorithms for the *nvalue* constraint. In Romand Barták and Michela Milano, editors, *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, Lecture Notes in Computer Science, Prague, Czech Republic, may 2005. Springer Verlag.
6. P. Turán. On an extremal problem in graph theory. *Mat. Fiz. Lapok*, 48:436–452, 1941. In Hungarian.
7. COSYTEC. *CHIP Reference Manual*, release 5.1 edition, 1997.
8. J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.
9. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
10. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In Romand Barták and Michela Milano, editors, *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, Lecture Notes in Computer Science, Prague, Czech Republic, may 2005. Springer Verlag.
11. Mats Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 3.11.1 edition, February 2004. `http://www.sics.se/sicstus/`.
12. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Manuscript, 1991.
13. Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of finite domain constraints. In P. Van Hentenryck, editor, *ICLP'94, Int. Conf. on Logic Programming*, MIT Press Series in Logic Programming, S. Margherita Ligure, Italy, 1994. The MIT Press.
14. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In M. Wallace, editor, *Principles and Practice of Constraint Programming (CP'2004)*, volume 3258 of *LNCS*, pages 107–122. Springer-Verlag, 2004.
15. N. Beldiceanu, T. Petit, and G. Rochart. Bornes de caractéristiques de graphes. In *JFPC*, Lance, France, june 2005. In French.

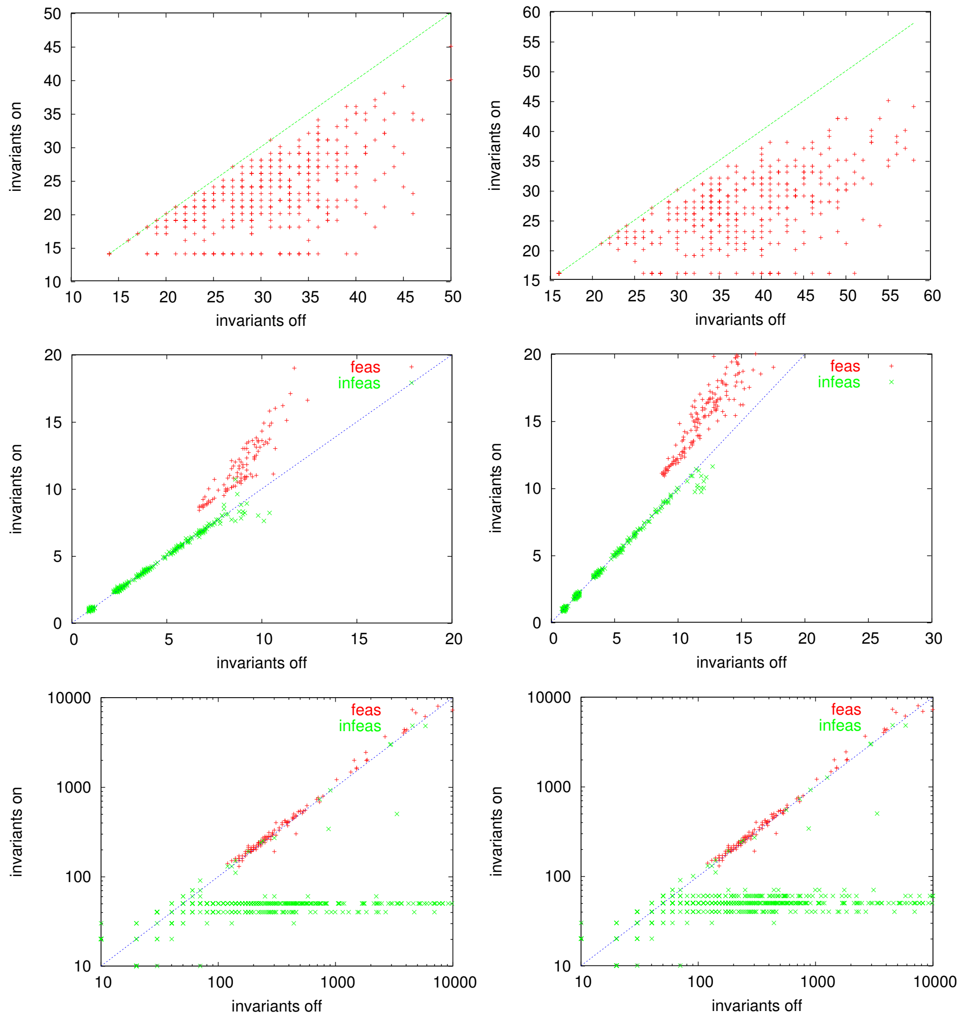


Fig. 6. Scatter plots of random instances. Top: comparing domain sizes. Middle: comparing runtime for finding all solutions. Bottom comparing runtime for finding first solution. Left: `group`. Right: `change_continuity`.

Among, Common and Disjoint Constraints

Christian Bessiere[1], Emmanuel Hebrard[2], Brahim Hnich[3], Zeynep Kiziltan[4], and Toby Walsh[2]

[1] LIRMM, CNRS/University of Montpellier, France, bessiere@lirmm.fr
[2] NICTA and UNSW, Sydney, Australia, {ehebrard, tw}@cse.unsw.edu.au
[3] 4C, University College Cork, Ireland, brahim@4c.ucc.ie
[4] University of Bologna, Italy, zkiziltan@deis.unibo.it

Abstract. AMONG, COMMON and DISJOINT are global constraints useful in modelling problems involving resources. We study a number of variations of these constraints over integer and set variables. For each case, we present a polynomial propagation algorithm, and identify when achieving a higher level of local consistency is intractable. Experiments demonstrate that such specialized propagators are useful in practice.

1 Introduction

Global constraints are an essential aspect of constraint programming. See, for example, [7, 3, 8, 2]. They specify patterns that occur in many problems, and exploit efficient and effective propagation algorithms to prune search. In problems involving resources, we often need to constrain the number of variables taking particular values. For instance, we might want to limit the number of tasks assigned to a particular resource. The AMONG, COMMON and DISJOINT constraints are useful in such circumstances.

The AMONG constraint was first introduced in CHIP to model resource allocation problems like car sequencing [3]. It counts the number of variables using values from a given set. A generalization of the AMONG and ALLDIFFERENT constraints is the COMMON constraint [2]. Given two sets of variables, this counts the number in each set which use values from the other set. A special case of the COMMON constraint also introduced in [2] is the DISJOINT constraint. This ensures that no value is common between two sets of variables. We study these three global constraints as well as seven other variations over integer and set variables. For each case, we present a polynomial propagation algorithm, and identify when achieving a higher level of local consistency is intractable. For example, rather surprisingly, even though the DISJOINT constraint is closely related to (but somewhat weaker than) the ALLDIFFERENT constraint, it is NP-hard to achieve generalised arc consistency on it.

2 Formal Background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A solution is an assignment of values to the variables satisfying the constraints. We consider both integer and set variables. A set variable S can be represented by a lower bound $lb(S)$ which contains the definite elements and an upper bound $ub(S)$ which contains the definite and potential elements. We use the following notations: X, Y, N, and M (possibly with subscripts) denote integer variables; S and T (again possibly with subscripts) denote set variables; $\mathcal{S}$ (possibly with a subscript) and $\mathcal{K}$ denote sets of integers; and v and k (possibly with a subscript) denote integer values. We write $\mathcal{D}(X)$ for the domain of a variable X. For integer domains, we write $min(X)$ and $max(X)$ for the minimum and maximum elements in $\mathcal{D}(X)$.

Constraint solvers often search in a space of partial assignments enforcing a local consistency property. A *bound support* for a constraint C is a partial assignment which satisfies C and assigns to each integer variable in C a value between its minimum and maximum, and to each set variable in C a set between its lower and upper bounds. A bound support in which each integer variable takes a value in its domain is a *hybrid support*. If C involves only integer variables, a hybrid support is a *support*. A constraint C is *bound consistent* (BC) iff for each integer variable X, $min(X)$ and $max(X)$ belong to a bound support, and for each set variable S, the values in $ub(S)$ belong to S in at least one bound support and the values in $lb(S)$ belong to S in all bound supports. A constraint C is *hybrid consistent* (HC) iff for each integer variable X, every value in $D(X)$ belongs to a hybrid support, and for each set variable S, the values in $ub(S)$ belong to S in at least one hybrid support, and the values in $lb(S)$ belong to S in all hybrid supports. A constraint C over integer variables is *generalized arc consistent* (GAC) iff for each variable X, every value in $D(X)$ belongs to a support. If all variables in C are integer variables, HC is equivalent to GAC, whilst if all variables in C are set variables, HC is equivalent to BC. Finally, we will compare local consistency properties applied to (sets of) logically equivalent constraints. A local consistency property Φ on C_1 is *strictly stronger* than Ψ on C_2 iff, given any domains, Φ removes all values Ψ removes, and sometimes more.

3 Integer Variables

3.1 Among constraint

The Among constraint counts the number of variables using values from a given set [3]. That is, $\text{Among}([X_1, .., X_n], [k_1, .., k_m], N)$ holds iff $N = |\{i \mid \exists j \,.\, X_i = k_j|$. For instance, we can use this constraint to limit the number of tasks (variables) assigned to a particular resource (value). Enforcing GAC on such a constraint is polynomial. Before we give an algorithm to do this, we establish the following theoretical results.

Lemma 1. *Given $\mathcal{K} = \{k_1, .., k_m\}$, $lb = |\{X_i \mid \mathcal{D}(X_i) \subseteq \mathcal{K}\}|$, and $ub = n - |\{X_i \mid \mathcal{D}(X_i) \cap \mathcal{K} = \emptyset\}|$, a value $v \in \mathcal{D}(N)$ is GAC for* Among *iff $lb \leq v \leq ub$.*

Proof. At most ub variables in $[X_1, .., X_n]$ can take a value from $\mathcal{K}$ and lb of these take values only from $\mathcal{K}$. Hence v is inconsistent if $v < lb$ or $v > ub$. We now need to show any value between lb and ub is consistent. We have $ub - lb$ variables that can take a value from $\mathcal{K}$ as well from outside $\mathcal{K}$. A support for $lb \leq v \leq ub$ can be constructed by assigning v variables to a value from $\mathcal{K}$ and $ub - v$ variables to a value from outside $\mathcal{K}$. □

Lemma 2. *Given $\mathcal{K} = \{k_1, .., k_m\}$, $lb = |\{X_i \mid D(X_i) \subseteq \mathcal{K}\}|$, $ub = n - |\{X_i \mid D(X_i) \cap \mathcal{K} = \emptyset\}|$, and $lb \leq min(N) \leq max(N) \leq ub$, a value in $D(X_i)$ may not be GAC for* Among *iff $lb = min(N) = max(N)$ or $min(N) = max(N) = ub$.*

Proof. The variables $[X_1, .., X_n]$ can be divided into three categories: 1) those whose domain contains values only from $\mathcal{K}$ (lb of them), 2) those whose domain contains both values from $\mathcal{K}$ and from outside ($ub - lb$ of them), and 3) those whose domain does not intersect with $\mathcal{K}$ ($n - ub$ of them). If $lb = min(N) = max(N)$ then exactly lb variables must take a value from $\mathcal{K}$. These variables can then only be those of the first category and thus $\mathcal{K}$ cannot be in the domains of the second category. If $min(N) = max(N) = ub$ then exactly ub variables must take a value from $\mathcal{K}$. These variables can then only be those of the first and the second category and thus any value $v \notin \mathcal{K}$ cannot be in the domains of the second category. We now need to show this is the only possibility for inconsistency. Consider an assignment to the constraint. Due to the variables of the first and the third category we have lb values from $\mathcal{K}$ and $n - ub$ values from outside $\mathcal{K}$. If $lb < max(N)$ then in the second category we can have at least one variable assigned to a value from $\mathcal{K}$, the rest assigned to a value outside $\mathcal{K}$ and satisfy the constraint. Similarly, if $min(N) < ub$ then in the second category we can have at least one variable assigned to a value outside of $\mathcal{K}$, the rest assigned to a value from $\mathcal{K}$ and satisfy the constraint. Hence, all values are consistent when $lb < max(N)$ or $min(N) < ub$. □

We now give an algorithm for the Among constraint.

Algorithm 1: GAC for Among($[X_1, .., X_n], \mathcal{K}, N$).

$lb := |\{X_i \mid D(X_i) \subseteq \mathcal{K}\}|$;
$ub := n - |\{X_i \mid D(X_i) \cap \mathcal{K} = \emptyset\}|$;
$min(N) := max(min(N), lb)$;
$max(N) := min(max(N), ub)$;
if $(max(N) < min(N))$ **then** fail;
if $(lb = min(N) = max(N))$ **then**
    foreach X_i . $\mathcal{D}(X_i) \nsubseteq \mathcal{K}$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \setminus \mathcal{K}$;
if $(min(N) = max(N) = ub)$ **then**
    foreach X_i . $\mathcal{D}(X_i) \cap \mathcal{K} \neq \emptyset$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \cap \mathcal{K}$;

Theorem 1 *Algorithm 1 maintains GAC on* Among($[X_1, .., X_n], [k_1, .., k_m], N$) *and runs in $O(nd)$ where d is the maximum domain size.*

Proof. (Sketch) By Lemmas 1 and 2, the algorithm maintains GAC. Computing lb and ub is in $O(nd)$. Updating the bounds of N is constant time. Updating

$\mathcal{D}(X_i)$ is in $O(d)$. Since there are n variables, pruning X_is is in $O(nd)$. GAC on Among is in $O(nd)$. □

An alternative method of propagating an Among constraint, which we consider in the experimental section, is with the global cardinality constraint Gcc [8].

$$\begin{aligned}&\textsc{Among}([X_1,\ldots,X_n],[k_1,..,k_m],N) \text{ iff}\\ &\textsc{Gcc}([X_1,\ldots,X_n],[k_1,..,k_m],[O_1,..,O_m]) \wedge\\ &\sum_{i\in\{1,..,m\}} O_i = N\end{aligned}$$

Unfortunately, this decomposition may hinder propagation. For example, consider $X_1, X_2 \in \{1,2\}$, $X_3 \in \{1,2,3\}$, $k_1 = 1$, $k_2 = 2$, and $N \in \{1,2\}$. The decomposition is GAC, but enforcing GAC on $\textsc{Among}([X_1,X_2,X_3],[k_1,k_2],N)$ would prune 1 and 2 from X_3 as well as 1 from N.

An interesting extension of the Among constraint is when we count not the variables taking some given values but those taking values taken by other variables. This is useful when, for example, the resources to be used are not initially known. We consider here two such extensions in which we replace $[k_1,..,k_m]$ either by a set variable S or by a sequence of variables $[Y_1,..,Y_m]$

$\textsc{Among}([X_1,..,X_n],S,N)$ holds iff N variables in X_i take values in the set S. That is, $N = |\{i \mid X_i \in S\}|$. Enforcing HC on this constraint is NP-hard in general.

Theorem 2 *Enforcing HC on* $\textsc{Among}([X_1,..,X_n],S,N)$ *is NP-hard.*

Proof. We reduce 3-Sat to the problem of deciding if such an Among constraint has a satisfying assignment. Finding hybrid support is therefore NP-hard. Consider a formula φ with n variables (labelled from 1 to n) and m clauses. Let k be $m+n+1$. To construct the Among constraint, we create $2k+1$ variables for each literal i in the formula such that $X_{i1}..X_{ik} \in \{i\}$, $X_{i(k+1)}..X_{i(2k)} \in \{-i\}$, and $X_{i(2k+1)} \in \{i,-i\}$. We create a variable Y_j for each clause j in φ and let $Y_j \in \{x,-y,z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. We let $N = n(k+1)+m$ and $\{\} \subseteq S \subseteq \{1,-1,..,n,-n\}$. The constraint $\textsc{Among}([X_{11},..,X_{1(2k+1)},..,X_{n1},..,X_{n(2k+1)},Y_1,..,Y_m],\ S,N)$ has a solution iff φ has a satisfying assignment. ♡

We now give a propagation algorithm for this Among constraint (algorithm 2). Notice that we assume all values are strictly positive. We highlight the differences with algorithm 1. The first modification is to replace each occurrence of $\mathcal{K}$ by either $lb(S)$ or $ub(S)$. In loop 1, we compute the contingent values of lb (resp. ub) when a value v is added to $lb(S)$ (resp. removed from $ub(S)$). These arrays are necessary for pruning N (lines 3, 6, 4, 7), when the minimum (resp. maximum) value of N cannot be achieved with the current lower (resp. upper) bound of S (conditionals 2 and 5). We also use them for pruning S (lines 8 and 9). Finally, we need to compute lb and ub, as they may have been affected by the pruning on S. This is done in line 10. The worst case time complexity is unchanged, as loop 1 can be done in O(nd).

Algorithm 2: Propagation for $\textsc{Among}([X_1,..,X_n],S,N)$.

$lb[0] := |\{X_i \mid D(X_i) \subseteq lb(S)\}|$;
$glb[0] := n - |\{X_i \mid D(X_i) \cap lb(S) = \emptyset\}|$;
$ub[0] := n - |\{X_i \mid D(X_i) \cap ub(S) = \emptyset\}|$;
$lub[0] := |\{X_i \mid D(X_i) \subseteq ub(S)\}|$;
foreach $v \in ub(S) \setminus lb(s)$ **do**
  $lb[v] := |\{X_i \mid D(X_i) \subseteq (lb(S) \cup \{v\})\}|$;
  $ub[v] := n - |\{X_i \mid D(X_i) \cap (ub(S) \setminus \{v\}) = \emptyset\}|$;
if $glb[0] < min(N)$ **then**
  $LB := \{lb[v] \mid v \in (ub(S) \setminus lb(S)), lb[v] \geq min(N)\}$;
  if *($LB \neq \emptyset$)* **then** $min(N) = min(LB)$;
else
  $min(N) := max(min(N), lb[0])$;
if $lub[0] > max(N)$ **then**
  $UB := \{ub[v] \mid v \in (ub(S) \setminus lb(S)), ub[v] \leq max(N)\}$;
  if *($UB \neq \emptyset$)* **then** $max(N) = max(UB)$;
else
  $max(N) := min(max(N), ub[0])$;
if *($max(N) < min(N)$)* **then** fail;
$lb(S) := lb(S) \cup \{v \mid ub[v] < min(N)\}$;
$ub(S) := ub(S) \setminus \{v \mid lb[v] > max(N)\}$;
if *($min(N) = max(N)$)* **then**
  $lb := |\{i \mid \mathcal{D}(X_I) \subseteq lb(S)\}|$;
  $ub := |\{i \mid \mathcal{D}(X_I) \cap ub(S) \neq \emptyset\}|$;
  if *($lb = min(N)$)* **then**
    foreach X_i . $\mathcal{D}(X_i) \not\subseteq lb(S)$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \setminus lb(S)$;
  if *($ub = max(N)$)* **then**
    foreach X_i . $\mathcal{D}(X_i) \cap ub(S) \neq \emptyset$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \cap ub(S)$;

The level of consistency achieved by this propagation algorithm is incomparable to BC. The following example shows that BC is not stronger: $X_1 \in \{2,3\}$, $X_2 \in \{2,3\}$, $X_3 \in \{1,2,3,4\}$, $lb(S) = ub(S) = \{2,3\}$, $min(N) = max(N) = 2$. The algorithm will prune $\{2,3\}$ from X_3, whereas a BC algorithm will not do any pruning. On the other hand, the following example shows that this algorithm does not enforce BC. Consider $X_1 \in \{1,2\}$, $X_2 \in \{1,2\}$, $X_3 \in \{3\}$, $X_4 \in \{3\}$, $X_5 \in \{4\}$, $X_6 \in \{4\}$, $X_6 \in \{5\}$, $X_8 \in \{5\}$, $lb(S) = \{1,2\}$, $ub(S) = \{1,2,3,4\}$, $N \in \{5,6,7,8\}$. The algorithm will not do any pruning whereas a BC algorithm will prune 5 from N.

The level of consistency achieved by this propagation algorithm is incomparable to BC. The following example shows that BC is not stronger: $X_1 \in \{2,3\}$, $X_2 \in \{2,3\}$, $X_3 \in \{1,2,3,4\}$, $lb(S) = ub(S) = \{2,3\}$, $min(N) = max(N) = 2$. The algorithm will prune $\{2,3\}$ from X_3, whereas a BC algorithm will not do any pruning. On the other hand, the following example shows that this algorithm does not enforce BC. Consider $X_1 \in \{1,2\}$, $X_2 \in \{1,2\}$, $X_3 \in \{3\}$, $X_4 \in \{3\}$, $X_5 \in \{4\}$, $X_6 \in \{4\}$, $X_7 \in \{5\}$, $X_8 \in \{5\}$, $lb(S) = \{1,2\}$, $ub(S) = \{1,2,3,4\}$,

$N \in \{5, 6, 7, 8\}$. The algorithm will not do any pruning whereas a BC algorithm will prune 5 from N.

We now consider the second generalization. $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ holds iff N variables in X_i take values in common with Y_j. That is, $N = |\{i \mid \exists j \,.\, X_i = Y_j\}|$. As before, we cannot expect to enforce GAC on this constraint.

Theorem 3 *Enforcing GAC on* $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ *is NP-hard.*

Proof. We again use a transformation from 3-SAT. Consider a formula φ with n variables (labelled from 1 to n) and m clauses. We construct the constraint $\textsc{Among}([Y_1, .., Y_m], [X_1, .., X_n], M)$ in which X_i represents the variable i and Y_j represents the clause j in φ. We let $M = m$, $X_i \in \{i, -i\}$ and $Y_j \in \{x, -y, z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. The constructed AMONG constraint has a solution iff φ has a model. ♡

To propagate $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$, we can use the following decomposition:

$$\begin{aligned}&\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N) \;\text{ iff}\\ &\textsc{Among}([X_1, .., X_n], S, N) \wedge \bigcup_{j \in \{1, .., m\}} \{Y_j\} = S\end{aligned}$$

We can therefore use the propagation algorithm just proposed for $\textsc{Among}([X_1, .., X_n], S, N)$. However, even if we were able to enforce HC on the decomposition (which is NP-hard in general to do), we may not make the original constraint GAC.

Theorem 4 *GAC on* $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ *is strictly stronger than HC on the decomposition.*

Proof: Consider $Y_1 \in \{1, 2, 3\}$, $X_1 \in \{1, 2\}$, $X_2 \in \{1, 2, 3\}$, $N = 2$. We have $\{\} \subseteq S \subseteq \{1, 2, 3\}$, hence the decomposition is HC. However, enforcing GAC on $\textsc{Among}([X_1, X_2], [Y_1], N)$ prunes 3 from Y_1 and X_2. ♡

3.2 Common constraint

A generalization of the AMONG and ALLDIFFERENT constraints introduced in [2] is the COMMON constraint. $\textsc{Common}(N, M, [X_1, .., X_n], [Y_1, .., Y_m])$ ensures $N = |\{i \mid \exists j \,.\, X_i = Y_j\}|$ and $M = |\{j \mid \exists i \,.\, X_i = Y_j\}|$. That is, N variables in X_i take values in common with Y_j and M variables in Y_j takes values in common with X_i.

Theorem 5 *Enforcing GAC on* COMMON *is NP-hard.*

Proof. Consider the reduction in the proof of Theorem 3. We let $N \in \{1, .., n\}$. The constructed COMMON constraint has a solution iff the original 3-SAT problem has a model. ♡

As we have a means of propagation for $\text{Among}([X_1,..,X_n],[Y_1,..,Y_m],N)$, we can use it to propagate the Common constraint using the following decomposition:

$$\begin{array}{l}\text{Common}(N,M,[X_1,..,X_n],[Y_1,..,Y_m]) \text{ iff} \\ \text{Among}([X_1,..,X_n],[Y_1,..,Y_m],N) \wedge \\ \text{Among}([Y_1,..,Y_m],[X_1,..,X_n],M)\end{array}$$

In the next theorem, we prove that we might not achieve GAC on Common even if we do so on Among.

Theorem 6 *GAC on* Common *is strictly stronger than GAC on the decomposition.*

Proof: Consider $N=2$, $M=1$, $X_1,Y_1 \in \{1,2\}$, $X_2 \in \{1,3\}$, $Y_2 \in \{1\}$, and $Y_3 \in \{2,3\}$. The decomposition is GAC. However, enforcing GAC on $\text{Common}(N,M,[X_1,X_2],[Y_1,Y_2,Y_3])$ prunes 2 from X_1, 3 from X_2, and 1 from Y_1. ♡

3.3 Disjoint constraint

We may require that two sequences of variables be disjoint (i.e. have no value in common). For instance, two sequences of tasks sharing the same resource might be required to be disjoint in time. The $\text{Disjoint}([X_1,..,X_n],[Y_1,..,Y_m])$ constraint introduced in [2] is a special case of the Common constraint where $N=M=0$. It ensures $X_i \neq Y_j$ for any i and j.

Theorem 7 *Enforcing GAC on* Disjoint *is NP-hard.*

Proof: We again use a transformation from 3-Sat. Consider a formula φ with n variables (labelled from 1 to n) and m clauses. We construct the Disjoint constraint in which X_i represents the variable i and Y_j represents the clause j in φ. We let $X_i \in \{i,-i\}$ and $Y_j \in \{-x,y,-z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. The constructed Disjoint constraint has a solution iff φ has a model. ♡

An obvious decomposition of the Disjoint constraint is to post an inequality constraint between every pair of X_i and Y_j, for all $i \in \{1,..,n\}$ and for all $j \in \{1,..,m\}$. Not surprisingly, the decomposition hinders propagation (otherwise we would have a polynomial algorithm for a NP-hard problem).

Theorem 8 *GAC on* Disjoint *is strictly stronger than AC on the decomposition.*

Proof: Consider $X_1,Y_1 \in \{1,2\}$, $X_2,Y_2 \in \{1,3\}$, $Y_3 \in \{2,3\}$. Then all the inequality constraints are AC. However, enforcing GAC on $\text{Disjoint}([X_1,X_2],[Y_1,Y_2,Y_3])$ prunes 2 from X_1, 3 from X_2, and 1 from both Y_1 and Y_2. ♡

Algorithm 3: BC for $\text{Among}([S_1, .., S_n], \mathcal{K}, N)$.

$InLb := f([lb(S_1, .., lb(S_n))], \mathcal{K})$;
$InUb := f([ub(S_1, .., ub(S_n))], \mathcal{K})$;
$min(N) := max(min(N), InLb)$;
$max(N) := min(max(N), InUb)$;
if $min(N) > max(N)$ **then** fail;
if $max(N) = InLb$ **then**
  foreach S_i . $lb(S_i) \cap \mathcal{K} = \emptyset$ **do** $ub(S_i) := ub(S_i) \setminus \mathcal{K}$;
if $min(N) = InUb$ **then**
  foreach S_i . $lb(S_i) \cap \mathcal{K} = \emptyset \wedge |\mathcal{K} \cap ub(S_i)| = 1$ **do**
    $lb(S_i) := lb(S_i) \cup \mathcal{K} \cap ub(S_i)$;

4 Set Variables

Many problems involve finding a set of values (for example, the set of nurses on a particular shift). It is useful therefore to have global constraints over set variables [9]. For instance, we might want to count the number of times each nurse has a shift during the monthly roster where each shift is a set variable listing the nurses on duty. This could be achieved with a global constraint that counted the values occurring in a sequence of set variables.

4.1 Among constraint

We consider an Among constraint over set variables that counts the number of these variables which contain one of the given values. That is, $\text{Among}([S_1, .., S_n], [k_1, .., k_m], N)$ ensures that $N = |\{i \mid \exists j \,.\, k_j \in S_i\}|$. Enforcing BC on such a constraint is polynomial. We propose an algorithm to do this where we define the function $f([\mathcal{S}_1, .., \mathcal{S}_n], \mathcal{K})$ to be $|\{i \mid \mathcal{S}_i \cap \mathcal{K} \neq \emptyset\}|$.

Theorem 9 *Algorithm 3 maintains BC on* $\text{Among}([S_1, .., S_n], \mathcal{K}, N)$ *and runs in* $O(nd)$ *where* d *is the size of the maximum upper bound of the* S_i *.*

Proof. Soundness is relatively immediate. N cannot be greater than the number of variables having k_j's in their upper bounds or smaller than the number of variables having k_j's in their lower bounds. Furthermore, if $max(N)$ is equal to the number of variables having k_j's in their lower bound, there is no hope to satisfy the Among constraint if we use a value k_j in another variable. If $min(N) = InUb$, then for each S_i, if there exists only one element in its upper bound (but not in its lower bound) which is also in $\mathcal{K}$, then that element has necessarily to belong to the lower bound of S_i as it cannot be covered by another S_j otherwise $min(N) < InUb$. Finally, if $min(N) > max(N)$ we necessarily fail.

To show completeness, we need that when we do not fail, the domains returned are bound consistent. Consider an integer k such that $InLb \leq k \leq InUb$. We can construct an assignment of S_i's where exactly k of them take a value in $\mathcal{K}$. We first assign all S_i's with their lower bound. $InLb$ of the S_i's necessarily contain some k_j since their lower bound overlaps $\mathcal{K}$. For $k - InLb$ variables among the $InUb - InLb$ variables with some k_j in their upper bound but none in their lower bound, we take some k_j from their upper bound to obtain a satisfying assignment with $N = k$. Since $min(N) \geq InLb$ and $max(N) \leq InUb$ (lines 3 and 4), N is BC. Suppose now a value v in $ub(S_i)$. The only case in which v should not belong to $ub(S_i)$ is when v is in the k_j's, none of the k_j's appear in $lb(S_i)$, and no more variable can take values in the k_j's, i.e., $InLb = max(N)$. Then, v will have been removed from $ub(S_i)$ (line 6). In addition, suppose v should belong to $lb(S_i)$. This is the case only if there is no k_j in $lb(S_i)$, and v is the only value in $ub(S_i)$ appearing in the k_j's, and $InUb = min(N)$. Then, v will have been added in $lb(S_i)$ (line 7).

Computing the counters $InLb$ and $InUb$ is in $O(nd)$. Updating the bounds on N is constant time. Deleting values that are not bound consistent in a $ub(S_i)$ or adding a value in $lb(S_i)$ is in $O(d)$. Since there are n variables, this phase is again in $O(nd)$. Bound consistency on Among is in $O(nd)$. $\square$

Note we can also add non-empty or cardinality conditions to the S_i without making constraint propagation intractable.

We again consider an extension in which we replace $[k_1, .., k_m]$ by a set variable S. Unlike the previous Among constraint, enforcing BC on $\text{Among}([S_1, .., S_n], S, N)$ is NP-hard in general.

Theorem 10 *Enforcing BC on* $\text{Among}([S_1, .., S_n], S, N)$ *is NP-hard.*

Proof. We reuse the reduction from the proof of Theorem 2 with minor modifications. We create $2k + 1$ set variables for each literal i in the formula such that $S_{i1}..S_{ik} \in \{i\}..\{i\}$, $S_{i(k+1)}..S_{i(2k)} \in \{-i\}..\{-i\}$, and $S_{i(2k+1)} \in \{\}..\{i, -i\}$. We create a set variable T_j for each clause j in φ and let $T_j \in \{\}..\{x, -y, z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. We let $N = n(k + 1) + m$ and $\{\} \subseteq S \subseteq \{1, -1, .., n, -n\}$. The constraint $\text{Among}([S_{11}, .., S_{1(2k+1)}, .., S_{n1}, .., S_{n(2k+1)}, T_1, .., T_m], S, N)$ has a solution iff φ has a model. ♡

Note that the constraint remains intractable if the S_i are non-empty or have a fixed cardinality. We can easily modify the reduction by adding distinct "dummy" values to S_i and T_j respectively. We also add these dummy values to the lower bound of S.

Despite this intractability result, we can easily modify Algorithm 3 to derive a filtering algorithm for $\text{Among}([S_1, .., S_n], S, N)$ without changing the complexity. We use the lower bound of S (resp. $ub(S)$) in the computation of $InLb$ (resp. $InUb$). Also, instead of $\mathcal{K}$ in line 6 (resp. line 7), we use $lb(S)$ (resp. $ub(S)$). Finally, we need to consider the bounds of S. We remove v from $ub(S)$ if $|\{S_i \mid lb(S_i) \subseteq lb(S) \cup \{v\}\}| > max(N)$. Similarly, we add v to $lb(S)$ if $|\{S_i \mid ub(S_i) \cap ub(S) \setminus \{v\} \neq \emptyset\}| < min(N)$. We can easily extend the soundness proof of Theorem 9 for this algorithm, but we skip it here for space reasons. Such an algorithm does not achieve BC.

The constraint $\text{Among}([S_1, .., S_n], [T_1, .., T_m], N)$ is very similar since several set variables $[T_1, \ldots, T_m]$ behave like their union. That is, $\text{Among}([S_1, .., S_n], [T_1, .., T_m], N)$ is similar to $\text{Among}([S_1, .., S_n], T, N)$ with $T = \bigcup_{j \in \{1..m\}} T_j$.

Algorithm 4: BC for DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$).

1 $S := \bigcup_{i \in \{1,..,n\}} (lb(S_i))$;
2 $T := \bigcup_{i \in \{1,..,m\}} (lb(T_i))$;
3 **if** $S \cap T \neq \emptyset$ **then** fail;
4 **foreach** S_i **do** $ub(S_i) := ub(S_i) \setminus T$;
5 **foreach** T_j **do** $ub(T_j) := ub(T_j) \setminus S$;

4.2 Common constraint

We may also want to post a COMMON constraint on set variables. COMMON($N, M, [S_1, .., S_n], [T_1, .., T_m]$) ensures that $N = |\{i \mid \exists j \,.\, S_i \cap T_j \neq \emptyset\}|$ and $M = |\{j \mid \exists i \,.\, S_i \cap T_j \neq \emptyset\}|$. Enforcing BC on such a constraint is intractable as it is an extension of the previous AMONG constraint. We can reduce AMONG($[S_1, .., S_n], [T_1, .., T_m], N$) to COMMON($N, M, [S_1, .., S_n], [T_1, .., T_m]$) by setting M to $\{0, .., m\}$.

Since we have a means of propagation for AMONG($[S_1, .., S_n], [T_1, .., T_m], N$), we can use it to propagate the COMMON constraint by decomposing it into two such AMONG constraints. However, such a decomposition hurts propagation. Consider the set variables S_1, S_2, S_3, T_1, T_2 and T_3 with $\{i\} \subseteq S_i \subseteq \{i\}$, $N = 1$, $\{\} \subseteq T_i \subseteq \{i\}$, and $M = 2$. The two AMONG constraints of the decomposition are BC while COMMON is inconsistent.

4.3 Disjoint constraint

We finally consider DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$). Unlike GAC on DISJOINT with integer variables, we can maintain BC on DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$) in polynomial time.

Theorem 11 *Algorithm 4 maintains BC on* DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$) *and runs in* $O((n + m)d)$.

Proof. We first show that the algorithm is sound. If there exists one value which occurs both in the lower bound of one of S_i and in the lower bound of one of the T_j, then we necessarily fail (Line 3). If a value is consumed by one of the T_j, then we cannot satisfy the constraint if this value is allowed to be consumed by one of the S_i (Line 4). Similarly, if a value is consumed by one of the S_i, then we cannot satisfy the constraint if this value is allowed to be consumed by one of the T_j (Line 5).

To show completeness, we prove that either we fail or we return bound consistent domains. We only consider the S_i as the reasoning is analogous for the T_j. First, if the lower bounds of the S_i do not overlap those of the T_j, then assigning all S_i and T_j to their lower bound is a solution. Thus, the lower bounds are BC. Now, for each value in the upper bound of each S_i, we can construct a satisfying assignment involving v by assigning all other S_i to their lower bounds, S_i to its lower bound plus the element v, and all the T_j to their corresponding lower bounds as none has v as an element.

	GCC+SUM			AMONG		
Size	#fails	cpu time	#solved	#fails	cpu time	#solved
10	6	0.02	9/10	6	0.01	9/10
15	6,475	1.56	29/52	6,468	0.99	29/52
20	2,425	1.17	20/35	2,425	0.66	20/35
25	317	0.41	13/20	317	0.17	13/20
30	12,472	11.87	6/10	12,472	5.41	7/10
35	14	0.44	20/56	13	0.12	22/56

Table 1. Mystery shopper problem (prob004 in CSPLib). Each instance is run with a 5 minutes time limit. #fails and cpu time are averaged over instances solved by both methods.

If d is the total number of values appearing in the upper bounds of the set variables, then at worst case the complexity of line 4 is $O(nd)$ and of line 5 is $O(md)$. Hence, the algorithm runs in $O((n+m)d)$. □

Note that if we add a cardinality restriction to the size of the set variables, it becomes NP-hard to enforce BC on this constraint.

5 Experimental Results

We ran some experiments to test the efficiency and effectiveness of these propagation algorithms. We used a model for the Mystery Shopper Problem [6] due to Helmut Simonis that appears in CSPLib. This model contains a large number of AMONG$([X_1,..,X_n],[k_1,..,k_m],N)$ constraints. As ILOG's Solver does not contain this global constraint, we implemented our AMONG constraint and compared it with the decomposition into a set of GCC constraints and a SUM constraint as shown previously. In order to generate a range of instances, we used the following protocol. The duration in weeks as well as the number of varieties of shoppers, visits per saleslady and number of areas are all set to 4 (as in the original problem). The number of salesladies takes its value in $\{10,15,20,25,30,35\}$, and the number of shoppers is the next multiple of 4 that is not consecutive. All the possible ways of partitioning the salesladies into 4 areas (that do not trivially violate the constraints) are solved. We report the results in Table 1. The best results are obtained using our specialized propagator. Although the number of fails does not reduce greatly, our algorithm is much more efficient.

6 Related Work

AMONG$([X_1,..,X_n],[k_1,..,k_m],N)$ was first introduced in CHIP by [3]. However, we are not aware of any published algorithm to maintain GAC on such a constraint nor of any study of the extensions considered here. A closely related constraint is the COUNT constraint [10]. COUNT$([X_1,..,X_n],v,op,N)$ where $op \in \{\leq,\geq,<,>,\neq,=\}$ holds iff N op $|\{i \mid X_i = v\}|$. The AMONG constraint is more general as it counts the variables taking values from a set whereas COUNT counts those taking a given value. The algorithm of AMONG can easily be adapted to cover the operations considered in COUNT.

There are other counting and related constraints for which there are specialised propagation algorithms such as global cardinality [8], the NVALUE [1], SAME and USEDBY [4].

In [5], a wide range of counting and occurrence constraints are specified using two primitive global constraints, ROOTS and RANGE. For instance, the AMONG constraint is decomposed into a ROOTS and set cardinality constraint. Similarly, the COMMON constraint is decomposed into two ROOTS, two RANGE and two set cardinality constraints. However, specialized propagator like those proposed here are likely to be faster and more effective than generic propagators like ROOTS and RANGE. We ran some experiments with the mystery shopper problem and observed a factor of two saving in runtimes with a specialized propagator on the more difficult problems.

7 Conclusions

We have studied a number of variations of the AMONG, COMMON and DISJOINT constraints over integer and set variables. Such constraints are useful in modelling problems involving resources. In Table 2, we present a summary of our complexity results. For integer variables, we propose a polynomial time propagation algorithm for the AMONG constraint that achieves GAC. We prove that AMONG becomes intractable when we count the number of variables using values from *a set variable* or a *sequence of integer variables*. Nevertheless, we propose a polynomial algorithm to propagate the former and show how this algorithm can be used to propagate the latter. We also show that the COMMON constraint is intractable in general, and this holds even in the special case of the DISJOINT constraint when the number of variables in common is zero. The last result is somewhat surprising, since the DISJOINT constraint is related to (and weaker than) the ALLDIFFERENT constraint. When we demonstrate the intractability of a constraint like DISJOINT, we also present a polynomial method to propagate the constraint. Finally, we consider AMONG, COMMON and DISJOINT constraints over set variables rather than integer variables. We show that most of the results on integer variables hold for set variables with the exception that the DISJOINT constraint now becomes tractable. Experimental results show the value of these specialised propagation algorithms in practice.

References

1. Beldiceanu, N. 2001. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proc. of CP 2001*, 211–224. Springer.
2. Beldiceanu, N. 2000. Global constraints as graph properties on a structured network of elementary constraints of the same type. SICS Technical Report T2000/01.

Constraint	Tractability
AMONG($[X_1, .., X_n], \mathcal{K}, N$)	GAC is in P
AMONG($[X_1, .., X_n], S, N$)	HC is NP-hard
AMONG($[X_1, .., X_n], [Y_1, .., Y_m], N$)	GAC is NP-hard
COMMON($N, M, [X_1, .., X_n], [Y_1, .., Y_m]$)	GAC is NP-hard
DISJOINT($[X_1, .., X_n], [Y_1, .., Y_m]$)	GAC is NP-hard
AMONG($[S_1, .., S_n], \mathcal{K}, N$)	BC is in P
AMONG($[S_1, .., S_n], S, N$)	BC is NP-hard
AMONG($[S_1, .., S_n], [T_1, .., T_m], N$)	BC is NP-hard
COMMON($N, M, [S_1, .., S_n], [T_1, .., T_m]$)	BC is NP-hard
DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$)	BC is in P

Table 2. Summary of complexity results.

3. Beldiceanu, N., and Contegean, E. 1994. Introducing global constraints in CHIP. *Mathematical Computer Modelling* 20(12):97–123.
4. Beldiceanu, N., Katriel, I., and Thiel, S. 2004. Filtering algorithms for the *same* and *usedby* constraints. MPI Technical Report MPI-I-2004-1-001.
5. Bessiere, C., Hebrard, E., Hnich, B., Kizilitan Z. and Walsh, T. 2005. Counting and Occurrence Constraints. To appear in *Proc. of IJCAI 2005*.
6. Cheng, B.M.W., Choi, K.M.F., Lee, J.H.M. and Wu, J.C.K. 1999. Increasing Constraint Propagation by Redundant Modeling: an Experience Report. *Constraints* 4(2): 167–192.
7. Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI 1994*, 362–367.
8. Régin, J.-C. 1996. Generalized arc consistency for global cardinality constraints. In *Proc. of AAAI 1996*, 209–215.
9. Sadler, A., and Gervet, C. 2001. Global reasoning on sets. In *Proc. of Workshop on Modelling and Problem Formulation (FORMUL'01)*.
10. Swedish Institue of Computer Science. 2004. *SICStus Prolog User's Manual, Release 3.12.0*.

Improved algorithm for finding (a,b)-super solutions

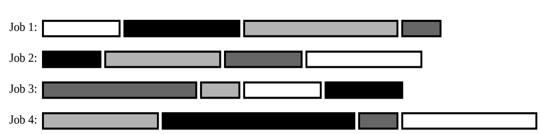
Emmanuel Hebrard[1], Brahim Hnich[2], and Toby Walsh[1]

[1] NICTA and UNSW, Sydney, Australia, {ehebrard, tw}@cse.unsw.edu.au
[2] 4C, University College Cork, Ireland, brahim@4c.ucc.ie

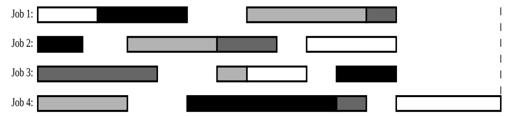
Abstract. In [?], the authors introduced to constraint programming the notion of (a, b)-super solutions. They are solutions in which, if a small number of variables lose their values, we are guaranteed to be able to repair the solution with only a few changes. This concept is useful for scheduling in dynamic and uncertain environments when the robustness of the schedule is a valuable property. We introduce a new algorithm for finding super solutions that improves upon the method introduced in [?] in several dimensions. This algorithm is more space efficient as it only requires to double the size of the original constraint satisfaction problem. The new algorithm also permits us to use any constraint toolkit to solve the master problem as well as the sub-problems generated during search. We also take advantage of multi-directionality and of inference based on the *neighborhood* notion to make the search for a solution faster. Moreover, this algorithm allows the user to easily specify extra constraints on the

1 Introduction

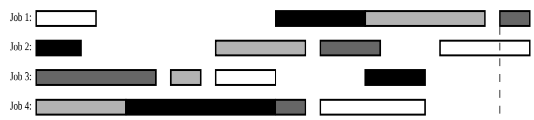
In [?], the authors introduced to constraint programming the notion of (a, b)-super solutions. Super solutions are a generalization of both fault tolerant solutions [?] and super models [?]. These are solutions such that a small (bounded) perturbation on the input will have proportionally small repercussions on the outcome. For instance when solving a scheduling problem, we may want that, in the event of a machine breaking, or of a task executing longer than expected, the rest of the schedule should change as little as possible if at all. As a concrete example, consider the following job-shop scheduling problem, where we need to schedule four jobs consisting of four activities, each requiring a different machine. The usage of a machine is exclusive, and the sequence of a job is to be respected.



The second figure shows an optimal solution.



One may argue that this solution is not robust. Indeed activities are tightly grouped and a "break" on a machine, and the subsequent delay, may trigger further delays in the schedule. On the other hand the next figure shows a solution where, for a small makespan increase, a large proportion of activities can be delayed of two units of time, without affecting at all the rest of the schedule.



Such solutions are thus *stable* in the sense that when invalidated, a close alternative solution can be applied. However, there are a number of other ways to capture robustness. for instance in [**?**], the approach to robustness is probabilistic and a robust solution is simply one that is likely to remain valid after a contingent change. The problem of scheduling under uncertainty has been widely studied in the past (See [**?**] and [**?**] for instance). We wish to investigate how super solutions compare to these specialized methods. For instance, if we consider the *slack-based* framework [**?**], the intuitive idea is that a local perturbation will be "absorbed" if enough slack is available, end therefore solutions involving slack are preferred. Now, one can think of a scenario where the best reaction to a delay or a break would not be to delay the corresponding (plus perhaps few other) activity, but to postpone it and advance the starting time of another activity instead. This situation is not captured by slack based method. Although, it is important to notice that if the latter approach aims at minimizing the impact of a delay on the makespan, it is only a secondary consequence for super solutions. Indeed, the main concern is to limit the number of activities to reschedule. Therefore, a direct comparison is difficult. The concept of *flexible schedule* is more closely related to our view of robustness as this method promotes stability against localized perturbations.

On the other hand, super solutions have a priori several drawbacks compared to such specialized approaches.

Firstly, finding super solutions is a difficult problem and as a result the methods proposed so far are often restricted to toy problems like the one used in the previous example. Algorithms with better performance have been proposed for very restricted classes of super solution. However, if we do not restrict ourselves to these classes, solving a problem of reasonable size is often out of reach with the current approaches.

Another difficulty is that, being a general framework, it is not always immediately applicable to a specific problem. In [**?**] we showed how super solutions have to be adapted to cope with specifics of job shop scheduling problems in particular. For instance, if variables are activities and values are time points, then we cannot schedule to an earlier time point as a response to a break. Moreover, moving the same activity to the next consecutive time point may not be a valid alternative.

In this paper we introduce a new algorithm that can help to address the above drawbacks. The central feature of the new algorithm is that it is closer to a "regular" solver, we solve both the original problem as well as sub-problems generated during search using any standard constraint solver. Therefore, methods that have been proven to be effective in a particular area (like shaving [**?**], specialized variable orderings [**?**] or specialized constraint propagators) can be used both for the main problem and the sub-problems. We also propose a *more effcient* and *more effective* algorithm than what has been proposed in [**?**]. The new algorithm is more efficient as we avoid solving *all* sub-problems and it is *more effective* by using the information gathered when solving these sub-problems to get more pruning on future variables. Moreover, this architecture also helps to adapt the criteria of robustness to the problem. Indeed, to model a particular requirement we can just add it as a constraint to the sub-problems.

2 Formal background and notations

A constraint satisfaction problem (CSP) P consists of a set of variables $\mathcal{X}$, a set of domains $\mathcal{D}$ such that $\mathcal{D}(X_i)$ is the finite set of values that can be taken by the variable X_i, and a set of constraints $\mathcal{C}$ that specify allowed combinations of values for subsets of variables. We use upper case for variables (X_i) and lower case for values (v). A full or partial instantiation $S = \{\langle X_1 : v_1\rangle, \dots \langle X_n : v_n\rangle\}$ is a set of assignments $\langle X_i : v_j\rangle$ such that $v_j \in X_i$. We will use $S[i]$ to denote the value assigned to X_i in S. A (partial) solution is an instantiation satisfying the constraints. Given a constraint C_V on a set of variables V, a *support* for $X_i = v_j$ on C is a partial solution involving the variables in V and containing $X_i = v_j$. A variable X_i is *generalized arc consistent* (*GAC*) on C iff every value in $\mathcal{D}(X_i)$ has support on C. A constraint C is GAC iff each constrained variable is GAC on C, and a problem is GAC iff all constraints in $\mathcal{C}$ are GAC. Given a CSP P and a subset $A = \{X_{i_1}, \dots X_{i_k}\}$ of $\mathcal{X}$, a solution S of the restriction of P to A (denoted $P|_A$) is a partial solution on A such that if we restrict $\mathcal{D}(X_i)$ to $\{S[i]\}$ for $i \in [i_1..i_k]$, then P can be made GAC without domain wipe-out.

We introduce some notations used later in the paper. the function $H(S, R)$ is defined to be the Hamming distance between two solutions R and S, i.e., the number of variables assigned to different values in S and R. We also define $H_A(S, R)$ to be the Hamming distance restricted to the variables in A.

$$H_A(S, R) = \sum_{X_i \in A} (S[i] \neq R[i])$$

An *a-break* on a (partial) solution S is a combination of a variables among the variables in S. A *b-repair* of S is a (partial) solution R such that $H_A(S, R) = |A|$ and $H(S, R) \leq (a + b)$. In other words, R is an alternative solution for S

such that if the assignments of the variables in A are forbidden, the remaining "perturbation" is restricted to b variables.

Definition 1. *A solution S is an (a, b)-super solution iff for every $a' \leq a$, and for every a'-break of S, there exists a b-repair of S.*

3 The basic algorithm

We first describe a very simple and basic version of the algorithm without any unnecessary features. Then we progressively introduce modifications to make the algorithm more efficient and more effective.

The basic idea is to ensure that the current partial solution is also a partial super solution. In order to do so, as many sub-problems as possible breaks for this partial solution have to be solved. The solutions to these sub-problems are partial repair solutions. We therefore work on a copy of the original problem that we change and solve for each test of *reparability.* Note that the sub-problem is much easier to solve than the main problem, for several reasons. The first reason is that each of the sub-problems is polynomial. Indeed, since a repair solution must have less than $a + b$ discrepancies with the main solution, the number of possibilities is bounded by $n^{a+b}d^{a+b}$. Typically, the cost of solving one such sub-problem will be far below this bound since constraint propagation is used. Another reason is that we can reuse the variable ordering dynamically computed whilst solving the main problem. Furthermore, we will demonstrate later that not all breaks have to be checked, and that we can infer inconsistent values from the process of looking for a repair. Pruning the main problem is critical, as it not only reduces the search tree, but also reduces the number of sub-problems to be solved.

The algorithm for finding super solutions is, in many respects, comparable to a global constraint. However, it is important to notice that we *cannot* define a global constraint ensuring that the solution returned is a super solution as this condition depends on the variables domains, whilst constraints should only depend on the values assigned to variables and not their domains. For example, consider two variables X_1, X_2 taking values in $\{1, 2, 3\}$ such that $X_1 < X_2$. The assignment $\langle X_1 = 1, X_2 = 3\rangle$ is a $(1, 0)$-super solution, however, if the original domain of X_1 is $\{1, 3\}$, then $\langle X_1 = 1, X_2 = 3\rangle$, is *not* a $(1, 0)$-super solution. Nevertheless, the algorithm we introduce could be seen as a global constraint implementation as it is essentially an oracle tightening the original problem It is however important to ensure that this oracle is not called for every change in a domain as it can be costly.

Initialization: In Algorithm **??**, we propose an pseudo code for finding super solutions. The input is a CSP, i.e., a triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and the output a (a, b)-super solution S. We first create a copy P' of P, where $X'_i \in \mathcal{X}'$ iff $X_i \in \mathcal{X}$ and $C' \in \mathcal{C}'$ iff $C \in \mathcal{C}$. This copy will be used to find b-repairs. At any point in the algorithm, $\mathcal{D}(X_i)$ (resp. $\mathcal{D}'(X'_i)$) is the *current* domain of X_i

(resp. X'_i). The set $Past \subseteq \mathcal{X}$ contains all variables that are already bound to a value and we denote $Past'$ the set containing the same variables, but "primed", $Past' = \{X'_i | X_i \in Past\}$.

Algorithm 1: `super-MAC`(P, a, b)

```
Data    : P, a, b
Result  : S: an (a, b)-super solution
S ← ∅; Past ← ∅; P' ← P;
if ¬backtrack(P, P', S, Past, a, b) then
  └ print "NO SUPER-SOLUTION";
print "A SUPER-SOLUTION FOUND: S";
```

Main Backtracker Procedure (Algorithm **??**) searches and backtracks on the main problem P. It is in very similar to a classical backtracker that maintain GAC at each node of the search tree, except that we also add a call to the procedure `reparability` at each node. Note that any solver or local/global consistency property can be used instead as long as the procedure `reparability` is called. A possible way of implementing `reparability` –in a standard constraint toolkit– can be as a global constraint containing internally the extra data structure P' and an associated specialised solver. As such global constraint can be costly, it should not be called more than once per node.

Algorithm 2: `backtrack`$(P, P', S, Past, a, b)$: Bool

```
if Past = X then return True;
choose X_i ∈ X \ Past;
Past ← Past ∪ {X_i};
foreach v ∈ D(X_i) do
  | save D;
  | D(X_i) ← {v};
  | S ← S ∪ {⟨X_i : v⟩};
  | if AC-propagate(P) & reparability(P, P', S, Past, a, b)  then
  |   └ if backtrack(P, S, Past, a, b) then return True;
  | restore D;
  └ S ← S − {⟨X_i : v⟩};
Past = Past − {X_i};
return False;
```

Enforcing reparability: Procedure `reparability` (Algorithm **??**) makes sure that each a-break of the solution S has a b-repair. If $|S| = k$ then we check all combinations of less than a variables in S, that is $\sum_{j \le a} \binom{k}{j}$ breaks. This number has no closed form, though it is bounded above by k^a. For each a-break, we model the problem of the existence of a b-repair using P'. Given the main solution S and a break A, we need to find a b-repair, that is, a solution R of $P'|_{Past'}$ such that $H_A(S, R) = |A|$ and $H(S, R) \le |A| + b$. The domains of all variables are set to their original state. Then for any $X'_i \in A$, we remove the value $S[i]$ from

$\mathcal{D}'(X'_i)$, thus making sure that $H_A(S,R) = |A|$. We also add an AtMost$k$Diff constraint that ensures $H(S,R) \leq k$, where $k = |A|+b$. Finally, we solve $P'|_{Past'}$, it is easy to see that any solution is a b-repair.

Algorithm 3: `repairability`$(P, S, Past, a, b)$:Bool

```
foreach A ⊆ Past' such that |A| ≤ a do
    foreach X_i ∈ A do
        D'(X'_i) ← D(X_i) − {S[i]};
    k ← (|A| + b);
    S' ← solve(P'|_Past' + AtMostkDiff(X', S));
    if S' = nil then return False;
return True;
```

Propagating the AtMost$k$Diff *constraint:* The AtMost$k$Diff constraint is defined as follows:

Definition 2. AtMost$k$Diff$(X'_1, \ldots X'_n, S)$ *holds iff* $k \geq \sum_{i\in[1..n]}(X'_i \neq S[i])$

This constraint ensures that the solution we find is a valid partial b-repair by constraining the number of discrepancies to the main solution to be lower than $a + b$. To enforce GAC on such a constraint, we first compute the smallest expected number of discrepancies to S. Since S is a partial solution we consider the *possible* extensions of S. Therefore, when applied to the auxiliary CSP P' this number is simply

$$d = |\{i|\mathcal{D}'(X'_i) \cap \mathcal{D}(X_i) = \emptyset\}|$$

We have three cases:

1. If $d < k$ then the constraint is GAC as every variable can be assigned any value providing that all other variables X'_i take a value included in $\mathcal{D}(X_i)$, and we will still have $d \leq k$.
2. If $d > k$ then the constraint cannot be satisfied.
3. If $d = k$ then we can set the domain of any variable X'_i such that $\mathcal{D}(X'_i) \cap \mathcal{D}(X_i) \neq \emptyset$ to $S[i]$.

Comparison with previous algorithm This new algorithm is simpler than the one given in [**?**] as no extra data structure is required for keeping the current state of a repair. Moreover, the space required is at most twice the space required for solving the original problem, whilst the previous algorithm stored the state of each search for a b-repair. We want to avoid such data structures as they are exponential in a. Even though a is typically a small constant, this can be prohibitive. Another advantage in doing so is that the search for a repair can easily be done, whereas in the previous algorithm, doing so would have been difficult without keeping as many solver states as breaks, since the search was starting from the point it ended in the previous call.

The search tree explored by this algorithm is strictly smaller than that explored by the previous algorithm. The methods are very comparable as they both solve sub-problems to find a b-repair for each break. However, since the sub-problems solved by this previous algorithm were implemented as a simple backtrack procedure (without constraint propagation), it was not possible to check if a b-repair would induce an arc inconsistency in an unassigned variable.

4 Improvements

Now we explore several ways of improving the basic algorithm.

4.1 Repair multi-directionality

The multi-directionality is a concept used for instance for implementing general purpose algorithms for enforcing GAC. The idea is that a tuple is a support for every value it involves. The same metaphor applies when seeking repairs instead of supports. In our case, suppose that we look for a $(2,2)$-super solution and suppose that R is a repair solution for a break on the variables $\{X, Y\}$ that require reassigning the variables $\{V, W\}$. This constitutes also a repair for $\{X, V\}, \{X, W\}, \{Y, V\}, \{Y, W\}$ and $\{V, W\}$. We therefore need not to look for repair for these breaks.

We used a simple algorithm from Knuth [**?**] to generate all $\leq a$-breaks. This algorithm generates the combinations in a deterministic manner, and therefore constitutes an ordering on these combinations. Moreover, this ordering has the nice property that given one combination in input, one can compute the rank of this combination in the ordering in linear time on the size of the tuple. The size of the tuple is in our case a small constant, we thus have an efficient way of knowing if the break that we currently consider is covered by an earlier repair. Each time a new repair is computed, all breaks it covers are added to a set, then when we generate a combination, we simply check that its index is not in this set otherwise we do not need to find a repair for it.

4.2 Neighborhood-based inference:

The second observation that we make to improve the efficiency is less obvious but has broader consequences. First, let us introduce some necessary notation:

A path linking two variables X and Y is a sequence of constraints $C_{V_1}, \ldots C_{V_k}$ such that $i = j+1 \Rightarrow V_i \cap V_j \neq \emptyset$ and $X \in V_1$ and $Y \in V_k$, k is the length of the path. The distance between two variables $\delta(X, Y)$ is the length of the shortest path between these variables ($\delta(X, X) = 0$). $\Delta_d(X)$ denotes the neighborhood at a distance exactly d of X, i.e., $\Delta_d(X) = \{Y \mid \delta(X, Y) = d\}$. $\Gamma_d(X)$ denotes the neighborhood up to a distance d of X i.e., $\Gamma_d(X) = \{Y \mid \delta(X, Y) \leq d\}$. Similarly, we define the neighborhood $\Gamma_d(A)$ (resp. $\Delta_d(A)$) of a subset of variable A as simply $\bigcup_{X \in A} \Gamma_d(X)$ (resp. $\Delta_d(A)$).

Now we can state the following lemma which will be central to all the subsequents improvements. It shows that if there exists a b-repair for a particular a-break A, then all reassignments are within the neighborhood of A up to a distance b.

Lemma 1. *Given a solution S and a set A of a variables, the following equivalence holds:*

$$\exists R \text{ s.t. } (H(S|_A, R|_A) = a \text{ and } H(S, R) < d)$$
$$\Leftrightarrow$$
$$\exists R' \text{ s.t. } (H(S|_A, R'|_A) = a \text{ and}$$
$$H(S|_{\Gamma_{d-a}(A)}, R'|_{\Gamma_{d-a}(A)}) = H(S, R') < d)$$

Proof. We prove this lemma constructively. We start from two solutions S and R that satisfy the premise of this implication and construct R' such that S, R' satisfy the conclusion. We have $H(S, R) = k_1 < k$, therefore exactly k_1 variables are assigned differently between S and R. We also know that $H(S|_A, R|_A) = |A| = a$ therefore only $b = k_1 - a$ are assigned differently outside A. Now we change R into R' in the following way. Let d be the smallest integer such that $\forall X_i \in \Delta_d(A),\ R[i] = S[i]$. It is easy to see that $d \leq b$ as $\Delta_{d_1}(A)$ and $\Delta_{d_2}(A)$ are disjoint iff $d_1 \neq d_2$. We let all variables in $\Gamma_d(A)$ unchanged, and for all other variables we set $R'[i]$ to $S[i]$. Now we show that R' satisfies all constraints. Without loss of generality, consider any constraint C_V on a set of variables V. By definition, the variables in V belongs to at most two sets $\Delta_{d_1}(A)$ and $\Delta_{d_2}(A)$ such that d_1 and d_2 are consecutive (or possibly $d_1 = d_2$). We have 3 cases:

1. $d_1 \leq d$ and $d_2 \leq d$: all variables in V are assigned as in R, therefore C_V is satisfied.
2. $d_1 > d$ and $d_2 > d$: all variables in V are assigned as in S, therefore C_V is satisfied.
3. $d_1 = d$ and $d_2 = d + 1$: the variables in $\Delta_{d_2}(A)$ are assigned as in S, and by definition of R', the variables in $\Delta_{d_1=d}(A)$ are assigned as in S, therefore C_V is satisfied.

□

Computing this neighborhood is not time expensive, as it can be done as a preprocessing step. A simple breadth first search on the constraint graph, i.e., the graph were any two variables are connected iff they are constrained by the same constraint. The neighborhood $\Gamma_d(A)$ of a break A is recomputed each time, however it just requires a simple union operation over the neighborhood of the elements in A.

Updates of the auxiliary CSP: The first use of Lemma **??** is straightforward. We know that, for a given break A, there exists a b-repair only if there exists one that sharea all assignments outside $\Gamma_b(A)$. Therefore, we can make P' equal to the current state of P apart from $\Gamma_b(A)$. This does not make the algorithm stronger. However, we can then post an AtMost$k$Diff constraint only on $\Gamma_b(A)$ instead of $\mathcal{X}'$, since we have all the pruning on $\mathcal{X}' \setminus \Gamma_b(A)$ for "free".

Avoiding useless repair checks: Now suppose that $\Gamma_{b+1}(A) \subseteq Past'$. Then we know that this break has already been checked at the previous level in the search tree, and the repair that we found has the property that assignments on $\Delta_{b+1}(A)$ are the same as in the current solution. Thus any extension of the current partial solution, is also a valid extension of this repair. Therefore we know that this repair will hold in any subtree, hence we do not need to check it unless backtracking beyond this point.

Tightening the AtMostkDiff *constraint:* Considering the property of Lemma **??**, we can tighten the AtMostkDiff constraint by forbidding some extra tuples. Doing this, we get a tighter pruning when doing arc consistency, while keeping at least one solution, if such solution exists. The first tightening is that all differences outside $\Gamma_b(A)$ are forbidden. But, we can do even more inference. For instance, suppose that for that we look for a 3-repair for the break $\{X'_1\}$ and that $\Delta_1(X'_1) = \{X'_2, X'_3\}, \Delta_2(X'_1) = \{X'_4\}, \Delta_3(X'_1) = \{X'_5, \}$, and the domains are as follows:

$$X'_1 = \{1, 2\}, X'_2, X'_3 = \{3, 4\}, X'_4, X'_5 = \{1, 2, 3, 4\}$$

Moreover, suppose that for the main backtracker, the domains are as follows:

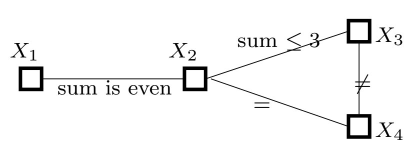
$$X_1 = 3, X_2, X_3, X_4, X_5 = \{1, 2\}$$

We can observe that already 2 reassignments are made at distance 1 from X_1. As a consequence, if X'_5 was to be assigned differently to X_5, then X_4 would have to be equal to X_4, and therefore there is no discrepancy on any variable from $\Delta_2(X'_1)$, hence there must be a repair such that any variable outside $\Gamma_2(X_1)$ is assigned as in the main solution. We can thus prune the values 3 and 4 from $\mathcal{D}(X'_5)$ (to make it equal to X_5).

Inference from repair-seeking to the main CSP: We can infer that some values of the main CSP will never participate in a super solution while seeking for repairs. This allows us to prune the future variables, which can greatly speed up the search process, especially when combined with GAC propagation on "regular" constraints. For instance consider constraint problem P, composed of the domain variables:

$$X_1 = \{1, 2, 4\}, X_2 = \{1, 2\}, X_3 = \{1, 2\}, X_4 = \{1, 2\}$$

subject to the following constraint network:



We have $P' = P$, and it is easy to see that P is arc consistent. Now suppose that we look for a $(1, 1)$-super solution, and our first decision is to assign the value 1 to X_1. The domains are reduced so that P remains arc consistent:

$$X_1 = \{1\}, X_2 = \{1\}, X_3 = \{1, 2\}, X_4 = \{1, 2\}$$

Then we want to make sure that there exist a 1-repair for the break $\{X_1\}$. We then consider P' where $\mathcal{D}(X'_1)$ is set to $\mathcal{D}(X'_1) \setminus \{1\}$. Moreover the constraint AtMost2Diff is posted on $\Gamma_1(\{X_1\}) = (X_1, X_2)$:

$$X'_1 = \{2,4\}, X'_2 = \{1,2\}, X'_3 = \{1,2\}, X'_4 = \{1,2\}$$

Since $P'|_{\{X'_1\}}$ is satisfiable, (for instance, $\{\langle X'_1 : 2\rangle\}$ is a partial solution that does not produce a domain wipe out in any variable of P') we continue searching. However, if before solving P' in order to find a repair we first propagate arc consistency, then we obtain the following domains:

$$X'_1 = \{2,4\}, X'_2 = \{2\}, X'_3 = \{1\}, X'_4 = \{2\}$$

Observe that $2 \in \mathcal{D}(X_3)$ whilst $2 \notin \mathcal{D}(X'_3)$, this means that no repair for X_1 can assign the value 2 to X_3. However, Lemma **??** works in both direction, since $X'_3 \notin \Gamma_1(X'_1)$, we can conclude that X'_3 and X_3 should be equal, and therefore we can prune the value 2 directly from X_3. In this toy example, this removal will make P arc inconsistent, and therefore we can conclude without searching that X_1 cannot be assigned to 1.

Notice that this extra pruning comes at no extra cost, the only condition that we impose is to make P' arc consistent *before* searching on it. After this arc consistent pre-processing for a break A, any value pruned from the domain of a variable $X'_i \in \mathcal{X}' \setminus \Gamma_b(A)$, can be pruned from X_i as well.

The main drawback of this method is that as soon as the problem involves a global constraint (for instance "all the variables must be different"), then typically we have $\Gamma_1(X_i) = \mathcal{X}$ for any $X_i \in \mathcal{X}$. Therefore all previous improvements based on neighborhood are useless. However, one can make such inference, but using a different reasoning, even in the presence of large arity constraints. The idea is the following: Suppose that after enforcing GAC on P', the least number of discrepancies is exactly $a+b$, that is, $Diff = \{i|\mathcal{D}(X_i) \neq \mathcal{D}'(X'_i)\}$ & $|Diff| = a+b$. We can deduce that any variable X'_j such that $j \notin Diff$ must be equal to X_j, for *any* b-repair. Indeed, it applies to any repair, since only pre-processing and no search was used. Therefore, we can prune domains in both P and P' as follows:

$$\forall i \notin Diff \ \mathcal{D}(X_i) \leftarrow \mathcal{D}(X_i) \cap \mathcal{D}(X'_i) \ \& \ \mathcal{D}(X'_i) \leftarrow \mathcal{D}(X_i)$$

We can therefore modify `reparability` by taking into account the previous observations (Algorithm **??**).

5 Extensions

In [**?**], the authors propose to extend or restrict super solutions in several directions to make them more useful practically. In scheduling problems, we may have restrictions on how the machines are likely to break, or how we may repair them. Furthermore, we have an implicit temporal constraint that forbid some

Algorithm 4: `repairability`$(P, S, Past, a, b)$:Bool

```
covered ← ∅;
foreach A ⊆ Past′ such that |A| = a and A ∉ covered and Γ_{b+1} ⊈ Past′ do
    foreach X_i ∈ A do
        D′(X′_i) ← D′(X′_i) − {S[i]};
    foreach X_i ∈ (𝒳 \ Γ_b(A)) do
        D′(X′_i) ← D(X_i);
    if ¬AC-propagate(P′) then return False;
    Diff ← {i | D(X_i) ≠ D′(X′_i)};
    if Diff = a + b then
        foreach X_i ∈ (𝒳 \ Diff) do
            D(X_i) ← D′(X′_i) ← (D′(X′_i) ∩ D(X_i));
    foreach X_i ∈ (𝒳 \ Γ_b(A)) do
        D(X_i) ← D′(X′_i);
    S′ ← solve(P′|_{Past′} + AtMost(|A| + b)Diff(Past′, S));
    if S′ = nil then return False;
    Diff ← {i | S′[i] ≠ S[i]};
    foreach A′ ⊆ Diff such that |A′| = a do
        covered ← covered ∪ {A};
```

reassignments. For instance, when a variable breaks, there are often restrictions on the alternative value that it can take. When the values represent time, then an alternative value might have to be larger than the broken value. Alternatively, or in addition, we may want the repair to be chosen among larger values, for some ordering. It may also be the case that certain values may not be brittle and so cannot break. Or that if certain values are chosen, they cannot be changed. This algorithm allows to express these restrictions (and many more) very easily, as they can be modelled as extra constraints on P'.

For instance to model the fact that an alternative value has to be larger than the broken value, one can post the unary constraint $X > S[X]$, where X is any variable involved in the break, and $S[X]$ is the value assigned to this variable in the main solution. Moreover, one can change the constraint AtMost$k$Diff itself. For instance, suppose that we are only interested in the robustness of the overall makespan, and we are solving a sequence of *deadline job shop*. One can extend the deadline of P' by a given (acceptable) quantity q, and omit the AtMost$k$Diff constraint. The resulting solution will be one such that no "break" of size less than or equal to a can result in a makespan increase of more than q.

Another concern is that the semantic of the super solution depends on the model chosen for solving a problem. For instance, an efficient way of solving the job shop scheduling problem is to search over sequences of activities on each resource, rather than assigning start times to activities. In this case, two solutions involving different start times may map to a single sequence. Therefore the semantic of a super solution is changed, a break or a repair implies a modification of the order of activities for a resource. It is therefore interesting to think of ways of solving a problem using a model whilst ensuring that the solution is a super solution for another model. If it is possible to channel both representations, then one can solve one model whilst applying the `reparability` procedure to the second model.

5.1 Optimization

Finding super solutions is still, and will certainly remain a difficult problem. One way to avoid this difficulty is to try to get a solution as close as possible to a super solution. We can then start from an initial solution found using the best available method, and then we improve its *reparability* with a branch and bound algorithm.

The (a, b)-*reparability* of a solution is defined as the number of combinations of less than a variables that are covered by a b-repair. In [**?**], the authors report that turning the procedure into a branch and bound algorithm that maximize reparability is the most promising way of using super solutions in practice as an (a, b)-super solution may not always exist. Moreover doing so, one can get a "regular" solution with the fastest available method, and improve its reparability afterward.

The algorithm introduced here can easily be adapted in this way. The procedure `reparability` would return the number of b-repairs founds, instead of failing when a break does not accept one. The main backtracker would then backtrack either if the problem is made arc inconsistent or the value returned by `reparability` is less than the reparability of the best full solution found so far. We rewrite the procedure `reparability` adapted to this purpose in Algorithm **??**.

Algorithm 5: `reparability`$(P, S, Past, a, b)$:Int

$covered \leftarrow \emptyset$;
foreach $A \subseteq Past'$ *such that* $|A| = a$ *and* $A \notin covered$ *and* $\Gamma_{b+1} \not\subseteq Past'$ **do**
  foreach $X_i \in A$ **do**
    $\mathcal{D}'(X'_i) \leftarrow \mathcal{D}'(X'_i) - \{S[i]\}$;
  foreach $X_i \in (\mathcal{X} \setminus \Gamma_b(A))$ **do**
    $\mathcal{D}'(X'_i) \leftarrow \mathcal{D}(X_i)$;
  $S' \leftarrow$ `solve`$(P'|_{Past'} + \textsc{AtMost}(|A| + b)\textsc{Diff}(Past', S))$;
  if $S' \neq nil$ **then**
    $Diff \leftarrow \{i | S'[i] \neq S[i]\}$;
    foreach $A' \subseteq Diff$ *such that* $|A'| = a$ **do**
      $covered \leftarrow covered \cup \{A\}$;
  return $|covered|$;

Unfortunately, if all other improvements still hold, we cannot prune P as a result of a pruning when pre-processing P', since a break without repair is allowed.

6 Future work

Our next priority is to implement this algorithm for finding super solutions to the job-shop scheduling problem. We will use a constraint solver that implements shaving, the ORR heuristic described in [**?**], as well as constant time propagators for enforcing GAC on precedence and overlapping constraints. As these two constraints are binary, the neighborhood of a variable is limited. Hence the theoretical results introduced here should apply. Moreover, we expect this reasoning to offer a good synergy with a strong global consistency method such as shaving. Indeed more inference can be done on the repairs (without searching) than with GAC, and thus more values can be pruned in the main search tree because of the robustness requirement. We therefore expect to be able to solve much larger problems than the instances solved in [**?**].

7 Conclusion

We have introduced a new algorithm for finding super solutions that improves upon the previous method. The new algorithm is more space efficient as it only requires to double the size of the original constraint satisfaction problem. The new algorithm also permits us to use any constraint toolkit to solve the master problem as well as the sub-problems generated during search. We also take advantage of repair multi-directionality and of inference based on just a restricted *neighborhood* of constraints. Moreover, this algorithm allows the user to easily specify extra constraints on the repairs. For example, we can easily specify that all repairs should be later in time than the first break.

Acknowledgements

Emmanuel Hebrard and Toby Walsh are supported by National ICT Australia. Brahim Hnich is currently supported by Science Foundation Ireland under Grant No. 00/PI.1/C075. We also thank ILOG for a software grant.

Solving the Magic Square Problem by Using Weak Symmetries

Roland Martin and Karsten Weihe

Darmstadt University of Technology
Algorithmics Group
64283 Darmstadt, Germany,
{martin,weihe}@algo.informatik.tu-darmstadt.de

Abstract. In this article we tackle the magic square problem and a variation of it by using a novel approach of symmetry, which we call *weak symmetry*.
Weak symmetries occur in different application fields like planning, scheduling and manufacturing as well as in extensions of classical problems.
In contrast to a proper symmetry a weak symmetry acts only on a subset of the variables and preserves the feasibility state only with respect to a subset of the constraints. Therefore, breaking weak symmetries with standard techniques would lead to a loss of solutions.
We discuss a modelling technique which is based on additional variables (we will call them *SymVar – Symmetry Variable*). These variables enable us to achieve symmetry breaking on the symmetric variables of the problem without losing solutions.
We discuss an approach that solves a relaxed version of the magic square first and then tries to find a solution for the original problem. We will also consider a variation of the magic square problem that introduces weights such that the problem becomes an optimisation problem. For both scenarios we will compare our approach to a standard approach and present computational results.

1 Introduction

Symmetries transform a (partial) solution into a symmetric (partial) solution and preserve the state of feasibility: no-goods are tranformed into symmetric no-goods while feasible solutions are transformed into symmetric feasible solutions. Therefore, symmetries decompose the search space into classes of symmetric solutions, whereby each class either contains feasible solutions only or infeasible solutions only.

When searching for all solutions to a problem it is sufficient to find only one solution in each class of solutions. The symmetric equivalents can be derived by applying the symmetry function exhaustively to each class after the search process. Therefore symmetries should be excluded from the search space to speed up the search.

Various techniques have been proposed for symmetry handling. In general it is done by reformulating the model, excluding the symmetry up-front via additional constraints, breaking it during the search or by a combination thereof.

Weak symmetries act only on a subset of the variables and satisfy only a subset of the constraints of the problem. Therefore, weak symmetries preserve the state of feasibility only with respect to the subset of variables they act on and only for the constraints they satisfy. This means that if two solutions are symmetric under the weak symmetry they yield different full solutions with potentially different feasibility states.

But weak symmetries cannot be simply broken, since this would result in a loss of solutions that cannot be derived afterwards. Nonetheless we will present a modelling technique that enables us to deal with weak symmetries such that they can be broken without losing solutions.

Our technique transforms the weak symmetry into a proper symmetry. Therefore, all above mentioned kinds of symmetry breaking are possible once the technique is applied.

Although we can break the weak symmetry on the problem that does not automatically result in a speed-up. Our approach needs more variables and constraints such that it cannot be predicted whether using our approach is fruitful for every scenario. Therefore we will discuss the benefits and drawbacks of this approach and present results that show the efficiency of using SymVars to break weak symmetries.

Weak symmetries occur in many fields of applications and have already been discovered and identified in planning, scheduling and model checking.(See [1] - [8]). Also extensions to classical problems like the rack configuration problem (see [9] and [10]) contain weak symmetries. To our best knowledge up to now there is no approach that tackles the problem by modelling. But in [4] the problem is considered from a group theoretical background.

This article introduces a modelling approach that addresses the problem. The benefit in solving the problem by modelling is that it can be applied without changing or adjusting the solver used for the problem or write further code to tackle the problem. Furthermore one is not restricted to a specific solver when using the modelling approach. And as mentioned before, all other kinds of symmetry handling are possible in conjunction with this approach.

We would like to stress that our intention is not to present a most efficient approach to solve the magic square problem (there are several CP techniques (See [5] and [6]) as well as constructive algorithms (See [7]) that solve the problem efficiently) but to show that weak symmetries have an impact on problems where they occur. We chose the magic square problem since it is well-known and researched. It also gives the opportunity to evaluate our approach very easily with a standard approach such that the comparison of the approaches is fair.

Section 2 gives the definitions on weakly decomposable problems, weak symmetry and how SymVars are applied. In Section 3 we present our model for solving the magic square by the SymVar approach and compare it with a standard approach. We also present computational results. In Section 4 we introduce a relaxation of the magic square problem that is an optimisation problem. Again we compare our model with a standard approach and present computational results. Section 5 concludes the computational results and gives an outlook to future ideas and work.

2 Weak Symmetry Description

2.1 Prerequisites

We characterize a satisfaction problem by $P = (X, C)$, whereby $X = \{x_1, \ldots, x_n\}$ is the set of variables and $C = \{c_1, \ldots, c_m\}$ is the set of constraints.

For an optimisation problem we just extend this formulation to $P = (X, C, f)$, where X and C are defined as above and f is the objective function. [1]

A solution to P is denoted by $s_P = (x_1 = v_1, \ldots, x_n = v_n) = (X)$. This means that each variable in X is assigned a value of its corresponding domain.

In the following we introduce weak symmetries. We will consider a reformulation of the magic square problem that follows the idea of weak symmetries. The new formulation is introduced bit by bit with the definitions in this section.

In the magic square problem a $n \times n$ matrix with entries from 1 to n^2 is sought such that the sum of the entries in each row, column and the main diagonals has the same value. This value is called the magic number m of the square. For each square of order n this number is computed by $m = \frac{n^3+n}{2}$. The symmetries acting on the problem are the eight symmetries of rotations and reflections of the matrix.

2.2 Weak Symmetry Definition

Weak symmetries act on problems with special properties. To characterise weak symmetries we first define weakly decomposable problems. The goal of this decomposition is to have all variables and constraints that are respected by the weak symmetry in one sub-problem and the asymmetrical data in an other sub-problem.

Definition 1 (Weakly Decomposable Problem)
A problem $P = (X, C)$ is **weakly decomposable** *if it decomposes into two subproblems $P_1 = (X_1, C_1)$ and
$P_2 = (X_2, C_2)$ with the following properties:*

$$X_1 \cap X_2 \neq \emptyset \tag{1}$$
$$X_1 \cup X_2 = X \tag{2}$$
$$C_1 \cap C_2 = \emptyset \tag{3}$$
$$C_1 \cup C_2 = C \tag{4}$$
$$C_2 \neq \emptyset \tag{5}$$

(1) states that P_1 and P_2 contain a subset of shared variables (namely $X_1 \cap X_2$). These variables have to assume the same values in both subproblems to deliver a feasible solution to P. Therefore they link both problems. Without that restriction the problem would be properly decomposable. (2) and (4) states that none of the variables and constraints of the original problem P are lost. Furthermore (3) and (4) state that C_1 and

[1] Note that in standard CP techniques the objective function f can be represented by a constraint that tightens with the search. The objective value is then represented by a variable. Therefore, it is sufficient to regard only satisfaction problems.

C_2 is a partition of C. Basically this is not necessary for feasibility. A constraint could be in both subsets (if defined on $X_1 \cap X_2$ only) but would be redundant for one of the problems because the solution to the other subproblem would already satisfy this constraint. Therefore, this is just a question of efficiency. The last property states that P_2 is not allowed to be unconstrained. But note that this restriction does not hold for P_1, since we consider the weak symmetry acting on P_1 and a problem without constraints is perfectly symmetric.

An another example (besides the magic square problem) for a weakly decomposable problem is the magic knight tour. (See [11] and [12]). In this problem a knight tour on a chessboard is sought for where the numbers of the moves constitutes a magic square. The weakly decomposition is that P_1 consists of the magic square problem and P_2 constitutes that – when following the numbers from 1 to n^2 – this is a knight tour.

In the magic square problem:
The magic square problem is weakly decomposable. P_1 consists of the matrix as the variables and the constraints that state that the sum of the rows and columns equals the magic number. P_2 consists just of the constraints that the sum of the main diagonals is the magic number. Let $N = \{1, \ldots, n\}$, whereby n is to length of the square.

$$
\begin{aligned}
P_1\colon X_1 &= \{square_{ij}, i, j \in N\} \\
C_1 &= \{alldifferent(square), \\
&\qquad \textstyle\sum_{r \in N} square_{rc} = m, c \in N, \\
&\qquad \textstyle\sum_{c \in N} square_{rc} = m, r \in N \\
&\qquad \}
\end{aligned}
$$

$$
\begin{aligned}
P_2\colon X_2 &= \{\} \\
C_2 &= \{\textstyle\sum_{i \in N} square_{ii} = m, \\
&\qquad \textstyle\sum_{i \in N} square_{i,n+1-i} = m \\
&\qquad \}
\end{aligned}
$$

A symmetry that acts on the subproblem P_1 (but not on P_2) is considered a weak symmetry.

Definition 2 (Weak Symmetry)
Given a weakly decomposable problem P with a decomposition (P_1, P_2).
A symmetry S on P_1 is called a **weak symmetry** *on P with respect to the decomposition (P_1, P_2) iff S acts on P_1 but not on P_2.*

The intention of the decomposition of the problem is that X_1 contains all symmetric variables (and only these) and X_2 contains also the rest of the variables.
The gain is that we get a subproblem that is not affected by the weak symmetry (P_2) and a subproblem where the weak symmetry affects all variables and all constraints (P_1).

In the magic square problem:
The weak symmetries acting on the problem are the row and column permutations of the square. That means that every row and/or column permutation of a solution to P_1 constitutes a new symmetric solution that satisfies also the constraints of P_1. Therefore, the weak symmetry acts as a normal symmetry on P_1 but not on P_2. In addition some of the permutations may also satisfy the constraints stated in P_2 (but most will not do). That means that a solution to P_1 yields both feasible and infeasible solutions for the problem P.[2] For simplicity we only regard the column permutation at the moment. This is totally valid which is shortly explained in 3.1.

2.3 Weak Symmetry Breaking

Since the weak symmetry does not act on the whole problem, it cannot be broken on the whole problem. But it can be broken in P_1 (where it acts as a proper symmetry). This means that in the search tree equivalent solutions of P_1 are identified with each other. On the other hand we would lose the symmetric solutions if we broke the symmetry. Therefore we need a way to represent these solutions explicitly because they are needed in order to solve P_2 which delivers a full solution for P.

In order to represent these symmetric solutions we introduce additional variables called *SymVars* and state constraints on these variables. These variables encode the symmetric equivalents of a solution. We consider these variables and constraints to form a new sub-problem $P_{sym} = (X_{sym}, C_{sym})$, whereby X_{sym} consists of the SymVars and C_{sym} consists of the constraints that state the feasible symmetric solutions.

Notation 1 *Given a weakly decomposable problem P with a decomposition (P_1, P_2), $P_1 = (X_1, C_1), P_2 = (X_2, C_2)$.*
Further given a set of SymVars $X_{sym} = \{y_1, \ldots, y_\ell\}$ that constitutes the variables of the subproblem P_{sym}.
A solution to P_1 is denoted by $s_{P_1} = (X_1)$.
A solution to P_{sym} is denoted by $s_{P_{sym}} = (X_1, X_{sym}) = (s_{P_1}, X_{sym})$
A solution to P_2 is denoted by $s_{P_2} = (X_1, X_{sym}, X_2) = (s_{P_{sym}}, X_2)$.
A solution to P_2 is automatically a solution to P.
Let $s_{P_1} = (v_1, \ldots, v_n)$ be a solution to P_1, where v_i is a value of the domain of x_i, $i \in N$.
A solution $s_{P_{sym}} = (s_{P_1}, v'_1, \ldots, v'_\ell)$ is a symmetric solution to s_{P_1}, where v'_j is a value of the domain of y_j, $j \in \{1, \ldots, \ell\}$

The solving order now is to search a solution s_{P_1} to P_1, determine a symmetric equivalent $s_{P_{sym}}$ in P_{sym} and use this solution to determine a solution to P_2 which already states a solution to P. Note that when considering P_{sym} the variables X_1 are already assigned. The same holds for P_2 with X_1 and X_{sym}.

Consequences:

[2] In Section 4 we will consider a problem where all weakly symmetric solutions of P_1 are feasible for P_2 but have different objective values.

- Every feasible value assignment to the SymVars constitutes a symmetric solution to s_{P_1}
- None of the values in X_1 have to be reconsidered to receive a symmetric equivalent
- The symmetry can be broken in P_1 because all symmetric solutions to S_{P_1} are expressed by $s_{P_{sym}}$

In the magic square problem:

For each column of the square a SymVar is introduced. An assignment $SymCol_i = j$ means that the i-th column is permuted to the j-th column.

$$\begin{aligned}
&P_{sym}:\\
&X_{sym} = \{square_{ij}, i, j \in N,\\
&\qquad\quad SymCol_i, i \in N\\
&\qquad\quad \}\\
&C_{sym} = \{alldifferent(SymCol)\}
\end{aligned}$$

General solving order:

To solve P we consider the partial solution $s_{P_{sym}}$. When a solution to P_2 is found the search backtracks and reconsiders values for the SymVars to determine a new solution. All these solutions are symmetric equivalents to the solution s_{P_1}. Only when the search backtracks and reconsiders variables in X_1 a solution for a different equivalence class can be found.

2.4 Related Work

The intention of this article is not to show a technique to solve the magic square problem most efficiently as already stated before. We just use it as a nice problem to show the effects of weak symmetries and the application of SymVars. Therefore we discuss related work in terms of work on weak symmetries only.

There is an approach by W. Harvey [4] that regards the problem of weak symmetries from a group theoretical point of view. Basically the problem is regarded as a symmetric relaxation. The idea is to relax some constraints on the problem which makes the problem more symmetric. The symmetry is broken on the symmetric relaxation and symmetric solutions are then derived by using the software package GAP [15].

The ideas and theory behind both approaches (the one considered in this article and that of W.Harvey) are basically the same. The symmetric relaxation approach is somewhat more generic allowing more freedom in the realisation and implementation of the techniques used to solve the problem. Therefore it seems very promising. The difference of the two approaches is the way the symmetric solutions are considered and generated.

At the moment a comparison between the two approaches is not possible. But this is subject to future joint work for the authors and W. Harvey.

The strength in our approach is that it is based on plain modelling. That means that it can be used for every constraint programming solver and does not require software or self-written routines to adapt a problem suitably. Nonetheless the approach is open to in-cooperate own code. By doing so the efficiency of the approach could prosper substantially. But that is subject to future work.

2.5 Application Fields of the Weak Symmetry Approach

Weak symmetries can be found in satisfaction problems as well as in optimisation problems. There are standard problems like the magic square problem that have weak symmetries with the proper decomposition. Also there are extensions of standard problems that have weak symmetries. [3]

Following the weakly decomposable problem definition (where weak symmetries are based on) a problem that has weak symmetries can be regarded as a core problem with an extension. The core problem is P_1 and the extension is P_2 in our definitions.

Extending a problem often results in an artificial problem (whereby the core problem itself is somewhat artificial). Nonetheless there are a lot of real-world problems that comprise weak symmetries.

An example is a line-balancing problem (See [13],[3]) from automated manufacturing. In this problem the task is to find a setup for several mounting machines in a mounting-complex such that the visibility of the machines resulting by this setup is maximised. The core problem is to find the setup and the "extension" is to maximise the visibilities.

The results of solving this problem by using SymVars (See [3]) justify the approach.

3 Solving Magic Squares using SymVars

Since the problem is already introduced in the last section we just state the complete remodelling of the problem in the following. Nonetheless the subproblems are the same:

P_1: Find a magic square without the diagonal constraints
P_{sym}: Consider a permutation of the magic square
P_2: Check whether the permuted magic square also respects the diagonal constraints.

Note that the model is not explicitly split. All the variables and constraints are included. Only the implicite variable ordering determines the subproblems.

We will state two variations. In the first variation only the column permutation is weak and in the second variation column and row permutations are weak. In both approaches we break the symmetries of rotations and reflections of the matrix since we do this also in the standard model.

For comparison with our models we consider a standard modeling of the problem that already breaks the symmetries of rotations and reflections of the matrix.

3.1 Re-modelling of the Magic Square Problem Using SymVars - Column Permutation is Weak

In this scenario the SymVars enable us to consider column permutations of the solution determined by P_1. Note that the row symmetry is also weak. But we do not consider this here. This is perfectly valid. If we do not respect this weak symmetry we just lose the possibility of breaking the symmetry in P_1. On the other hand, we do not need further

[3] We will discuss such an extension in Section 4.

SymVars since breaking weak symmetries basically means to introduce SymVars. The next scenario will deal with the row permutation accordingly.

P:

$$
\begin{aligned}
&X = \{square_{ij}; i, j \in N,\\
&\quad SymCol_i; i \in N\\
&\quad \}\\
&C = \{\\
&\quad alldifferent(square)\\
&\quad \textstyle\sum_{r \in N} square_{rc} = m, c \in N\\
&\quad \textstyle\sum_{c \in N} square_{rc} = m, r \in N\\
&\quad alldifferent(SymCol)\\
&\quad \textstyle\sum_{i \in N} square_{i,SymCol_i} = m, i \in N\\
&\quad \textstyle\sum_{i \in N} square_{i,SymCol_{n+1-i}} = m, i \in N\\
&\quad \}
\end{aligned}
$$

The first three constraints are adapted from P_1, the fourth is adapted from P_{sym} and the last two constraints are originally from P_2 but each entry of a column i in these constraints is replaced by its symmetric representative, the SymVar $SymCol_i$.

Weak Symmetry Breaking Since all column permutations of a solution to P_1 are performed by assignments of the SymVars this symmetry can be broken on P_1 without losing solutions.

Efficiency Discussion The task to find a solution to P_1 is easier in this scenario since the diagonal constraints do not have to be taken into account. In addition the weak symmetry of the column permutation can be broken which has an impact on the search space.

Whenever a solution to P_1 is found column permutations are considered to find a magic square that respects also the diagonal constraints. This does not necessarily mean to check all $n!$ permutations since propagation will prune the domains of the SymVars as well. When a permutation is found such that the constraints of P_2 are respected a solution to P is found. Backtracking considers a new permutation and if no other is feasible a new assignment for the matrix is sought and again permutations are considerd.

Since not every solution of P_1 yields a solution that satisfies P there will be no feasible permutation for such a solution. In this case the work done in P_{sym} is fruitless and effectively a waste of time. But to our knowledge there is no simple indicator that determines when a solution of P_1 also yields at least one magic square. [4]

It is not clear upfront whether the gain in symmetry breaking does outweigh this drawback. But our computational results (See 3.3) show encouraging results.

[4] It would be possible to reason on the magic sums of the square and the distribution of the numbers of a magic sum to the square. Magic sums are the building blocks of a magic square (See [7] for more about magic sums). There have to be at least two magic sums which have individually no two entries in the same row or column such that a permutation satisfies the diagonal constraints.

When P_1 is solved with the SymVar approach there is no constraint propagation on the diagonal constraints since they are also defined over the SymVars (and they are not instantiated in P_1).

Example: In the magic square problem of order 4 the magic number is 34. A partial assignment like the following would be considered infeasible and abandoned by the standard approach:

16	x	x	x
x	15	x	x
x	x	10	x
x	x	x	x

It is not infeasible for the SymVar approach since it may yield valid assignments to the whole matrix such that a column permutation satisfies the main diagonal constraints.

The efficiency problem in using SymVars in the magic square problem is that not every solution of P_1 yields at least one feasible solution of P.[5]

Consider the following variable assignment for a square of order 4:

1	4	14	15
5	8	10	11
12	9	7	6
16	13	3	2

This assignment does not result in a magic square by a column permutation.

The best efficiency of the SymVar approach is observed when every solution of P_1 yields only feasible solutions.

3.2 Re-modelling of the Magic Square Problem Using SymVars - Column and Row Permutations are Weak

Additionally to the SymVars for the columns we introduce SymVars for the rows as well. Therefore the constraints for the diagonals change such that each entry of a row j in these constraints is replaced by the symmetric representative, the SymVar $SymRow_j$.

In this scenario row and column permutations of the matrix are possible. Moreover a combination of row and column permutations is possible. That enables us to break even more symmetries in P_1.

P:

$X = \{$

$\quad square_{ij}; i, j \in N,$

$\quad SymCol_i; i \in N$

$\quad SymRow_i, i \in N$

$\quad \}$

[5] In Section 4 we will see a relaxation of the problem where all solutions to P_1 result in solutions to P.

$$
\begin{aligned}
&C = \{\\
&\quad alldifferent(square)\\
&\quad \textstyle\sum_{r \in N} square_{rc} = m, c \in \{1, \ldots, n\}\\
&\quad \textstyle\sum_{c \in N} square_{rc} = m, r \in N\\
&\quad alldifferent(SymCol)\\
&\quad alldifferent(SymRow)\\
&\quad \textstyle\sum_{i \in N} square_{SymRow_i, SymCol_i} = m, i \in N\\
&\quad \textstyle\sum_{i \in N} square_{SymRow_i, SymCol_{n+1-i}} = m, i \in N\\
&\quad \}
\end{aligned}
$$

Weak Symmetry Breaking In addition to the column symmetry the row symmetry can be broken in P_1. Therefore the solution space is narrowed again.

Efficiency Discussion The gain in efficiency with this model comes from additional symmetry breaking. Also former infeasible solutions may now be feasible since a combination of row and column permutations may yield a feasible solution. Therefore we increased the chance that a solution to P_1 can be extended (by a permutation) to a feasible solution to P.

But again there are solutions to P_1 that cannot be extended to a feasible solution to P. The example in the last section can also be applied here. For the matrix on Page 9 there is no column and/or row permutation such that the permuted matrix is a magic square.

For such matrices the overhead is even higher since column and row permutations are considered. In large instances this could lead to a great amount of overhead.

3.3 Computational Results

We considered magic squares of order three to six with a time limit of thirty minutes. For larger instances there are no solutions within this time limit. We compare the standard approach with the two approaches introduced in the section.

The instances of order three and four are too small to show any significant differences for the approaches. Since instances of order higher than six are not capable within the time limit our basis of results is very small. New experiments have to be done with a larger time limit. Due to time restrictions there was no possibility to test the instances with a larger time limit. But first test indicate that the limit would have to be set to several hours if not days. So we consider the computational results in this very section not complete and subject to future work. Nonetheless a larger time limit will not change these results considerably.

We used ILOG OPL Studio 3.51 (See [14]) for the computation on a computer equipped with an AMD XP processor with 2 GHz and 1280 MB RAM.

Order	Approach	First Sol	# Sol Found
5	Standard	0.031	140400
	SV Col	0.016	71892
	SV Col+Row	0.031	107896
6	Standard	110.1	21747
	SV Col	0.156	195200
	SV Col+Row	26.82	32900

Order states the order of the magic square. *Approach* states which approach was used. *First Sol* gives the time in seconds when the first solution was found. *# Sol Found* markes how many solutions have been found within the time limit of thirty minutes.

Conclusion The results are quite interesting. Our first feeling was that this would be an disadvantage scenario for weak symmetries since too much search is wasted in solutions to P_1 that yield no feasible solution to P.

But this is only partly true. The first solution was found in the same time basically up to order five. For the instance of order six both our models find a solution earlier. The SV Col approach is most efficient in this scenario. While the SV Col+Row approach finds a solution four times faster than the standard approach, the SV Col approach finds a solution about sevenhundred times faster.

When comparing the number of solutions found the standard approach performs better in the instance of order five but is worse for order six. Also the SV Col approach performs worse than the SV Col+Row approach for order five but this is changed for order six.

Our guess is that the SV Col+Row approach loses too much time with considering solutions of P_1 that are infeasible for P. Therefore a rather small part of the search space of P_1 can be investigated.

Although it was not part of the research we detected that restricting the problem even more by forcing that the diagonals are ordered increasingly have interesting effects. We found solutions of magic squares of higher orders that could not be found by the standard approach even with the same symmetry breaking constraints imposed. In some cases the standard approach indicated that there is no solution (contradiction in the constraints). With the SymVar approach we found the first solution for the magic square with order seven in 10.4 seconds and for the magic square with order eight in 0.85 seconds. Since we restrict the solution space in a way that solutions are lost, this is a completely different scenario. Although these results seems interesting and it seems that magic squares with more restrictions (like diagonally ordered magic squares) could be found easier with our approach this is beyond the scope of this article. But this is subject to future work.

4 Weighted Magic Square – An Optimisation Relaxation

In this section we consider a relaxation of the magic square problem. First we totally relax the diagonal constraints. Additionally we state a weight for each column of the matrix. The numbers in a column are now weighted. The objective is to maximise the total weight of the matrix. A column permutation of the matrix yields now a different total weight.

We regard the scenario of optimisation within a given time limit. and compare the standard approach with our SymVar approach with SymVars for the columns.

Since the problem definition now yields that all symmetric solutions may have different weights the rotation and reflection symmetries cannot be broken without losing solutions.

The SymVar approach performs considerably good in this scenario. This is due to the fact that every solution to P_1 yields a feasible solution to P. Moreover, all the permutations are feasible.

4.1 Model of the Weighted Magic Square Problem

Basically we use the same models as stated in Section 4. The difference is that the diagonal constraints are totally relaxed and instead an objective function is introduced. We have a weight for each column denoted by $weight_i, i \in N$ and the objective function is represented by a variable $ObjVal$. The objective value is computed by the sum of the product of each column multiplied by the weight on the column:

$ObjVal = \sum_{col \in N} weight_{col} \prod_{row \in N} square_{row,col}$

Since the products of the columns are different for different permutations they yield different solutions due to the imposed weights.

For our approaches SV Col and SV Col+Row the objective function is adapted such that the occurences of the rows and columns are substituted by the according SymVar. Which leads for example in the SV Col+Row approach to the computation:

$ObjVal = \sum_{col \in N} weight_{col} \prod_{row \in N} square_{SymRow_{row}, SymCol_{col}}$

Symmetry Breaking The difference to the problem considered in Section 4 is that every magic square yields a different objective value. Therefore the rotation and reflection symmetries cannot be broken in the standard problem. In the SymVar approaches the according weak symmetries can be broken (Note that this does not include the rotation and reflection symmetries).

Efficiency Discussion In this scenario every solution to P_1 yields only feasible solutions for P. They just differ by their objective value. Therefore the SymVar approach is highly effective. The symmetry breaking on P_1 reduced the search space considerably and only feasible solutions are considered.

The drawback for the standard approach is that no symmetry breaking is possible. An alternative would be to solve the standard approach with symmetry breaking and store the solutions. Afterwards the objective value for the symmetric solutions are computed.

But this is not possible for the scenario that we are given a time limit for computation since the problem would have to be solved exhaustively before a first objective value is computed.

Since the number of solutions increases exponentially with each order the scenario of optimisation within a time limit seems more than adequate. Even for a small magic square of order six there are more than $1.77 \cdot 10^{19}$ solutions (which respect the diagonal constraints!) (See [7]).

4.2 Results

Again we considered squares of order three to six within a time limit of one hour. For higher orders we did not find results within the time limit.

We used ILOG OPL Studio 3.51 (See [14]) for the computation on a computer equipped with a AMD XP processor with 2 GHz and 1280 MB RAM.

Order	Approach	First Sol	Time	Best Sol	Time	Solved
4	Standard	53064	0.015	71112	9.0	75.89
	SV Col	42036	0.001	71112	3.14	21.62
	SV Col+Row	44123	0.016	71112	3.21	7.01
5	Standard	3158736	0.016	4877544	2041	–
	SV Col	2902584	0.016	4918329	251.4	–
	SV Col+Row	2781549	0.031	4813212	3250.5	–
6	Standard	347647008	0.32	422148768	2889	–
	SV Col	329923720	0.43	407033968	1192	–
	SV Col+Row	336178248	0.67	445325878	528.5	–

Again *Order* and *Approach* marks the order of the magic square and the used approach. *First Sol* and *Time* states the objective value of the first found solution and the time when it was found. *Best Sol* and *Time* states the best solution found within the time limit and when it was found. *Solved* states how much time was needed to solve the problem exhaustively.

Conclusion In all instances the objective value of the first solution was higher for the standard approach. But what cannot be seen in this statistic is that the increase from solution to solution is rather small for this approach and mostly the second or third solution of the SymVar approaches (found within milliseconds) does outperform this value and dominate it for a considerable amount of time.

The first solution is found in all approaches within the first second with not too much difference between the approaches. Within the time limit the best solution comes from one of the SymVar approaches whereby this solution is found long before the standard approach found its best solution.

Unfortunately only the instance of order four shows the reduction in the search space. An optimum is found with the SymVar approaches in a third of the time it took the standard approach. In addition the SV Col approach solves the problem entirely 3.5 times faster and the SV Col+Row approach about ten times faster than the standard approach. This cannot be observed for instances of higher order since the time limit is much too small. Several hours if not days would be necessary.

Again there is no winner from the two SymVar approaches. Therefore it cannot be predicted which approach performs better on the long run. What was not applied in this tests are ordering heuristics for the permutations. Based on the weights it is possible to order the SymVar instantiations such that better solutions are found earlier. First results (that are not included here) indicate that this increases the quality of the first found solution considerable. This is even more interesting for magic squares of higher order since better propagation on the SymVars can be performed which saves time.

5 Conclusion and Outlook

We presented two different scenarios of the magic square problem solved by our approach and compared with a standard approach. The first scenario was the standard magic square problem with a weakly decomposition. In this scenario the gain in using SymVars can be observed but due to the complexity of the problem only small instances could be tested. The gain in this scenario when using SymVars is that the first feasible solution is found earlier by this approach than by the standard approach. Therefore it suits the idea of constraint satisfaction where just one solution is sought but fast.

The second scenario was a relaxation of the magic square combined with introducing weights on the columns of the matrix. In this approach the SymVar approach performs considerably good since every solution of P_1 yield only feasible solutions to P. Here symmetry breaking achieves a gain in efficiency.

Although the magic square problem was just chosen for reasons of simplicity it would be very interesting to investigate more complex structures of magic squares like ultramagic or diagonally ordered magic squares using SymVars. Our feeling is that using SymVars is increasing effectiveness if the original problem P is getting harder. In this case the gain in symmetry breaking should pay off. But this is still subject to more detailed research.

References

1. Peter Gregory *Almost–Symmetry in Planning* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005.
2. Alastair Donaldson *Partial Symmetry in Model Checking* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005.
3. Roland Martin *Approaches to Symmetry Breaking for Weak Symmetries* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005.
4. Warwick Harvey *Symmetric Relaxation Techniques for Constraint Programming* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005.
5. Carla Gomes and Meinolf Sellmann *Streamlines Constraint Reasoning* 10th International Conference on Principles and Practice of Constraint Programming,(CP04), Toronto, Canada, 2005.
6. Iain McDonald *NuSBDS: Symmetry Breaking made Easy* SymCon'03: Third International Workshop in Constraint Satisfaction Problems, Kinsale, Ireland, 2003.
7. W. Trump *How many magic squares are there?* Found at: *http://www.trump.de/magic-squares/howmany.html*, 2004.

8. Warwick Harvey *The Fully Social Golfer Problem* SymCon'03: Third International Workshop in Constraint Satisfaction Problems, Kinsale, Ireland, 2003.
9. Z. Kiziltan, B. Hnich *Prob031: Rack Configuration Problem* Found at *http://4c.ucc.ie/ tw/csplib/prob031*,2003.
10. I. Gent, T. Walsh, B. Selman *CSPLib: a problem library for constraints* Found at *http://4c.ucc.ie/ tw/csplib*, 2003.
11. Hosted by Guenter Stertenbrink *Computing Magic Knight Tours* Found at *http://magictour.free.fr*, 2003.
12. Compiled by George Jelliss *Knight's Tour Notes* Found at *http://www.ktn.freeuk.com*, 2003.
13. Rico Gaudlitz *Optimization Algorithms for Complex Mounting Machines in PC Board Manufacturing* Diploma Thesis, Darmstadt University of Technology, 2004.
14. V. Hentenryck, I. Lustig *The OPL Optimization Programming Language* The MIT Press.,1999.
15. The GAP Group *GAP - Groups, Algorithms, and Programming, Version 4.3, 2002* Found at *http://www.gap-system.org*, 2003.

On Generators of Random Quasigroup Problems

Roman Barták*

Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, Prague, Czech Republic
roman.bartak@mff.cuni.cz

Abstract. Random problems are a good source of test suites for comparing quality of constraint satisfaction techniques. Quasigroup problems are representatives of structured random problems that are closer to real-life problems and hence might be more suitable for benchmarking. In this paper, we describe in detail generators for Quasigroup Completion Problem (QCP) and Quasigroups with Holes (QWH). In particular, we study an improvement of the generator for QCP that produces a larger number of satisfiable problems by using propagation through the all-different constraint. We also re-formulate the algorithm for generating QWH that is much faster than the original generator. Last but not least, we provide an experimental comparison of all presented generators.

1 Introduction

Generators of random problems are a useful source of problem instances for testing constraint satisfaction algorithms. Writing generators for some types of problems, like Random CSP [11], is not a complicated task but it could be more complicated for other types of problems, typically for structured problems like quasigroup problems. This paper gives all necessary information for researchers that would like to use quasigroup problems as benchmarks.

The quasigroup problems have been first proposed as a benchmark domain for constraint satisfaction algorithms in [6]. The basic idea is to find a completion of a partial Latin square representing the multiplication table of a quasigroup. Hence, they called the problem a *Quasigroup Completion Problem* (QCP). The generator for a QCP should produce a partial Latin square that can be completed to a full Latin square. However, the generator proposed in [6], which fills random values in randomly selected cells of the table, falls short on this task especially when more values should be filled in. Gomez and Selman observed a behavior of the generator similar to phase transition with satisfiable instances on one side, unsatisfiable instances on the other side, and hard instances in between. Shaw et al. [14] proposed an improvement of this generator based on propagation through the all-different constraint [13]. Their algorithm generates a larger number of satisfiable instances that can be used for testing solvers. It preserves the phase transition behavior but it generates satisfiable instances on both sides and it makes the phase transition crispier.

* Supported by projects 1M0021620808 and MSM0021620838 of the MŠMT.

The difficulty of QCP generators is that they do not guarantee production of satisfiable instances only. This complicates usage of such generators for testing incomplete solving algorithms because when the solving algorithm did not find a solution, it is not clear whether no solution exists or the algorithm is not able to find it. In the first case (no solution exists) the algorithm can be "glorified" for doing a good job, in the second case, the algorithm can be blamed for being incomplete. Therefore another benchmark domain based on quasigroups has been proposed in [1] that guarantees generation of satisfiable instances. This benchmark domain uses the same idea as a QCP, that is completing a partially filled Latin square, but it differs in how the incomplete Latin square is obtained. The idea is to punch holes into a randomly generated complete Latin square so the obtained partial Latin square can surely be completed. Hence, this benchmark domain is called *Quasigroups With Holes* (QWH). Unfortunately, the authors of QWH did not provide all the details on generating QWH problems. It is a pity because generating randomly distributed QWH problems is a non-trivial task based on strong theoretical results presented in [9].

The contribution of this paper is threefold. First, we will give all the details on algorithms for generating random instances of QCP and QWH problems so the interested reader will be able to write his or her own generator based on the presented algorithms. Second, we will propose a reformulated algorithm for generating QWH problems that is significantly faster then the original algorithm from [9]. Last but not least, we will present an empirical comparison of all presented generators so the readers can select one that suits best their needs.

The paper is structured as follows. In the next section, we will introduce the terminology on quasigroups and Latin squares. In Section 3, we will describe the Quasigroup Completion Problem (QCP) and its relevance to real-world problems and we will discuss two generators of QCP. In Section 4, the ideas behind Quasigroups With Holes (QWH) will be explained, the original QWH generator will be presented in details, and the reformulated generator will be introduced. The paper is concluded by an experimental evaluation of the generators where we will compare quality and time efficiency of the generators.

2 Quasigroups and Latin Squares

A *quasigroup* is an ordered pair (Q, $\bullet$), where Q is a set and $\bullet$ is a binary operation on Q such that the equations $a \bullet x = b$ and $y \bullet a = b$ are uniquely solvable for every pair of elements a, b in Q. The cardinality of the set Q is called an *order* of the quasigroup. Let N be the order of the quasigroup Q then the multiplication table Q is a table of size N×N such that the cell at the coordinates (x,y) contains the result of the operation $x \bullet y$ (for simplicity reasons we expect Q to be a totally ordered discrete set and so the rows and columns of the multiplication table can be indexed by the elements of Q). The multiplication table of the quasigroup must satisfy a property that in each row of the table, each element of the set Q occurs exactly once, and similarly in each column of the table, each element of Q occurs exactly once (see Figure 1A). Thus, the multiplication table defines a *Latin square*.

We say that a Latin square of order N is *partial* or *incomplete* if the table of size N×N is partially filled in such a way that no symbol occurs twice in a row or in a column (see Figure 1B). If the table is filled completely then we are speaking about a *complete Latin square*. Note that it is easy to generate a complete Latin square of any order. First, we take some permutation of the elements in Q. Second, we fill the first row of the table with this permutation. Third, in each subsequent row, we shift the permutation one element to the right and the superfluous element on the right is filled in the first cell of the row (see Figure 1C). However, this method does not produce every Latin square. In fact, generating any Latin square of a given order with a uniform probability is a non-trivial task [9].

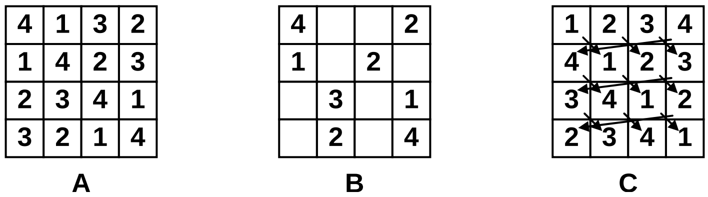


Fig. 1. A Latin square (A), a partial Latin square (B), and a simple process of generating a complete Latin square (C)

The problem of finding a complete Latin square can be stated as a constraint satisfaction problem in the following way. Assume, that the cells of a Latin square of order N are denoted by the variables with the domain {1,…,N}. Then the property of the Latin square can be described by a set of binary inequality constraints posted between every pair of variables that are either in the same row or in the same column. The constraint network for this CSP has N^2 nodes representing the variables and $N^2(N-1)$ edges representing the binary constraints. The network is highly structured – there are 2N interconnected clusters of size N (each cluster connects the variables from a single row or a single column). Moreover, there exists a path of maximal length two between any two nodes so the constraint network has a so called *small world topology*. Nowadays the binary inequality constraints in each row and column can be encapsulated into an all-different constraint which achieves stronger pruning and makes the problem easier to solve (but still cannot solve the problem of any order [5,8]).

3 Quasigroup Completion Problem

As we showed in the previous section, a Latin square can be modeled as a CSP so it can serve as a benchmark domain for constraint satisfaction algorithms. We also sketched a simple algorithm to find a complete Latin square so such a benchmark is not very challenging. Assume now, that some cells in the Latin square are pre-filled, we have a partial Latin square, and the task is to determine whether the empty cells can be filled in such a way that we obtain a complete Latin square. Gomez and Selman [6] proposed this new benchmark based on completing partial Latin squares and they called it a *Quasigroup Completion Problem* (QCP). The problem is parameterized by the order of a Latin square and by the number of filled cells. Formally, the Quasigroup Completion Problem is described by a pair $\langle N,p \rangle$, where N is an order of the Latin square to be completed and p is a *filling ratio*, that is the ratio between the number of pre-filled cells and the total number of cells (N^2).

Pre-assigning some values to variables modeling the Latin square introduces perturbations into the structure of constraint network which makes the structure similar to that found in real-world domains like scheduling and experimental design [8]. A particular real-life problem that maps directly to the above mentioned problem of completing a partially filled Latin square is the problem of assigning wavelengths to routes in fiber-optic networks [10]. Note also that the Quasigroup Completion Problem is known to be an NP-complete problem [4]. Not surprisingly, the straightforward constraint model with the all-different constraints cannot be used alone to solve instances of higher order (34 and more) [8] and more sophisticated techniques like hybrid algorithms [8] or dual models with special value selection heuristics [5] are necessary. This makes the problem non-trivial and hence interesting as a benchmark for comparing constraint satisfaction techniques. This benchmark bridges the gap between the purely random problems like a Random CSP and the highly structured problems.

The question now is how to generate random instances of QCP, in particular how to select the cells to be pre-filled for a given QCP $\langle N,p \rangle$. One possible model could be selecting the cell to be filled with the probability p. Let us call it a *model A* similarly to the classification used for Random CSPs [11]. Another possibility is to select exactly $\lfloor pN^2 \rfloor$ cells to be filled, where $\lfloor X \rfloor$ means the closest (to X) integer between X and 0. Let us call it a *model B*. In this paper we will study the model B, where the cells to be filled are selected randomly and uniformly. We use a random generator that selects uniformly and randomly $\lfloor pN^2 \rfloor$ different elements from the set $\{0,\ldots,N^2-1\}$. Each such element z represents a position in the Latin square of order N that can be described by the coordinates $\langle 1+\lfloor z/N \rfloor, 1+(z \mathit{\ mod\ } N) \rangle$ (Figure 2).

0	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**
12	**13**	**14**	**15**

Fig. 2. A linear encoding of the positions of cells in a Latin square of order 4

The second open question is how to select a value to be filled in a given cell. The basic requirement is that the values in cells in each row and in each column must be different. So, when selecting a value for the cell in the position $\langle x,y \rangle$, this value must be different from the values already filled in the cells of the row x and in the cells of the column y. We propose the following simple technique based on constraint propagation through binary inequalities. Latin square is modeled as a CSP as described in the previous section using binary inequalities between the variables of the same row and of the same column. For a cell to be filled (the cell selection process is described in the previous paragraph), we select randomly a value from the current domain of respective variable. Then the problem is made arc consistent which means that the value is removed from the variables of the same row and of the same column. Consequently, when selecting a value for the next cell, the domain contains only the values that are different from already filled values in the same row and in the same column. This technique mimics the behavior of the original generator from [6]. It ensures that only valid Latin squares are generated, that is no symbol occurs twice in a row or in a column. However, because of incompleteness of constraint propagation we cannot guarantee that a “completable” Latin square is found. Figure 3 shows a situation where a bad initial value selection causes impossibility to fill another cell. Note that any generator attempting to generate values one by one suffers from this problem.

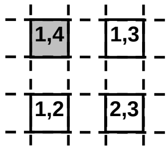


Fig. 3. The problem of simple QCP generators. If value 1 is selected for the top left cell then no complete Latin square exists.

When looking at Figure 3 we can see that if value 4 is selected for the top left cell then the above problem does not occur. Therefore, it might be useful to enhance the generator by allowing a *shallow backtracking* that can try another (randomly selected) value after an immediate failure. This process is repeated until a value is found or all values were tried. It is still possible that no value for the variable is found so this technique does not guarantee finding a valid Latin square but the hope is that it increases chances to find one. Unfortunately, our preliminary experiments showed that this technique does not increase the number of generated valid instances (on average). Note that the generator should produce the random problems fast so its complexity should not be exponential. Therefore, we cannot use full backtracking (probably incomplete search might be used but we did not try it yet).

Another option how to improve chances of finding a value for the variable is strengthening constraint propagation to remove more inconsistent values from the domains. As we already mentioned, there is a natural way how to strengthen propagation in the constraint model for Latin squares – using the all-different constraint by Régin. This approach has already been proposed in [14] – we will present a detail experimental comparison of both generators later in the paper. It will show that the all-different generator produces a higher number of satisfiable instances.

4 Quasigroups With Holes

As we already mentioned, the main problem of QCP is that the generators cannot guarantee production of satisfiable benchmark instances which could cause problems when evaluating incomplete solving techniques. In the previous section we described a method that increases the number of satisfiable instances via strengthening constraint propagation, but this method still does not guarantee satisfiability (see the next section for experimental justification of these claims). It would be possible to accompany the proposed generator by an algorithm that filters the unsatisfiable instances. Still, the problem is that for some parameters the generator does not produce a valid instance and hence no satisfiable instance is available for evaluation. This happens typically in the area where the hardest problems settle (see the section on experiments) so it would be beneficial if the generator produces satisfiable instances directly. Surprisingly, it is often difficult to develop a direct generator of satisfiable instances. The problem with such generators is that they should not be biased in the sense that the generator should produce any satisfiable instance with a uniform distribution. Therefore, the simple generator of complete Latin squares described in the second section is not appropriate because it produces Latin squares with a specific structure only (and hence, completing such Latin squares is not a difficult task).

The paper [1] proposes a direct generator for satisfiable quasigroup problems. The idea is to generate a complete Latin square to which a fraction of holes is punched. The resulting incomplete Latin square is then guaranteed to be satisfiable. This problem is called *Quasigroups With Holes* (QWH). However, the problem of generating uniformly distributed Latin squares is non-trivial. Actually, the generator is not described in [1] and the reader is referred to the paper [9] which describes the method and gives a theoretical justification. In the next paragraphs, we will survey the method from [9], we will present the QWH generator based directly on this method, and then we will reformulate the generator to work directly with the Latin squares.

4.1 Original Generator

Jacobson and Matthews [9] proposed a method for generating uniformly distributed random Latin squares by randomly traversing a graph, where nodes correspond to Latin squares and edges describe transformations between the Latin squares. They proved that the diameter of the graph is $4(N-1)^2$, where N is the order of the Latin square. It means that the minimal distance between two Latin squares is no greater than $4(N-1)^2$ so it is possible to obtain any Latin square from a given Latin square in $4(N-1)^2$ moves. The QWH generator can be conceived as follows. We start with a random Latin square generated for example by the method described in Figure 1C. After performing $4(N-1)^2$ random moves we should obtain any Latin square with uniform probability. The open question is how to perform a move, that is, how to transform one Latin square into another Latin square. We will answer this question in the following paragraphs.

To simplify description of moves, Jacobson and Matthews proposed to extend the graph by nodes describing so called improper Latin squares where the condition of a Latin square is violated a “little” (see below). Then the diameter of the new graph and hence the minimal distance between any two (proper or improper) nodes is bounded by $2(N-1)^3$ (for a formal proof see [9]). They represent the Latin square of order N by a contingency table *f* of size N×N×N that contains {0,1} values only. The condition on a Latin square (in each row and in each column, each element appears exactly once) is then equivalent to the formulas:

$$\forall x, y \in \{1,\ldots,N\} \sum_{z\in\{1,\ldots,N\}} f(x,y,z) = 1 \qquad (a)$$
$$\forall x, z \in \{1,\ldots,N\} \sum_{y\in\{1,\ldots,N\}} f(x,y,z) = 1 \qquad (b)$$
$$\forall y, z \in \{1,\ldots,N\} \sum_{x\in\{1,\ldots,N\}} f(x,y,z) = 1 \qquad (c)$$

Basically, *x* and *y* describe the coordinates of the cell and *z* describes the element in the cell (*x*,*y*) if *f*(*x*,*y*,*z*)=1. So formula (*a*) says that exactly one element is filled to the cell (*x*,*y*), formula (*b*) says that the element *z* appears exactly once in the row *x*, and formula (*c*) says that the element *z* appears exactly once in the column *y*. We call a Latin square with the above (proper) contingency table a *proper Latin square*. An *improper Latin square* is defined by the (improper) contingency table satisfying the conditions (*a*)-(*c*) but allowing exactly one element of the contingency table to contain value -1.

Now, it is easier to formulate the moves as operations over (proper and improper) contingency tables. Assume that we start with a proper contingency table. We select randomly a cell of *f* such that *f*(*x*,*y*,*z*) = 0 and we will try to increase this value by one which is equivalent to assigning the value *z* to the cell (*x*,*y*). Each line in *f* containing the cell (*x*,*y*,*z*) must hold a cell filled by one according to (*a*)-(*c*). Let *x'*, *y'*, and *z'* be the indexes of these lines. These coordinates define a sub-cube in the contingency table with nodes at (*x*,*y*,*z*), (*x*,*y*,*z'*), (*x*,*y'*,*z*), (*x'*,*y*,*z*), (*x'*,*y*,*z'*), (*x'*,*y'*,*z*), (*x*,*y'*,*z'*), and (*x'*,*y'*,*z'*) (see Figure 4). If we increase the value in *f*(*x*,*y*,*z*) by one then we need to decrease the values in *f*(*x*,*y*,*z'*), *f*(*x*,*y'*,*z*), *f*(*x'*,*y*,*z*) by one to keep the conditions (*a*)-(*c*) valid. Next, the values in *f*(*x'*,*y'*,*z*), *f*(*x*,*y'*,*z'*), *f*(*x'*,*y*,*z'*) must be increased by one and finally the value in *f*(*x'*,*y'*,*z'*) must be decreased by one. If all these operations are performed then visibly the conditions (*a*)-(*c*) hold again. However, it may happen that the value in *f*(*x'*,*y'*,*z'*) will become -1, in the case that *f*(*x'*,*y'*,*z'*)=0, but this will be the only cell with a negative value (see Figure 4).

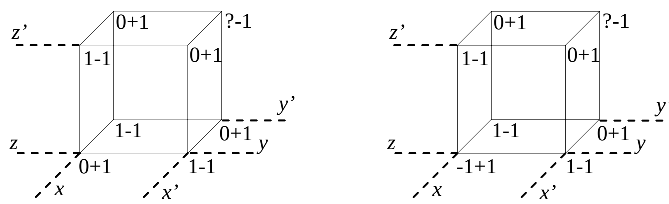


Fig. 4. A plus/minus one move in the proper (left) and improper (right) contingency table.

Notice that if we start with a cell such that *f*(*x*,*y*,*z*) = -1 (the contingency table is improper) then we can perform the same set of operations as above and again we will obtain either a proper or improper contingency table (Figure 4 right). Hence the above described mechanism specifies moves between proper and improper contingency tables. Jacobson and Matthews showed that on average after N such moves we will obtain a proper contingency table describing a Latin square of order N. Figure 5 shows the algorithm for a single move. By using information about the diameter of graph with nodes marked by Latin squares (see above) we propose to do at least $2(N-1)^3$ such moves and then stop when a proper contingency table is obtained.

```
move
   find x,y,z s.t.
     if f is improper then f(x,y,z)=-1
     if f is proper then f(x,y,z)=0
   find x′,y′,z′ s.t. f(x′,y,z)=f(x,y′,z)=f(x,y,z′)=1
   // if f is proper then these points are unique
   // if f is improper then there are two choices
   //   for each point, select one point randomly
   increase f(x,y,z),f(x,y′,z′),f(x′,y,z′),f(x′,y′,z)
   decrease f(x,y,z′),f(x,y′,z),f(x′,y,z),f(x′,y′,z′)
end move
```

Fig. 5. The algorithm for move between contingency tables.

2.2 Reformulated Generator

In the previous section we presented the algorithm for moves between proper and improper contingency tables. Notice that if the contingency table is improper, which happens when $f(x',y',z')$ becomes -1, then the next move starts with $f(x',y',z')$ that will be increased by one. The improper contingency table describes a situation when two values, z and the original value in (x',y'), are assigned to the cell (x',y') at the same time. According to the above observation, we know that in the next move the original value in (x',y') or z will be unassigned and so we propose to postpone assignment of z to the cell (x',y') to the next move. Before assigning the value we will check whether the value in (x',y') is z'. If this is true then we got a proper Latin square and we can stop the sequence of improper moves (so z is not assigned to the cell which is equivalent to assigning it and unassigning it in the next step). Otherwise we assign the value z to the cell (x',y'), we took the original value in this cell and "propagate" it further. Figure 6 describes how the values are moved between the cells.

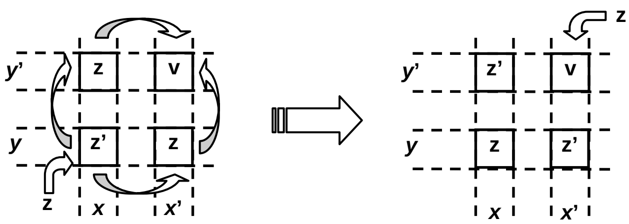


Fig. 6. Shifting values in a Latin square when the value z should be placed to position (x,y).

The above idea can now be encoded using the data structures describing directly a Latin square instead of its contingency table. Figure 7 shows the algorithm for moving between the proper Latin squares directly. The move is started with a random position (x,y) and a random value z to be placed there: `proper_move(x,y,z,z)`.

When the procedure stops, a proper Latin square is obtained and another random move can be started. Notice that if the Latin square is improper then there are two positions in the row *x* and two positions in the column *y* where the value *v* is located. In such a case, one position in the column and one position in the row are selected randomly. If the position is selected deterministically, for example the first found position, then the algorithm starts cycling! Like for the original generator we propose to call the procedure `proper_move` at least $2(N-1)^3$ times (including the recursive calls inside `proper_move`) so every Latin square can be obtained with uniform probability.

```
proper_move(x,y,z,v)
  z' ← table(x,y)
  if z'=v then table(x,y) ← z, return
  y' ← a position (column) of cell with v in the row x
  x' ← a position (row) of cell with v in the column y
   // if z=v then x' and y' are unique
   //  otherwise there are two such positions,
   //   one position is selected randomly
  table(x,y) ← z
  table(x,y') ← z'
  table(x',y) ← z'
  proper_move(x',y',v,z')
end proper_move
```

Fig. 7. The algorithm for move between proper Latin squares.

5 Experimental Results

We have implemented the presented generators using the clpfd library [3] of SICStus Prolog version 3.11.2. All presented results were accomplished under Windows XP Professional on 1.8 GHz Pentium 4 with 512 MB RAM. The running time is measured in milliseconds via the statistics predicate with the `walltime` parameter. The results of 100 runs are presented.

2.2 Generator Quality

Generators of random problems are expected to produce problems in the whole spectrum of their parameters. In our first experiment, we measured the number of generated partial Latin squares relative to the number of attempts to generate a problem. Recall, that the generator should produce a partial Latin square, namely no symbol occurs twice in a row or in a column, with a given filling. Figure 8 shows the result for Latin squares of order 30 and different filling ratios. Notice that the original QCP generator falls short on the task of generating problems where more pre-filled cells are requested. Actually, the generator is not able to produce any instance when the filling ratio is greater than 72%. This is not so surprising because the more cells should be pre-filled the higher probability is that no value can be found for some cell (see Figure 3). A similar behavior can be observed for the QCP-alldiff generator but thanks to stronger propagation via all-different constraints the chances to select a consistent value increases and hence the generator is still able to produce some instances. It is a pity that the papers [6,14] proposing these generators did not mention this feature, probably because the authors used Latin squares of small orders (below 20) where this behavior cannot be observed. For the sake of completeness, let us highlight that the QWH generators always produce a problem instance.

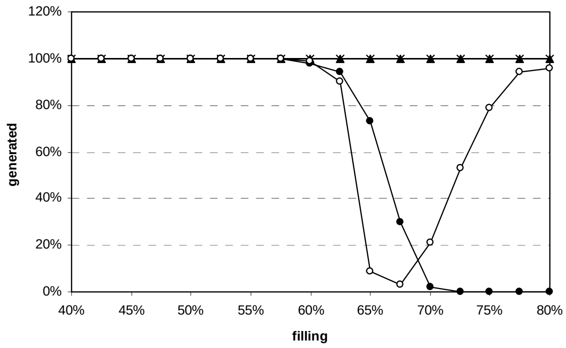


Fig. 8. The relative number of generated problems for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

The second feature that we focused on is the "hardness" of the generated problems. We used the presented generators to produce partial Latin squares of order 30 which is the border where the quasigroup problems are non-trivial but still solvable by standard constraint satisfaction techniques [8]. To solve the problem we used a standard MAC algorithm with the constraint model using all-different constraints, "smallest domain first" variable selection, and "minimal value first" value selection. We used a time limit of 2 minutes to solve each problem (there are some extra hard instances that would prevent finishing experiments in a reasonable time if timeout is not used).

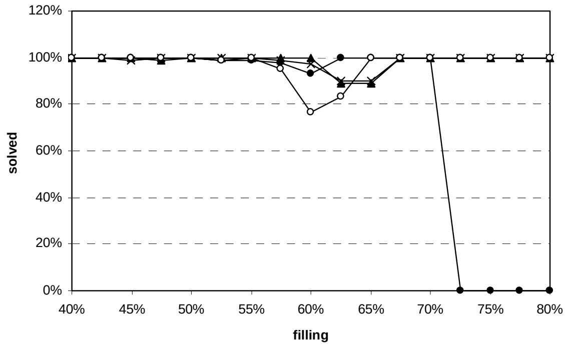


Fig. 9. The relative number of solved problems for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

Figure 9 shows a relative number of solved instances as a function of filling ratio. As we can see most of the generated problems are solvable within the 2 minutes timeout but there are some instances around 62% that were not solved. This is a first indication where the hard problems might settle but it does not show yet how hard the problems are. No problem is generated by QCP-orig for filling rations above 72% and hence no problem is solved there.

We compare hardness of the generated problems by measuring runtime of the above described straightforward solver when solving the problems generated by the studied generators. Figure 10 shows median runtime to solve the generated problems. By solving the problem we understand finding a completion of the partial Latin square or proving that no completion exists. This experiment brought some surprising results. First, the phase transition area is shifted for the QWH generators towards the area with a higher filling ratio (in comparison with the QCP generators). Second, the QCP-alldiff generator produces the hardest to solve instances. This could be caused by using the all-different constraints both inside the QCP-alldiff generator and inside the solver, but we have no evidence of this (probably trying another solving approach might show whether the generated instances are hard in general). Finally, notice that the QCP-orig generator produced quite easy problems. We have observed the above mentioned features for Latin squares of other orders too.

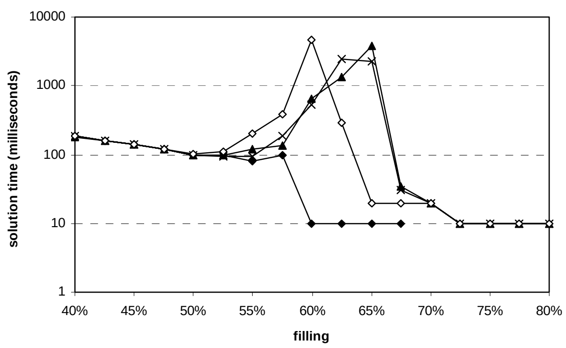


Fig. 10. Median solution time in milliseconds (logarithmic scale) for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ×: QWH-new).

As we already mentioned, it should be clear whether the generator produces satisfiable instances or not. Production of satisfiable instances is especially important when the problems are used to compare incomplete algorithms like local search techniques or incomplete depth-first search techniques [2]. In the next experiment we measured the number of satisfiable instances among the solved problems. Figure 11 shows the relative number of satisfiable instances for Latin squares of order 30 and different filling ratios. QWH generators are guaranteed to produce satisfiable instances; the experiment just confirmed this feature. Hence these generators are appropriate for providing instances to compare incomplete algorithms. The behavior of QCP-orig generator with satisfiable instances on one side and unsatisfiable instances on the other side has already been presented at [6]. However, taking in account Figure 9, we can deduce that no satisfiable instance is generated for larger filling ratios simply because no instance is generated there. Hence the conclusions in [6] are a bit misleading because the readers might expect that QCP-orig produces unsatisfiable instances for larger filling ratios which is not true in general (especially for higher order of Latin squares). The number of satisfiable instances produced by QCP-alldiff is also decreasing around the phase transition area but it increases again for large filling ratios. Our other experiments (not presented here) showed that the area with a smaller number of satisfiable instances enlarges with increasing order of the Latin square. Nevertheless, QCP-alldiff might still be appropriate for generating problems used to compare complete algorithms.

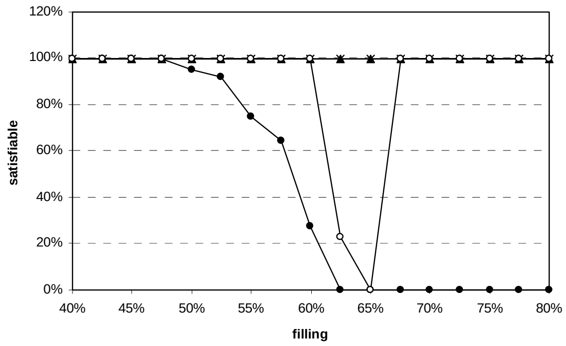


Fig. 11. The relative number of satisfiable instance for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

In our last experiment, we tried to estimate how complicated the generated problems are if the constraint model with all-different constraints is used. In particular, we measured the number of cells that have a value after the all-different constraints are posted and propagated but before search is started. Mean values among the consistent problems are presented in Figure 12. The dashed line indicates the initial filling produced by generators so the curves above this line indicate that additional values are deduced by the initial constraint propagation using the all-different constraints. Notice that for the filling ratio smaller than 60%, no values for additional cells were deduced while for the filling ratio greater than 70%, the values of all the variables were set using constraint propagation (so no search is necessary to solve the problem). We can see that the initial constraint propagation deduced more values for the problems produced by the QCP-all generator in comparison to the QWH generators. This is probably caused by using the all-different constraints during generation. Hence the QCP-alldiff produces instances with a larger number of pre-filled cells than requested. As proposed in [14] the intended filling can be achieved by measuring the total number of instantiated variables (including those instantiated through propagation) and stopping the generation process when this number is equal to the required filling. However, this technique makes the generator "less random" because some cells are filled by propagation rather than randomly. Moreover, this technique is not applicable to QWH generators that do not use propagation. If we use an assumption that stronger initial pruning means that the problems are easier for solving using the CP technology then the QWH generator produces harder problems in the phase transition area. This fits our observation from Figure 10, but recall that the phase transition area is shifted to smaller filling ratios for QCP-alldiff.

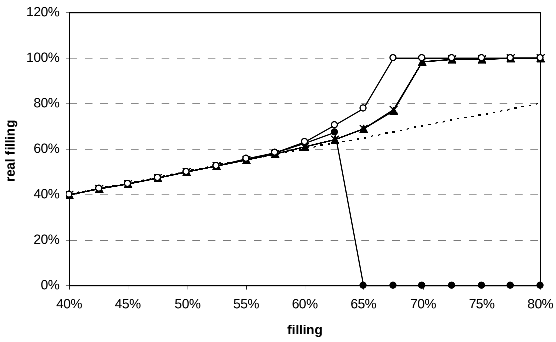


Fig. 12. The number of pre-filled cells (in percent) using the all-different constraints for the quasigroup problems of order 30
(●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

2.2 Generator Efficiency

Sometimes the generators of benchmark problems run off-line so they produce problems that are put into benchmark libraries. Nevertheless, in case of random problems, the generators are frequently used on-line to generate problems that are used immediately to test the solvers. In this second case, it is desirable for the generator to be fast (the users do not want to waste time by generating the problems).

We measured the runtime of studied generators to show how appropriate the generators are for on-line experiments. Figure 13 shows the runtime as a function of the filling ratio and Figure 14 shows the runtime as a function of the order of a Latin square. Visibly the better quality of the QCP-alldiff generator is paid-off by longer runtime. Moreover, the runtime of the QCP-alldiff generator increases faster than the runtime for the new QWH generator and from the order 50, it is actually slower. Consequently, the QCP-alldiff generator pays-off only for smaller order of the Latin square which also takes in account our discussion from the previous section. Recall that QCP-alldiff seems to produce the hardest to solve instances (Figure 10) so we believe that this generator is appropriate for comparing complete solving techniques. Despite the fact that the original QCP generator is very fast, we do not recommend its usage simply because it produces less problem instances and the generated problem instances are order of magnitude easier to solve (Figure 10) in comparison to other presented generators. The runtime of QWH generators is slower than the original QCP generator, but recall that all instances produced by the QWH generators are satisfiable, which makes the QWH generators the only choice for testing incomplete solving techniques. Notice also that the reformulated QWH generator is about two times faster than the original QWH generator.

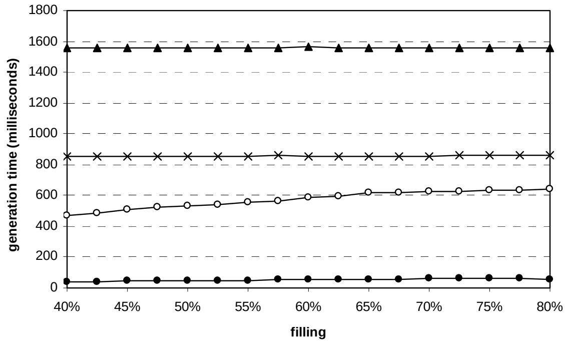


Fig. 13. The time (in milliseconds) to generate a quasigroup problem of order 30 and variable filling (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ×: QWH-new).

Figure 14 compares the runtimes of the generators on problems with a fixed filling ratio 0.6 and with changing order of a Latin square. We have selected the filing ratio 0.6 because it is within the phase transition region, however, we performed experiments with other filling ratios and the results were similar.

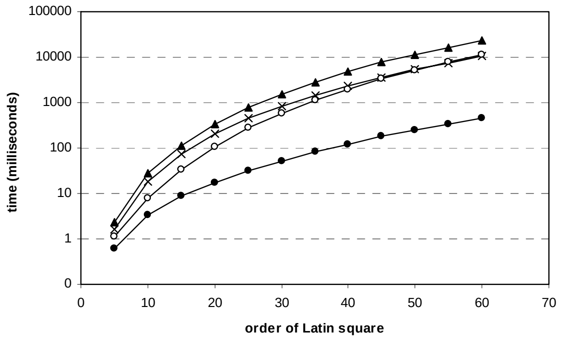


Fig. 14. The time (in milliseconds, a logarithmic scale) to generate a quasigroup problem with the filling ratio 0.6 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ×: QWH-new).

5 Conclusions

Completion of a partial Latin square is an interesting problem whose structure is close to real-life problems [7,10]. It is also a non-trivial problem [4] whose solving requires sophisticated techniques [5,8]. Finally, it is a problem whose instances can be generated randomly as a Quasigroup Completion Problem (QCP) [6] or Quasigroups With Holes (QWH) [1]. These features make the completion of partial Latin squares an ideal candidate for benchmarking constraint satisfaction techniques. In this paper, we studied the generators for both QCP and QWH and we provided detailed guidelines how to construct such generators. This alone is an important contribution because writing the generator for QWH is a non-trivial problem. Moreover, as far as we know this is the first paper in the CSP literature giving the exact description of the generator for QWH. We experimentally compared the existing generators and we proposed a reformulated version of the QWH generator that is much faster than the original generator. Even if the QWH generators are slower than the original QCP generator, their quality measured as a number of produced satisfiable instances is much higher. Hence, the QWH generators are appropriate to prepare problem instances for testing incomplete algorithms like in [2] while the QCP generators may still be useful for testing complete algorithms like in [12].

References

1. Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating Satisfiable Problem Instances. In Proceedings of the Seventeenth National Conference on Artificial Intelligence. AAAI Press (2000) 256–261
2. Barták, R., Rudová, H.: Limited Assignments: A New Cutoff Strategy for Incomplete Depth-First Search. In Proceedings of the 2005 ACM Symposium on Applied Computing. ACM (2005) 388–392
3. Carlsson, M., Ottosson, G., Carlson, B.: An Open-ended Finite Domain Constraint Solver. In Programming Languages: Implementations, Logics, and Programming. LNCS 1292. Springer-Verlag (1997)
4. Colbourn, C.: The Complexity of Completing Partial Latin Squares. Discrete Applied Mathematics 8 (1984) 25–30
5. Dotú, I., del Val, A., Cebrián, M.: Channeling Constraints and Value Ordering in the Quasigroup Completion Problem. In Proceedings of Eighteenth International Joint Conference on Artificial Inteligence. Morgan Kaufmann Publishers (2003) 1372–1373
6. Gomez, C., Selman, B.: Problem Structure in the Presence of Perturbations. In Proceedings of Fourteenth National Conference on Artificial Intelligence. AAAI Press (1997) 221–226
7. Gomez, C., Shmoys, D.: Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In Proceedings Computational Symposium on Graph Coloring and Generalizations (2002)
8. Gomez, C., Shmoys, D.: The Promise of LP to Boost CSP Techniques for Combinatorial Problems. In Proceedings CPAIOR'02 (2002) 291–305
9. Jacobson, M.T., Matthews, P.: Generating Uniformly Distributed Random Latin Squares. Journal of Combinatorial Designs 4 (1996) 405–437
10. Kumar, S.K., Russell, A., Sundaram, R.: Approximating Latin Square Extensions. Algorithmica 24 (1999) 128–138
11. MacIntyre, E., Prosser, P., Smith, B., Walsh, T.: Random Constraint Satisfaction: theory meets practice. In Principles and Practice of Constraint Programming - CP98. LNCS 1520. Springer-Verlag (1998) 325–339
12. Meseguer, P., Walsh, T.: Interleaved and Discrepancy Based Search. In Proceedings of 13th European Conference on Artificial Intelligence. Wiley (1998) 239–243
13. Régin, J.-Ch.: A filtering algorithm for constraints of difference in CSPs. In Proceedings of Twelfth National Conference on Artificial Intelligence. AAAI Press (1994) 362–367
14. Shaw, P., Stergiou, K., Walsh, T.: Arc Consistency and Quasigroup Completion. In Proceedings of the ECAI-98 workshop on non-binary constraints (1998)