HUNTER-GATHERER: Applying Constraint Satisfaction, Branch-and-Bound and Solution Synthesis to Computational Semantics

Stephen Beale

MCCS-96-289

Computing Research Laboratory
Box 30001
New Mexico State University
Las Cruces, New Mexico 88003

Abstract

This work[1] integrates three related AI search techniques and applies the result to processing computational semantics, both in the analysis of source text to discover underlying semantics, as well as in the planning of target text using input semantics. We summarize the approach as "Hunter-Gatherer:"

- Branch-and-Bound and Constraint Satisfaction allow us to "hunt down" non-optimal and impossible solutions and prune them from the search space.
- Solution Synthesis methods then "gather together" all optimal solutions while avoiding exponential complexity.

Each of these three general AI techniques will be described. We will look at how how they have been used to solve a variety of problems. These general techniques were extended or used in novel ways in this project. We will describe these extensions in detail and give examples of how they were applied to computational semantic processing. A major contribution of this work will also be in showing how and why Natural Language is a prime candidate for applying these methods, and how they can enable near-linear time processing. As part of this discussion, we will demonstrate the important result that by converting Text Planning to a constraint satisfaction problem, Means-End type planning can be replaced by an efficient constraint-based search through a complex tree. Finally, we will examine the results in the light of the Mikrokosmos Machine Translation project. This project is a large-scale Spanish-English MT system implemented at New Mexico State University. We will be able to evaluate the control mechanism presented here against a large corpus of sample texts. In particular, we will show that a search space in the billions (or in some cases ga-zillions) can be reduced to a few thousand or less, with a corresponding decrease in run-time.

[1]Research reported in this paper was supported in part by Contract MDA904-92-C-5189 from the U.S. Department of Defense.

Contents

1. Introduction **5**

1.1. Hunters and Gatherers in AI: An introduction to constraint-based AI techniques . . 5

1.2. Hunters and Gatherers in Computational Semantics 7

1.2.1. Applying the AI techniques to Computational Semantics 7

1.2.2. Natural-Language - a "Natural" Constraint Satisfaction Problem 9

2. Hunters and Gatherers in AI **11**

2.1. Constraint Satisfaction Problems 12

2.2. Solution Synthesis 19

2.3. Branch-and-Bound 34

2.4. Other Strategies for CSPs 36

3. Hunters and Gatherers in Computational Semantics **38**

3.1. Constraint Satisfaction Hunters in Computational Semantics 39

3.1.1. Identifying "Circles" of Dependency 41

3.1.2. Setting Up Constraint Dependencies 44

3.2. Solution Synthesis Gatherers in Computational Semantics 47

3.3. Using Branch-and-Bound in an Uncertain World 52

3.3.1. Implementing Branch-and-Bound 52

3.3.2. Branch-and-Bound Results 59

3.4. Constraint Satisfaction in Natural Language Generation: The PICARD Text Planning System 62

3.4.1. Text Planning for Machine Translation 63

3.4.2. Using Constraint Satisfaction to Enable Abstractions 66

3.5. Natural Language - a "Natural" CSP 70

3.5.1. Local Dependency in Computational Semantics 71

3.5.2. Classes of Problems for which HUNTER-GATHERER is Beneficial 74

4. Discussion **76**

4.1. Planning and Island Processing 76

4.2. Formal Properties of HUNTER-GATHER: Soundness and Completeness 77

4.2.1. Soundness 77

4.2.2. Completeness 78

5. Conclusion **79**

Acknowledgements **79**

1. Introduction

Fifty six million, six hundred eighty seven thousand, and forty. A big number, to be sure. This is the number of possible semantic analyses for an **average** sized sentence in the Mikrokosmos Machine Translation project. Complex sentences have gone past the trillions. If every combination could be accurately judged in one thousandth of a second, it would still take almost a day to analyze the average sentence. And you can forget about the hard ones.

And yet, understanding natural language sentences is intuitively not an exponential affair. Not every word in a sentence is dependent on every other word. Sentences can generally be broken up into relatively independent areas of self-contained meaning which then interact on a higher level to produce the meaning of the whole. This research aims to recognize that fact, analyze it, and apply appropriate AI techniques to take advantage of it.

This work integrates three related AI search techniques and applies the result to processing computational semantics, both in the analysis of source text to discover underlying semantics, as well as in the planning of target text using input semantics. We summarize the approach as "Hunter-Gatherer:"

- Branch-and-Bound and Constraint Satisfaction allow us to "hunt down" non-optimal and impossible solutions and prune them from the search space.
- Solution Synthesis methods then "gather together" all optimal solutions while avoiding exponential complexity.

We will describe each of these general AI techniques and look at how how they have been used to solve a variety of problems. These general techniques were then extended or used in novel ways in this project. We will describe these extensions in detail and give examples of how they were applied to computational semantic processing. A major contribution of this work will also be in showing how and why Natural Language is a prime candidate for applying these methods, and how they can enable near-linear time processing. As part of this discussion, we will demonstrate the important result that by converting Text Planning to a constraint satisfaction problem, Means-End type planning can be replaced by an efficient constraint-based search through a complex tree. Finally, we will examine the results in the light of the Mikrokosmos Machine Translation project. This project is a large-scale Spanish-English MT system implemented at New Mexico State University. We will be able to evaluate the control mechanism presented here against a large corpus of sample texts. In particular, we will show that a search space in the billions (or in some cases ga-zillions) can be reduced to hundreds, with a corresponding decrease in run-time.

This introduction will give brief descriptions of each of the main points to be covered in detail below. The interested reader will then be able to judge which sections of the report are of immediate interest. The text is divided into four main sections as follows:

1.1. Hunters and Gatherers in AI: An introduction to constraint-based AI techniques

In recent years, Constraint Satisfaction Problems (CSP) have received a good deal of attention in the computer science world (see [Tsang 93] for a detailed look at CSP). Constraint satisfaction techniques enable search procedures to prune off substantial portions of the search tree by

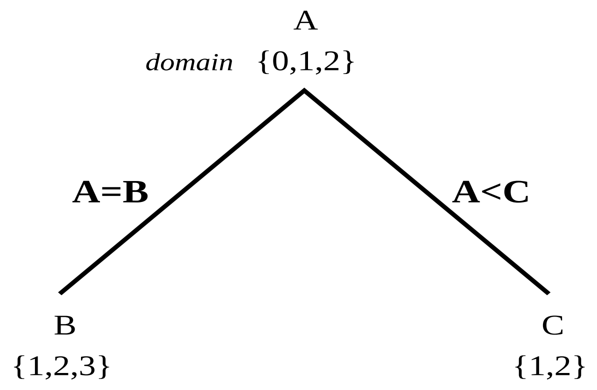

Figure 1: Basic Constraint Satisfaction Problem

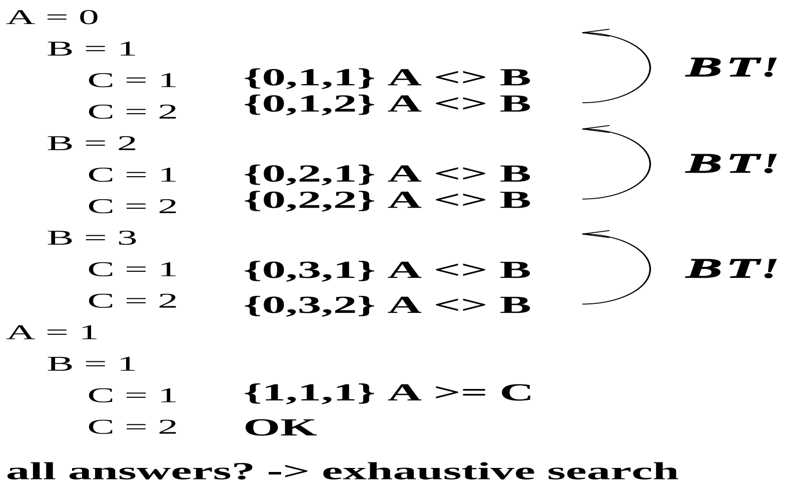

Figure 2: Backtracking

identifying solution sets/subsets that cannot meet input constraints. For example, in Figure 1, three variables, A, B and C, with the domains given, are to have values chosen subject to the two constraints: [A = B] and [A < C].

A naive parser would try every combination of A, B and C until it found one that met the constraints (Figure 2). If all correct answers are desired, an exhaustive search is required. Notice that certain combinations that always lead to failure (A=0, B=x) are tried again and again. Constraint satisfaction programming techniques can eliminate this unnecessary processing. For instance, it can reduce A's domain to {1,2}, since 0 as a value for A can never satisfy the [A = B] constraint. Actually, A's domain can be reduced to {1}, since 2 as a value for A can never satisfy the [A < C] constraint. Once A is reduced to {1}, then B can also be reduced to {1}, and C can be reduced to {2}. Only one choice for each variable is left; thus, the answer is achieved without search. In section 2.1, we will overview the algorithms used to achieve constraint "consistency" in a search and examine some of the applications which have benefited from this kind of treatment.

Solution synthesis (section 2.2) is a method of generating **all** valid answers to a search problem.

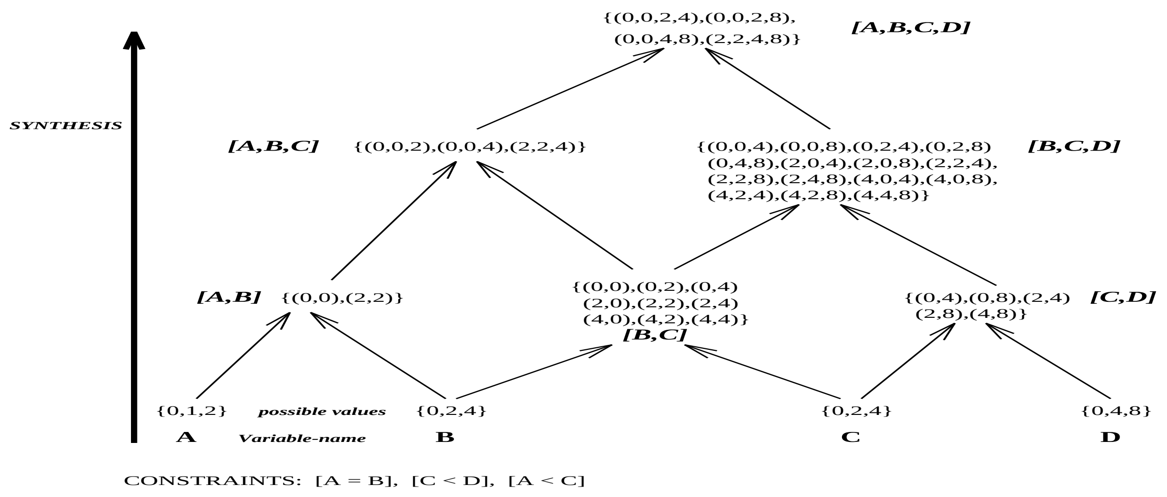

Figure 3: Solution Synthesis

Instead of working from the top of the tree down,[2] solution synthesis attempts to combine legal combinations of nodes from the bottom up. Figure 3 gives a graphical overview of the process. Used in conjunction with constraint satisfaction techniques, solution synthesis is a powerful method for avoiding exponential time requirements associated with conventional tree search.

Branch-and-bound techniques (section 2.3) can be used to find the optimal solution without resorting to heuristics.[3] The basic idea is displayed in Figure 4. In that Figure, a graph is displayed with the cost of each arc listed. The order of arc traversal used in a branch-and-bound algorithm is marked by circled numbers. The shortest total path accumulated at any point is always expanded next. If there is a tie, as is the case at the start node when no paths have been chosen yet, then the shortest next arc is taken. Arcs without circled numbers in Figure 4 were not tested, because they were extensions of paths of length greater than the length of a valid solution. Branch-and-bound identifies and eliminates paths which can be guaranteed to have more costly solutions than some valid solution.

Each of these AI methods, constraint satisfaction, solution synthesis and branch-and-bound will be described, the appropriate literature reviewed, and the situations under which they can be applied advantageously will be noted.

1.2. Hunters and Gatherers in Computational Semantics

1.2.1. Applying the AI techniques to Computational Semantics

In this section, we will discuss how these techniques were modified and adapted for use in computational semantics.

[2]CSP does not assume tree shaped search spaces. Neither do we, for that matter, although computational semantics generally present as tangled trees, a fact we take advantage of (see below).

[3]They can be used with heuristics as well, if desired.

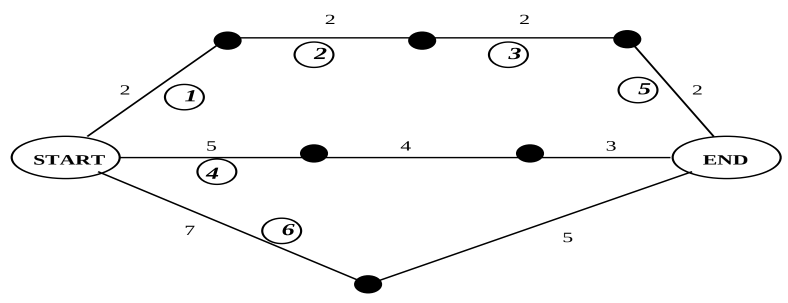

Figure 4: Branch-and-Bound

We will demonstrate how constraint satisfaction information allowed us to identify "circuits" of inter-dependence in the input. Solutions for each circuit are synthesized apart from the rest of the problem. In addition, we used the natural tree-like form of our inputs to guide further synthesizing efforts. Figure 5 graphically displays how circuits of inter-dependence and tree-circuits can be combined into larger and larger circuits. At each level, solutions can be synthesized utilizing results from circuits below.

Branch-and-bound techniques were refined for use with the solution synthesis and constraint satisfaction methods. As each of the "circuits" mentioned above are synthesized, certain sub-circuits will no longer be dependent on nodes outside of the circuit. These sub-circuits can be optimized, with non-optimal solutions "bound" and eliminated. This particular merging of techniques accounts for the majority of savings produced by our system.

Finally, constraint satisfaction principles were capitalized on to change means-end text planning into a simpler and faster type of Constraint Satisfaction Problem (CSP), which can be solved using optimized CSP algorithms. A typical means-end planning session goes something like the following:

- main goal: communicate that Frank was lazily reading a book on the couch.
    - Plan1: assume "read" communicates basic Read-Event.
        - preconditions: none
        - effects: instantiate Read-Event
        - goal-minus-effects: still need agent, theme, location and attitude.
            - plan agent
            - plan theme
            - plan location
            - plan attitude
    - Plan2: assume "peruse" communicates Read-Event with "lazy" attitude
        - preconditions: none
        - effects: instantiate Read-Event with "lazy" attitude
        - goal-minus-effects: still need agent, theme, location
            - plan agent
            - plan theme
            - plan location
    - Plan3: assume "potato" communicates Read-Event with "lazy" attitude with location on the couch. Requires a co-eating event.

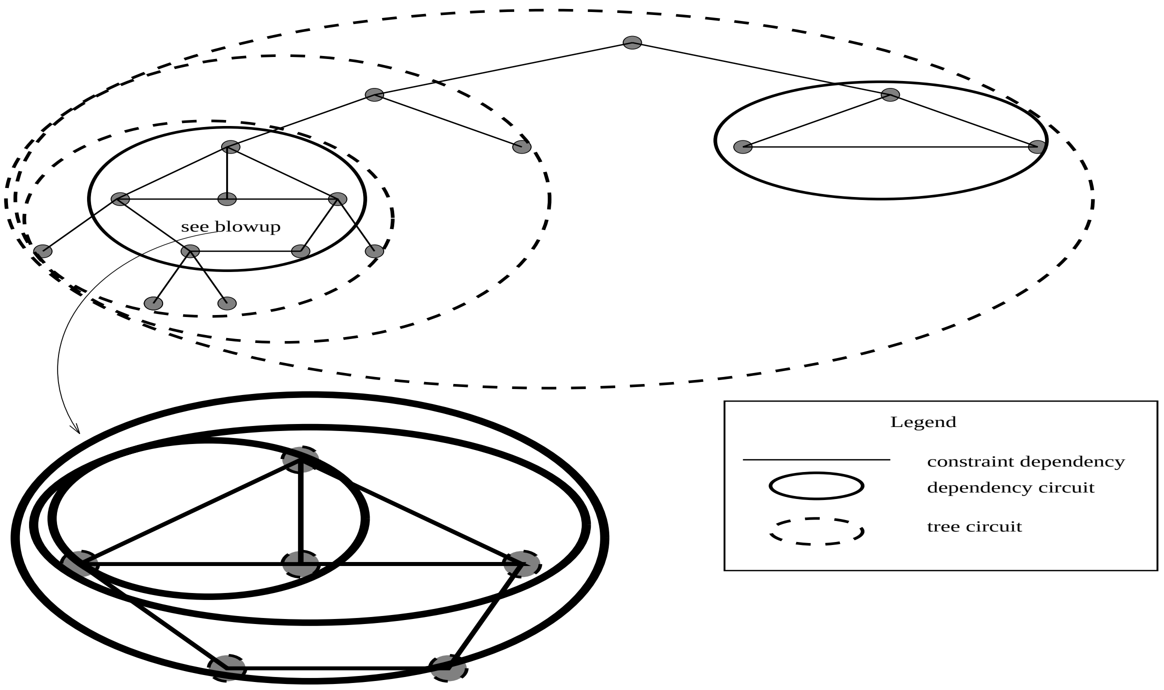

Figure 5: Circuits of Inter-Dependence

- ∗ preconditions: Eating-Event
  - · plan Eating-Event
- ∗ effects: instantiate Read-Event with "lazy" attitude and location on the couch.
- ∗ goal-minus-effects: still need agent, theme
  - · plan agent
  - · plan theme

Each of these plans would be constructed serially, with the most efficient plan being chosen at the end. In this case, Plan3 would be chosen if an Eating-Event could be planned to fulfill the precondition of "potato", otherwise Plan2 would be chosen.

Means-End planners typically are inefficient. Various methods have been used to improve their performance, with "macro-planning" receiving the most attention. Macro-planning will be discussed, and we will demonstrate how, by setting up our planner as a Constraint Satisfaction Problem, we can let the CSP techniques automatically identify macro-plans which can then be efficiently "searched" to find the optimal plan.

1.2.2. Natural-Language - a "Natural" Constraint Satisfaction Problem

Implicit in the work above is the idea that computational semantics naturally fits into the class of problems for which these types of constraint satisfaction techniques apply. We will introduce and discuss the notion of "local inter-dependence." We will show that natural language semantics fits this notion well. We will examine the effect of "long-distance dependencies" on this type of processing. We will look at the class of problems for which the methods described above do **not** work well and show that computational semantics typically does not conform to such problems. Finally, we will argue that the methods presented here are general enough for use in many AI search problems.

2. Hunters and Gatherers in AI

Search is the most common tool for finding solutions in artificial intelligence. That being true, the most common work in the AI smithery is aimed at refining that tool. Such work can be classified into two main areas:

1. Reducing the search space. Looking for sub-optimal or impossible solutions. Removing them. Killing them. "Hunting"
2. Efficiently extracting answer(s) from the search space. Collecting satisfactory answer(s). "Gathering"

The hunter is savage. She kills without mercy to serve the needs of the clan. The gatherer is gentle. He takes in all that supply the needs of the group.[4]

Much work has been done with regard to the hunters. Finding and using "heuristics" to guide search intelligently has been a major focus. Heuristics are necessary when other techniques cannot reduce the size of the search space to reasonable proportions. Under such circumstances, "guesses" have to be made to guide the search engine to the area of the search space most likely to contain acceptable answers. "Best-first" search (see, among many others, [Charniak, et. al., 1987]) is an example of how to **use** heuristics. Such a methodology is almost always combined with some concept of "satisficing" [Newell & Simon, 1972], a determination of whether a given answer is "good enough," regardless of the fact that other "better" solutions may still be present in the search space.

Discovering appropriate heuristics for any given problem is another matter. Often "experts" in the field must be consulted, their methods for finding solutions analyzed, and means of implementing those methods computationally invented. MYCIN [Buchanan and Shortliffe, 1984] is a typical "expert system" whose search is based upon heuristic medical knowledge.

This research does not address heuristic search. Heuristics, by definition, are guesses, and thus can only lead to the most probable answers. In contrast, this work claims that by using the modified CSP techniques presented below, the most optimal solution can be guaranteed for computational semantic problems, even under reasonable time constraints. This is not to say that heuristics cannot be valuable in computational semantics, and in fact, heuristics could be easily added to the methods described here.

Some further clarifications regarding heuristics must be made. Heuristics can be used more conservatively as a method to order the search, without resorting to "satisficing" determinations which may leave optimal solutions undiscovered. There are several general ordering heuristics that are common in constraint satisfaction algorithms. These heuristics are discussed below in the "Other Strategies for CSPs" section. In addition, knowledge sources which seek to constrain and evaluate solutions can also be seen as heuristics. For example, the fact that the Mikrokosmos lexicon constrains the **AGENT** of a **SPEAK** event to be a **HUMAN** is simply a heuristic. Metonymic speach, such as *The White House said today ...* often overrides the basic constraint. In all that follows, we assume a given knowledge source with all of its inherent heuristics. The control mechanisms developed here then find the best solution, **given** that knowledge.

[4]This "hunter-gatherer" business is undoubtedly politically-incorrect. Be advised that we are in no way denigrating pre-industrialist cultures.

The "hunting" techniques applied in this research are most closely related to the field of CSPs; a detailed summary of this work is given below. "Branch-and-bound" methods have been modified and adapted for work with CSPs, and are thus also described below.

"Gathering" has been studied much less in AI. Most AI problems are content with a single "acceptable" answer. Heuristic search methods generally are sufficient. Certain classes of problems, however, demand **all** correct answers. "Solution synthesis" addresses this need. Solution Synthesis techniques (Freuder, 1978; Tsang & Foster, 1990) iteratively combine (gather) partial answers together to arrive at a complete list of all correct answers. Often, this list is then rated according to some separate criteria in order to pick the most suitable answer. Solution synthesis methods will be described below.

In the section 3, "Hunters and Gatherers in Computational Semantics," the modifications and interactions of these "hunter-gatherers" utilized in this project will be developed. In particular, it will be shown that by combining branch-and-bound techniques with a novel solution synthesis method, the best solution for a computational semantic problem can be found in near-linear time. Also, the conversion of a means-end type text planner to a CSP will be described.

2.1. Constraint Satisfaction Problems

The seminal paper on CSP is [Mackworth, 1977]. In this paper he describes the central concepts of CSP and gives the basic consistency algorithms described below. [Mackworth and Freuder, 1985] and, later, [Mohr and Henderson, 1986] improved on the basic algorithms. [Freuder, 1978] introduces solution synthesis and [Tsang and Foster, 1990] present improved methods. [Tsang, 1993] is an indispensable resource for anyone interested in CSP.

The simple Constraint Satisfaction Problem presented in the introduction is a good example of the pitfalls of uninformed backtracking. Figure 2 displays the ever-present fact-of-life inherent in uninformed search. In that problem, seven combinations of A,B and C were tested before one was found that met the input constraints. If all answers to the problem were required, an exhaustive search (18 combinations in this case) would be required. All of this despite the fact that no search was required at all! By applying the basic principles of CSPs, the single correct answer falls out without search. This is not to say that search is never required; in most problems it is. But for many types of problems, using CSP techniques can drastically reduce the amount of processing needed.

DEFINITIONS

The following terminology will be used throughout this paper. A CSP consists of a set of **variables**, also called **nodes** . Each variable can take on a **value** taken from a set of values, called its **domain**. A variable's domain will always be presented between curly braces { }. An assignment of a value to a variable will be written <A,2>, meaning variable A has value 2. There are two types of constraints. **Unary constraints** restrict the domain of a variable without reference to any other variable. For instance, [A > 3] is a unary constraint for variable A. It will be assumed that there is one (or zero) unary constraint per variable.[5]

[5]If there is more than one, they can always be combined into one with ANDS: [A > 0 AND A < 10]

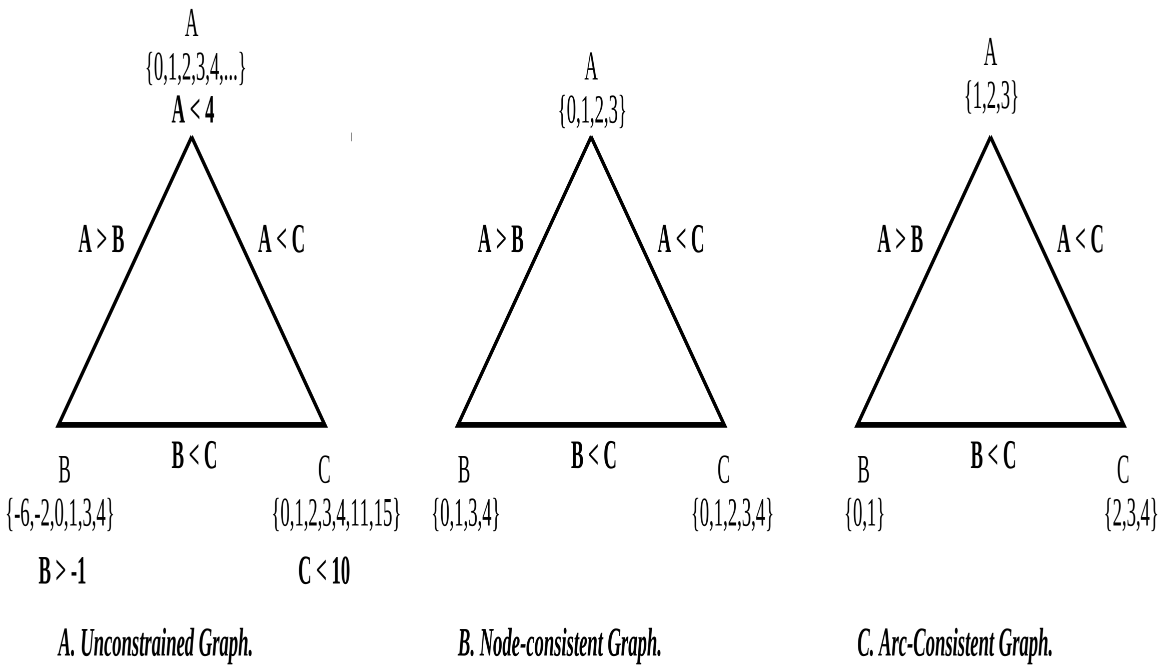

Figure 6: Constraint Satisfaction Problem - Consistency

Binary constraints[6] restrict the values a variable can take by comparing it to another variable. For instance, [A < B] is a binary constraint between A and B. Actual constraints will always be presented between straight brackets [ ]. They will also be represented as follows: C_A is a unary constraint for A, C_{AB} is a binary constraint between the arc AB. An **arc** is defined to be any pair of variables for which there is a binary constraint. It is assumed there is only one binary constraint per arc. In general, capital letters at the beginning of the alphabet (A,B,C ...) will represent variables, while those at the end (X,Y,Z) will represent unknown values.

A solution to a CSP is an assignment to each variable of a value taken from that variable's domain, such that all the unary and binary constraints are satisfied. A complete solution is the set of all such solutions for a CSP. A solution will be represented as a set of values enclosed within parentheses, such as (1,2,4,0), where 1 is the value of the first variable (generally A in these abstract examples), 2 is the value of the second variable, etc.. Alternatively, a solution sometimes is represented more explicitly as a set of variable-value assignments such as (<A,1>,<B,2>,<C,4>,<D,0>). A set of solutions will be a list of solutions enclosed in curly brackets: {(1,2,4,0),(1,2,4,1), ... }. Partial solutions will be represented similarly, with the assignment of values to variables determined by context. The fact that a partial solution satisfies a given constraint will be represented (X,Y) SAT C_{AB} for binary constraints,[7] and X SAT C_A for unary constraints.[8]

[6]Higher order constraints are also possible, but since they currently not used in Mikrokosmos, they will not be discussed here.

[7]Meaning that the partial solution which assigns value X to variable A and Y to B satisfies the binary constraint C_{AB}.

[8]The assignment of value X to variable A satisfies the unary constraint C_A.

Figure 6A is a slightly more complex CSP. A few more possible values have been added to the domains of each variable to more clearly demonstrate the types of consistency discussed below. Unary constraints are also included. The three central "consistency" checks used for CSPs are node consistency, arc consistency and path consistency.

1. Node Consistency.

   Node consistency (NC) is simply a state where the domains of each variable have been reduced to the set of all possible values that satisfy the unary constraints. In Figure 6A-B, the results of node consistency processing are displayed. For instance, the values 11 and 15 are removed from the domain of C since they do not satisfy C's unary constraint that C $<$ 10.

   Node consistency can often be assumed, either because there are no unary constraints, or because the unary constraints were used in setting up the problem. For instance, in the Mikrokosmos semantic analyzer, the unary constraints correspond to picking the correct lexicon entries based on the root word and surrounding syntax.

   A simple algorithm for enforcing node consistency is as follows:

```
1 PROCEDURE NC-1
2     FOR each variable, V
3         $C_V$ <- unary constraint for V
4         IF $C_V$ is null ;; no unary constraints
5             THEN do nothing
6             ELSE
7                 FOR each value in V's domain, X
8                     IF X SAT $C_V$
9                         THEN do nothing
10                        ELSE remove X from the domain of V
```

   This algorithm has time-complexity *O(an)*, where a is the maximum size of the domains and n is the number of variables to be examined. In all that follows, our primary goal will be to reduce the time complexity with respect to n, the number of variables.[9] Thus, in this respect, NC-1 is linear in time.

2. Arc Consistency.

   Arc consistency (AC) ensures that the binary constraints connecting any two nodes are satisfied. There have been various implementations of AC. AC-1, AC-2 and AC-3 are presented in [Mackworth 1977]. [Mohr and Henderson 1986] present the optimal AC-4 algorithm. The naive approach, AC-1, is given below:

```
1 PROCEDURE AC-1
2     NC-1 ;; ensure NC first
3     REPEAT
```

[9]In computational semantics, the maximum size of any domain is probably around 10 (10 word senses for a single lexical item). However, the average case domain size is less than 3. Thus, terms which involve domain size will be insignificant compared to the number of variables (actual words) in a problem, which can get to 40 or higher in long sentences. Later, time and space complexity terms involving the maximum number of constraints per variable will be used. A similar argument applies.

```
4       Changed <- false
5       FOR each arc, AB
6           $C_{AB}$ <- binary constraint between A and B
7           FOR each value in the domain of A, X
8               IF there is a value in the domain of B, Y
                such that (X,Y) SAT $C_{AB}$
9                   THEN do nothing
10                  ELSE
11                      REMOVE X from the domain of A
12                      Changed <- true
13  UNTIL NOT Changed
```

This algorithm cycles through each potential arc[10] to determine if for each value in the domain of the first variable there exists a value in the domain of the second variable that can satisfy the binary constraint on that arc. If a value is found for which this is not true, it is removed from the domain of the first variable. If any values are removed, the process must be started again from the beginning, since the removal might cause some arc-inconsistency in an arc that was already tested.

In Figure 6B-C, the arc-consistency results are shown. For example, in Figure 6B, the domain of B is $\{0,1,3,4\}$. When the arc AB is tested with the binary constraint [$A > B$] and the value 0 for variable A is tested, it is discovered that no values of B remain such that [$0 > B$]. Thus 0 must be removed from the domain of A. Note that this will impact on the testing of arc CA. When the value 1 is tested for C, no values for A remain in its revised domain $\{1,2,3\}$ such that [$A < 1$]. Thus 1 will need to be removed from the domain of C. This demonstrates the necessity of testing all arcs again after any value is removed, since if arc CA is tested before AB, the value of 1 for C is still consistent.

If a is the maximum size of the domains, e is the number of arcs and n is the number of variables, then there will be at most na values to be checked. If, in the worst case, one value is deleted for each iteration in the UNTIL loop, then the loop will be run na times. The first FOR loop (line 5) can be run a maximum e times per iteration of the REPEAT loop. The second FOR loop (line 7) can be called a maximum of a times per iteration of the first FOR loop, and the IF statement in line 8 may involve a maximum of a searches through the domain of B per each iteration of the second FOR loop. Thus, the complexity of the second FOR loop is $O(a^2)$, the complexity of the first FOR loop is $O(ea^2)$, and the complexity of the entire procedure is $O(nea^3)$. In unconstrained graphs, e can be at most n^2. However, in computational semantics, typically, e is proportional to linear n. Thus AC-1 typically has time-complexity $O(n^2a^3)$, which is not linear with respect to the number of variables.

Various improvements to AC-1 have been offered. AC-4 [Mohr and Henderson, 1986] is the optimal algorithm, offering time complexity of $O(ea^2)$. In AC-1, whenever a value is removed from a variable's domain, every arc has to be rechecked. AC-4 recognizes the fact that a value in a variable's domain "supports" an identifiable list of values in other variables. For instance, in Figure 6C, the value 0 in B "supports" the value 1 in A. The constraint [$A > B$] is enabled by this support. Remove the value 0 from B's domain and the value 1 in A is no longer supported. Therefore, if a value is deleted in one variable, only the values in other variables that are supported by the deleted value must be checked. Furthermore, the

[10]Note: the arcs must be identified before using this procedure. This can be accomplished by examining each constraint once. The algorithm above assumes separate arcs for both directions, i.e. AB and BA.

number of "supports" for a certain value can be tracked. Examining Figure 6C again, the two values in B, 0 and 1, **both** support the value 2 in A. This will be represented by saying SUPPORTS(B,0) = {(A,2,C_{AB}),...}[11] and SUPPORTS(B,1) = {¡A,2¿,...}. The fact that the value 2 in A has two values in B supporting C_{AB} is represented as *SUPPORT(A,2,C_{AB}) = 2*.[12] If one of the supporting values in B was removed, *SUPPORT(A,2,C_{AB}) = 1* would still be supported by one value. Thus, by recording how many supports each value-constraint pair has initially, and by subtracting one from that total each time a supporting value is removed, it can be determined when a value-constraint pair no longer can be supported. Whenever a value-constraint pair has no supports, the value must be deleted. An algorithm for AC-4 follows:

```
1 PROCEDURE AC-4
2    NC-1 ;; ensure node consistency first

     ;; Step I: INITIALIZATION
3    INITIALIZE array SUPPORT[V,X,AB] = 0
4    INITIALIZE SUPPORTS[V,X] = nil
5    INITIALIZE FAIL-LIST = nil

     ;; Step II: SETTING UP SUPPORT VARIABLES
6    FOR each arc, AB
7       FOR each value X in variable A
8          FOR each value Y in variable B
9             IF (A,B) SAT $C_{AB}$
10               THEN
11                  INCREMENT(SUPPORT[A,X,$C_{AB}$])
                    ;; increment the number of supporters for X
12                  APPEND(SUPPORTS[B,Y],(A,X,$C_{AB}$))
                    ;; record that Y supports X
13         IF (SUPPORT[A,X,$C_{AB}$] = 0)
           ;; X has no supports for this arc
14            THEN
15               APPEND(FAIL-LIST,(A,X))
16               REMOVE X from the domain of A

     ;; Step III: REMOVING UNSUPPORTED VALUES
17   WHILE FAIL-LIST <> { }
18       PICK first label (A,X) from FAIL-LIST
19       FAIL-LIST = FAIL-LIST - (A,X)
20       LABELS-SUPPORTED <- SUPPORTS[A,X]
21       FOR each label (B,Y,$C_{BA}$) in LABELS-SUPPORTED
         ;; determine if the supported value has any other supports left
22          DECREMENT(SUPPORT[B,Y,$C_{BA}$])
23          IF ((SUPPORT[B,Y,$C_{BA}$] = 0)
```

[11]That is, the assignment of value 0 to variable B supports the assignment of value 2 to variable A with respect to C_{AB}, along with any other supports.

[12]That is, the assignment of the value 2 to A has two supports for the C_{AB} constraint.

```
           AND NOT-MEMBER((B,Y),FAIL-LIST) ;; not pending failure already
           AND MEMBER(Y,DOMAIN(B))) ;; not already failed
24             APPEND(FAIL-LIST,(B,Y))
25             REMOVE Y from the domain of B
```

The INITIALIZATION stage is trivial, and, practically speaking, can be combined with step II. In step II, if the number of arcs is e, the maximum number of values in a variable is a, then it is easy to see that the inner-most IF statement (line 9) is executed a^2e times. In step III, in the worst case, the WHILE loop will be executed once for each value. If the number of variables is n, then there can be at most an values to be deleted. Finally, there can be at most a labels to be examined in the inner FOR loop (line 21), so that the total complexity of step III is $O(a^2n)$.[13] Note that n, while proportional to e, is always less than or equal to it in connected graphs. Therefore, the total complexity of AC-4 is $O(a^2n + a^2e) = O(a^2e)$. Again, in computational semantics e is typically proportional to n; thus, thus the time complexity for AC-4 is typically $O(a^2n)$.

Because space becomes an issue in AC-4, the space complexity will also be analyzed. The space complexity is dominated by the SUPPORTS array. There can be at most one entry in SUPPORTS for every variable-value pair, or an entries. Each entry can support every other variable-value pair, again an. Since there can be an entries, each of can hold up to an bits of information, the total complexity is $O(a^2n^2)$. Since e is equivalent to n^2, this is the same as $O(a^2e)$, which is what is reported in the literature. Again, it should be pointed out that in computational semantics, instead of every variable-value pair supporting every other variable-value pair (an), typically it is more on the line of ca, where c is a constant. Thus, implementations of AC-4 for natural language semantics typically have a space complexity of $O(a^2n)$, which is linear with respect to the number of variable.

3. Path Consistency.

Note that Figure 6C still contains some value combinations that theoretically could be removed ahead of time. For instance, the partial solution (B,C) = (1,2) is never possible. When B = 1 is assigned, A must be constrained to {2,3} to meet the [A > B] constraint. However, with C = 2, the [A < C] constraint could no longer be satisfied. Path consistency is an attempt to eliminate impossible partial solutions.

Path consistency requires that for any path[14] (A,B,C,...M), then for any value assignment A = X and M = Y, there must exist a value assignment for each of the variables B,C,...,L such that all binary constraints on **adjacent** variables are satisfied. In Figure 6C, the path in question is (B A C). When we assign B = 1 and C = 2, there is no value X for A left such that (X,B) SAT C_{AB} **AND** (X,C) SAT C_{AC}.

Path consistency (PC) algorithms can be found in [Tsang, 1993] and [Mohr and Hendersen, 1986]. The optimal algorithm runs with time complexity $O(a^3n^3)$ and space complexity $O(a^3n^3)$. Obviously, these complexities are much higher than those for AC-4 and NC-1.

[13]Note that [Tsang, 1993] as well as [Mohr and Henderson, 1986] give the complexity of step 3 as $O(a^2e)$. They assume each arc can add a items to the FAIL-LIST, so that the WHILE loop can be executed ae times. It is easy to see, though, that this is excessive. A value can only be deleted once from its domain. The APPEND(FAIL-LIST,...) statements (lines 15 and 24) can be implemented so that duplicate variable-value pairs are not added, and the MEMBER(Y,DOMAIN(B)) statement (in line 23) prevents any variable-value pair that has been deleted from being re-added to FAIL-LIST. Thus, an is the correct limit for the WHILE loop.

[14]A path is a set of variables, each variable of which has an arc between itself and its adjacent variables.

Fortunately, PC algorithms are not necessary. A dynamic application of AC algorithms performs the same function. To illustrate, assume that a pre-processor performs AC-4 on Figure 6A, yielding the CSP in Figure 6C. Next, submit the CSP to a AC-enabled search algorithm. Such an algorithm would choose one value for B, say B = 1. This, in effect, creates an "artificial island",[15] with the domain of B = $\{1\}$. Since we have removed the value 0 from the domain of B, the AC-enabled search can dynamically eliminate values in A and C which are supported only by B = 0. This would remove A = 1. After removing A = 1, the AC-enabled search would also eliminate C = 2. Thus, a dynamic AC-enabled search algorithm performs the same function as PC algorithms. The algorithm for an AC-enabled algorithm will be given below.

[15]Discussion on "islands" follows below.

2.2. Solution Synthesis

Solution synthesis is a method used to generate all solutions to a CSP. That is, all assignments of values to variables that satisfy the problem's constraints are produced by a solution synthesis algorithm. Often, this set of solutions can be further judged according to some separate criteria to obtain the optimal answer. For instance, in a modified traveling salesperson problem in which all cities must be visited, but in an order subject to certain constraints,[16] solution synthesis can generate the list of all possible answers that meet the constraints, from which the most optimal answer (presumably the one with the least total mileage) could be picked. [Freuder, 1978] introduced solution synthesis and [Tsang and Foster, 1990] refined it. Both of their research will be described below. We extend the work of Tsang and Foster and combine it with branch-and-bound techniques, all of which will be described in the "Hunters and Gatherers in Computational Semantics" section.

Solution synthesis constructs the set of all possible solutions by iteratively combining smaller solutions while propagating constraints. A simplified algorithm[17] which implements solution synthesis follows:

GENERAL ALGORITHM

```
1 PROCEDURE SS-1
      ;; initialize SOLUTION SET to set of order-1 solutions
2     SOLUTION-SET <- nil
3     FOR each variable, V
4         FOR each value, X in the domain of V that meets unary constraint $C_V$
5             SOLUTION-SET = SOLUTION-SET + ($<V, X>$)

6     FOR I = 2 to n, where n = number of variables
          ;;Create all partial solutions of length I
          ;;At this point, SOLUTION-SET contains all solutions of
          ;; length I-1
7         SOLUTION-SET <- SYNTHESIZE(SOLUTION-SET,I)

8 PROCEDURE SYNTHESIZE(SOLUTION-SET,I)
9     SOLUTION-SET-TEMP <- nil
10    FOR each SOLUTION-A in SOLUTION-SET
11        FOR each SOLUTION-B in SOLUTION-SET
12            IF SOLUTION-A and SOLUTION-B have I distinct variables
b             AND all like-variable assignments are the same THEN
13                POSSIBLE-SOLUTION <- UNION(SOLUTION-A,SOLUTION-B)
14                IF POSSIBLE-SOLUTION meets all I-ARY CONSTRAINTS
b                 AND NOT-MEMBER(POSSIBLE-SOLUTION,SOLUTION-SET-TEMP)
c                 AND SYNTHESIZED?(POSSIBLE-SOLUTION,SOLUTION-SET,I) THEN
15                    SOLUTION-SET-TEMP <- SOLUTION-SET-TEMP + POSSIBLE-SOLUTION
16    RETURN SOLUTION-SET-TEMP
```

[16]For example, Baltimore must come first because the supplies need to be picked up, then the cities where the perishable goods are sold must be next, etc.

[17]Note that this is simplified; Freuder's algorithm will be described below.

```
17 PROCEDURE SYNTHESIZED?(POSSIBLE-SOLUTION,SOLUTION-SET,I)
       ;; Make sure all sub-solutions of length I-1 of the new solution
       ;; are in SOLUTION-SET (which contains all valid solutions of order I-1.)
18     OK <- true
19     FOR all COMBOs of I-1 $< V, X >$ pairs in POSSIBLE-SOLUTION
20         IF NOT-MEMBER(COMBO,SOLUTION-SET) THEN
21             OK <- false
22     RETURN OK
```

Before giving an example, a walkthrough of the algorithm would be instructive. At the most abstract level, SS-1 creates partial solution sets of order k[18] by combining solution sets of order k-1. The initial solution set of order 1 is created in lines 3-5; a solution is added for each value of each variable. From there, solution sets of higher order are created using SYNTHESIZE (lines 8-16).

In SYNTHESIZE, solution sets of order I are created. The input SOLUTION-SET contains all solutions of order I-1. POSSIBLE-SOLUTIONs are created by combining compatible solutions of order I-1 that differ only in one variable.[19] For example, a solution of variables (A,B,C) combined with a solution of variables (A,B,D) would combine to create a solution of variables (A,B,C,D).[20] "Compatible" combinations are those which have like-variables assigned similarly. For example, $(< A, 1 >, < B, 2 >)$ can combine with $(< A, 1 >, < C, 3 >)$ to create a POSSIBLE-SOLUTION $(< A, 1 >, < B, 2 >, < C, 3 >)$, but $(< A, 1 >, < B, 2 >)$ cannot combine with $(< A, 4 >, < C, 3 >)$ because the assignments $< A, 1 >$ and $< A, 4 >$ are not compatible.[21]

Each POSSIBLE-SOLUTION must meet three tests (lines 14a-c). First, all I-ary constraints must be met. For instance, for an order 2 POSSIBLE-SOLUTION involving variables A and B, the constraints C_{AB} and C_{BA} must be met. Order 3 possible solutions must meet 3-ary constraints, and so on. In computational semantics, there are only binary constraints, so order 3 solutions and above will always pass this test. Second, line 14b simply ensures that duplicate solutions are not added. For instance, when adding variable C onto a partial solution (A,B), solutions for (A,B,C) are obtained. Later, when adding on variable B to (A,C), the same solutions would be obtained. Line 14b prevents this. Line 14c ensures that solutions of length I meet the constraints already calculated for solutions of length I-1. This is the synthesizing step. A simple example will illustrate. In order to allow a POSSIBLE-SOLUTION $= (< A, 0 >, < B, 1 >, < C, 2 >)$, a solution of order 3, the following order-2 solutions must exist:

$$\{(< A, 0 >, < B, 1 >), (< A, 0 >, < C, 2 >), (< B, 1 >, < C, 2 >)\}$$

In other words, an order-N solution cannot have any sub-solutions of order N-1 that were not already identified. Because of the way order I-1 solutions are combined, most of these sub-solutions

[18]A solution of order k is an assignment of values to k variables; a solution set of order k is the set of all solutions of order k.

[19]Two solutions, each of order I-1, which differ in only one variable, will combine to give a solution of order I.

[20]Line 12a guarantees SOLUTION-A and SOLUTION-B differ by one variable. Line 13 performs the combination.

[21]Line 12b checks for compatibility.

will be present.[22] For instance, in the example above a solution of variables (A,B) was combined with a solution of variables (B,C) to create a solution of variables (A,B,C). Thus, we do not need to check the (A,B) or (B,C) sub-solutions. However, the sub-solution involving variables (A,C) was synthesized and must be checked. If, for this example, an order I-1 solution $(< A, 0 >, < C, 2 >)$ does not exist, then this combination of values is unacceptable. The SYNTHESIZED? procedure in lines 17-22 performs this check. Again, this step ensures that any new order-I solution only contains order-(I-1) sub-solutions that had already been deemed legitimate.

In order to start moving this discussion towards natural language semantics, the solution synthesis algorithms will be exemplified using a simple computational semantic problem. Consider the following sentence:

(1) IBM acquired Jacob-Smith for ten-million-dollars.

For simplicity, assume that Jacob-Smith[23] and ten-million-dollars are phrasal entries in the lexicon. Also assume that we have an ontology,[24] or model of the world, that maps each of these words into the concepts[25] as shown in Figure 7. *IBM* (referred to below as I) maps into a single concept, **ORG**. *acquired* (referred to as A) maps into two possible concepts. The first, **TAKE-OVER**, constrains *IBM* to be an **ORG** and *Jacob-Smith* to be an **ORG**. Refer to Figure 7 for the rest of the possible assignments and constraints.

The SS-1 algorithm would proceed as follows. First, it would initialize SOLUTION-SET to the set of all order-1 nodes in lines 2-5. For convenience, we will present SOLUTION-SET as a group of subsets arranged according to the variables involved:[26]

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T - O >), (< A, OBT >)\}$
$N_J = \{(< J, HUM >), (< J, ORG >)\}$
$N_F = \{(< F, COST >), (< F, BEN >), (< F, PUR >), (< F, DUR >)\}$
$N_T = \{(< T, MON >)\}$

For example, N_A has two possible solutions, the first of which assigns the concept **TAKE-OVER** (T-O) to A (*aquired*).

If necessary, unary constraints would be applied in line 3. In computational semantics, unary constraints correspond to selecting the appropriate word-senses from the lexicon based on the word used and the surrounding syntax. In this case, those constraints were applied before beginning the algorithm.

[22]It is possible to optimize this algorithm so that only non-obvious sub-solutions are checked.

[23]An imaginary company name, as well as a person's name.

[24]See (Mahesh & Nirenburg, 1995) for a discussion of ontologies and their place in computational semantics

[25]Concepts will be in CAPS. We use very simple concepts symbolized with english words, and simply list the constraints that would be specified in an ontology. We also assume an inheritance hierarchy (HUMAN IS-A ANIMATE), and a mechanism for identifying metonymy; for instance (ORG IS-A ANIMATE) because an ORG has MEMBERS that are HUMAN, and (ORG IS-A INANIMATE), because an ORG has a BUILDING that is an INANIMATE. Metonymy will be discussed in more depth below.

[26]Here, for clarity, we present each solution as a set of variable-value pairs, i.e $N_I = \{(< I, ORG >)\}$. After this, we generally will present only the values, with the assignment to variables obvious in context, i.e $N_I = \{(\text{ORG})\}$.

```
WORD     CONCEPT          CONSTRAINTS             EXAMPLE

IBM (I)
         ORG
acquired (A)
         TAKE-OVER (T-O) [I=ORG TAKE-OVER J=ORG]
         OBTAIN (OBT)    [I=ANIMATE OBTAIN J=INANIMATE]
Jacob-Smith (J)
         HUMAN (HUM)
         ORG
for (F)
         COST            [A=EVENT FOR T=MONEY]  I bought it for 10 dollars.
         BENEFIC (BEN)   [A=EVENT FOR T=ANIMAL] I bought it for Sam.
         PURPOSE (PUR)   [T=EVENT FOR T=EVENT]  I bought it for mowing the lawn.
         DURATION (DUR)  [T=EVENT FOR T=TIME]   I hid for 10 hours.
ten-million-dollars (T)
         MONEY (MON)
```

Figure 7: Concept assignments for SS example.

Before line 6 is executed, then:

SOLUTION-SET = APPEND(N_I, N_A, N_J, N_F, N_T)

Next, all higher order nodes, including the final solution, are created in steps 6 and 7. First, order-2 nodes are constructed. When SYNTHESIZE is called with I=2, SOLUTION-SET contains all the solutions of order-1. In SYNTHESIZE, these order-1 solutions are combined into order-2 solutions. For instance, if, in lines 10 and 11:

SOLUTION-A = ($< I, ORG >$)
SOLUTION-B = ($< A, T - O >$)

Then, together, A and B have two distinct variables (I and A), checked in line 12, there are no like-variables, so line 12 is true, so in line 13:

POSSIBLE-SOLUTION = ($< I, ORG >, < A, T - O >$)

Line 14a is important only when I=2, as in this case, because computational semantic problems only have binary constraints. Two binary constraints must be examined, C_{IA} and C_{AI}. As shown in Figure 7, I does not constraint A for any value of A, but $< A, T - O >$ constrains I to be of type ORG. Since $< I, ORG >$ obviously meets this constraint, line 14a is true. No other solutions have been added for order-2 solutions, so line 14b is also true. Finally, since the only two sub-solutions of POSSIBLE-SOLUTION of order 1, ($< I, ORG >$) and ($< A, T - O >$), came directly from SOLUTION-SET, SYNTHESIZED? will obviously return true. In fact, SYNTHESIZED? will always be true for POSSIBLE-SOLUTIONS of order-2. It only becomes relevant for order-3 and higher nodes, when previously non-existent lower order sub-solutions are possible.

An example of when the binary constraint in line 14a rejects a POSSIBLE-SOLUTION occurs for

SOLUTION-A = $(< A, T - O >)$
SOLUTION-B = $(< J, HUM >)$

which yields

POSSIBLE-SOLUTION = $(< A, T - O >, < J, HUM >)$

The binary constraint, C_{AJ}, specifies that for the assignment $< A, T - O >$, J must be an ORG. However, the assignment $< J, HUM >$ does not meet this constraint, so this POSSIBLE-SOLUTION is rejected.

A complete listing of order-2 solutions for this example is shown below, with solutions rejected by binary constraints identified:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{IJ} = {(ORG,HUM),(ORG,ORG)}
N_{IF} = {(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{IT} = {(ORG,MON)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} ;; eliminate (T-O,HUM),(OBT,HUM)
N_{AF} = {(T-O,COST),(T-O,BEN),(T-O,PUR),(T-O,DUR),
(OBT,COST),(OBT,BEN),(OBT,PUR),(OBT,DUR)}
N_{AT} = {(T-O,MON),(OBT,MON)}
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{JT} = {(HUM,MON),(ORG,MON)}
N_{FT} = {(COST,MON)} ;; eliminate (BEN,MON),(PUR,MON),(DUR,MON)

Order-3 nodes are then synthesized by combining order-2 solutions. Note that from this point on, no reference needs to be made to constraints, because all the binary constraints information is implicit in the order-2 solution set. An example of synthesisizing an order-3 solution follows:

SOLUTION-A (from N_{IA}) = (ORG,T-O)
SOLUTION-B (from N_{AJ}) = (T-O,ORG)

Together, A and B have three distinct variables (I, A and J), and the only like-variable, A, has the same value assignment in both, so:

POSSIBLE-SOLUTION = $(< I, ORG >, < A, T - O >, < J, ORG >)$

Because there are no n-ary constraints for n $>$ 2, line 14a will be true from here on. It will also be assumed, starting now, that line 14b will prevent duplicate solutions from being added. Thus, the SYNTHESIZED? procedure called in line 14c is the only part left needing comment. In this case, the following sub-solutions of order-2 taken from POSSIBLE-SOLUTION are:

$(< I, ORG >, < A, T - O >)$
$(< A, T - O >, < J, ORG >)$
$(< I, ORG >, < J, ORG >)$

The first two come directly from SOLUTION-A and B. The third, however, was a result of the synthesis; therefore, it must be checked to see if it is a valid order-2 solution. A quick look at the list of order-2 solutions shows that this solution is present in N_{IJ}. Therefore, POSSIBLE-SOLUTION is a valid synthesis.

An example of a POSSIBLE-SOLUTION that does not meet the SYNTHESIZED? criterion occurs for:

SOLUTION-A (from N_{JF}) = (HUM,BEN)
SOLUTION-B (from N_{JT}) = (HUM,MON)

This gives:

POSSIBLE-SOLUTION = $(< J, HUM >, < F, BEN >, < T, MON >)$

Three sub-solutions of order-2 can be extracted from POSSIBLE-SOLUTION:

$(< J, HUM >, < F, BEN >)$
$(< J, HUM >, < T, MON >)$
$(< F, BEN >, < T, MON >)$

Again, the first two come directly from SOLUTION-A and B. The third, however, was synthesized. This time, though, the synthesized sub-solution cannot be found in the list of order-2 solutions, thus it cannot be allowed.

A complete list of order-3 solutions follows:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{IAF} = {(ORG,T-O,COST),(ORG,T-O,BEN),(ORG,T-O,PUR),(ORG,T-O,DUR),
(ORG,OBT,COST),(ORG,OBT,BEN),(ORG,OBT,PUR),(ORG,OBT,DUR)}
N_{IAT} = {(ORG,T-O,MON),(ORG,OBT,MON)}
N_{IJF} = {(ORG,HUM,COST),(ORG,HUM,BEN),(ORG,HUM,PUR),(ORG,HUM,DUR),
(ORG,ORG,COST),(ORG,ORG,BEN),(ORG,ORG,PUR),(ORG,ORG,DUR)}
N_{IJT} = {(ORG,HUM,MON),(ORG,ORG,MON)}
N_{IFT} = {(ORG,COST,MON)}
N_{AJF} = {(T-O,ORG,COST),(T-O,ORG,BEN),(T-O,ORG,PUR),(T-O,ORG,DUR),
(OBT,ORG,COST),(OBT,ORG,BEN),(OBT,ORG,PUR),(OBT,ORG,DUR)}
N_{AJT} = {(T-O,ORG,MON),(OBT,ORG,MON)}
N_{AFT} = {(T-O,COST,MON),(OBT,COST,MON)}
N_{JFT} = {(HUM,COST,MON),(ORG,COST,MON)}

It is interesting to note that the solution sets N_{IAF}, N_{IJF} and N_{AJF} all carry along values for F that, with a little bit of thought, could be eliminated. In N_{IJF}, N_{IJT} and N_{JFT}, inconsistent values for J are also kept. Freuder's algorithm, as well as Tsang's and our own, eliminate this.

Order-4 solutions are formed similarly:

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,T-O,ORG,BEN),(ORG,T-O,ORG,PUR),(ORG,T-O,ORG,DUR),
(ORG,OBT,ORG,COST),(ORG,OBT,ORG,BEN),(ORG,OBT,ORG,PUR),(ORG,OBT,ORG,DUR)}
N_{IAJT} = {(ORG,T-O,ORG,MON),(ORG,OBT,ORG,MON)}
N_{IAFT} = {(ORG,T-O,COST,MON),(ORG,OBT,COST,MON)}

N_{IJFT} = {(ORG,HUM,COST,MON),(ORG,ORG,COST,MON)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

Finally, the order-5 solutions are synthesized:

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

At each stage, higher order solutions are created by combining lower order solutions, while ensuring no unacceptable lower-order sub-solutions are introduced. The order-n solution set contains all of the valid answers for the problem.

Although various improvements to this algorithm will be offered, all solution synthesis algorithms will have the same worst-case time complexity. Assume a CSP for which all combinations of values for every variable meet all constraints. Furthermore, assume each variable has a values in its domain. For such a CSP, there exist a^n solutions. It is easy to see, therefore, that line 15 must be executed a^n times when the SYNTHESIZE procedure is called the last, n^{th} time. It is the nature of CSPs that worst-case (algorithmic) time behavior is always exponential because the number of possible solutions is always, theoretically, exponential. However, "worst-case" can be redefined non-algorithmically, and with respect to a certain class of problems, to mean the "typical" worst class complexity as measured experimentally. Such measurements will be described below.

FREUDER'S ALGORITHM

Freuder's algorithm [Freuder, 1978] eliminates some of the waste present in SS-1. Instead of only propagating constraints upward from lower-order nodes to higher order nodes, Freuder also propagates constraints downward, from higher order nodes to lower order nodes.

An example would clarify best. Suppose the following order-1 solutions were found for a simple CSP:

N_A = {(1),(2)}
N_B = {(3),(4)}
N_C = {(5),(6)}
N_D = {(7)}

Assume binary constraints were then used to acquire the following order-2 solutions:

N_{AB} = {(1,3),(2,3)}
N_{AC} = {(1,5),(1,6),(2,5)}
N_{AD} = {(1,7),(2,7)}
N_{BC} = {(3,5),(4,5),(4,6)}
N_{BD} = {(3,7),(4,7)}
N_{CD} = {(5,7),(6,7)}

At this point, it can be deduced that an assignment $< B, 4 >$ will never be compatible with any assignment for A. SS-1, however, blindly constructs the order-3 solutions:

N_{ABC} = {(1,3,5),(1,3,6),(2,3,5)}
N_{ABD} = {(1,3,7),(2,3,7)}
N_{BCD} = {(3,5,7),(4,5,7),(4,6,7)}

Clearly, (4,5,7) and (4,6,7) in N_{BCD} are impossible. SS-1 eventually gets the correct answer because it cannot construct any order-4 solutions with the assignment $< B, 4 >$, but it wastes much effort in doing it.

Freuder suggests allowing downward propagation of constraints. For instance, once it can be determined that a solution set of order I contains no instances of a particular sub-solution of order I - 1, that sub-solution can be eliminated from the order I - 1 solutions. Above, since the sub-solution ($< B, 4 >$) does not occur in N_{AB}, ($< B, 4 >$) can be removed from the order-1 solutions:

N_B = {(3)} ;; removed (4)

Whenever a solution is removed, all higher order solutions involving that solution must also be removed (upward propagation). In addition, whenever a solution is removed, downward propagation can occur again. In the example above, solutions of order-2 involving $< B, 4 >$ must be removed, giving:

N_{BC} = {(3,5)} ;; removed (4,5) and (4,6)
N_{BD} = {(3,7)} ;; removed (4,6)

Before creating new order-3 solutions, downward propagation can be repeated, since the assignment $< C, 6 >$ no longer is compatible with any assignment of B. Therefore, removing (6) from N_C yields:

N_C = {(5)}

which can be re-propagated upward to form the level-2 solutions:

N_{AC} = {(1,5),(2,5)} ;; removed (1,6)
N_{CD} = {(5,7)} ;; removed (6,7)

The resulting complete list of order-2 solutions is:

N_{AB} = {(1,3),(2,3)}
N_{AC} = {(1,5),(2,5)}
N_{AD} = {(1,7),(2,7)}
N_{BC} = {(3,5)}
N_{BD} = {(3,7)}
N_{CD} = {(5,7)}

Creating order-3 solutions from this yields the smaller set:

N_{ABC} = {(1,3,5),(2,3,5)}
N_{ABD} = {(1,3,7),(2,3,7)}
N_{BCD} = {(3,5,7)}

from which the order-4 solutions can be calculated easily:

N_{ABCD} = {(1,3,5,7),(2,3,5,7)}

SS-1 needs to be modified to include this downward propagation. One major change is that SOLUTION-SETs for all the previous levels need to be stored. Also, after a solution set is formed, it needs to be analyzed to see if it excludes any sub-solutions of the next lower order. If it does, these sub-solutions will need to be removed from the lower order SOLUTION-SET. Whenever a solution is removed from a SOLUTION-SET, higher order solutions utilizing it must also be removed, and the SOLUTION-SET it was removed from needs to be re-checked for downward propagation possibilities. SS-FREUDER implements these changes:

```
1 PROCEDURE SS-FREUDER
2     INITIALIZE SOLUTION-SET, SOLUTION-VC and SOLUTION-VC-1 arrays
      ;; initialize SOLUTION SET[1] to set of order-1 solutions
3     FOR each variable, V
4         FOR each value, X in the domain of V that meets unary constraint $C_V$
5             SOLUTION-SET[1] = SOLUTION-SET[1] + ($< V, X >$)

6     FOR I = 2 to n, where n = number of variables
          ;;Create all partial solutions of length I
          ;;At this point, SOLUTION-SET[I-1] contains all solutions of
          ;; length I-1
7         SOLUTION-SET[I] <- SYNTHESIZE(SOLUTION-SET[I-1],I)
          ;; set up data arrays to be used in DOWNWARD-PROPAGATE
7b        FOR each SOLUTION in SOLUTION-SET[I]
7c            VAR-COMBO <- the combination of variables used in SOLUTION
7d            SOLUTION-VC[I,VAR-COMBO] <-
                  SOLUTION-VC[I,VAR-COMBO] + SOLUTION
7e            FOR each combination, VAR-COMBO-I-1, of I-1 variables in VAR-COMBO
7f                PARTIAL-SOLUTION <- the sub-solution of SOLUTION
                      involving the variables in VAR-COMBO-I-1
7g                SOLUTION-VC-1[I,VAR-COMBO-I-1] =
                      SOLUTION-VC-1[I,VAR-COMBO-I-1] + PARTIAL-SOLUTION
7h        DOWNWARD-PROPAGATE(SOLUTION-SET,SOLUTION-VC,SOLUTION-VC-1,I)

8 PROCEDURE SYNTHESIZE(SOLUTION-SET-I-1,I)
9     SOLUTION-SET-TEMP <- nil
10    FOR each SOLUTION-A in SOLUTION-SET-I-1
11        FOR each SOLUTION-B in SOLUTION-SET-I-1
12            IF SOLUTION-A and SOLUTION-B have I distinct variables
b             AND all like-variable assignments are the same THEN
13                POSSIBLE-SOLUTION <- UNION(SOLUTION-A,SOLUTION-B)
14                IF POSSIBLE-SOLUTION meets all I-ARY CONSTRAINTS
b                 AND NOT-MEMBER(POSSIBLE-SOLUTION,SOLUTION-SET-TEMP)
c                 AND SYNTHESIZED?(POSSIBLE-SOLUTION,SOLUTION-SET-I-1,I) THEN
15                    SOLUTION-SET-TEMP <-
                          SOLUTION-SET-TEMP + POSSIBLE-SOLUTION
16    RETURN SOLUTION-SET-TEMP

17 PROCEDURE SYNTHESIZED?(POSSIBLE-SOLUTION,SOLUTION-SET-I-1,I)
      ;; Make sure all sub-solutions of length I-1 of the new solution
      ;; are in SOLUTION-SET-I-1 (which contains all valid solutions of order I-1.)
18    OK <- true
19    FOR all COMBOs of I-1 $< V, X >$ pairs in POSSIBLE-SOLUTION
```

```
20          IF NOT-MEMBER(COMBO,SOLUTION-SET-I-1) THEN
21              OK <- false
22      RETURN OK

23 PROCEDURE DOWNWARD-PROPAGATE(SOLUTION-SET,SOLUTION-VC,
                                              SOLUTION-VC-1,I)
        ;; SOLUTION-SET,SOLUTION-VC and SOLUTION-VC-1 are arrays, assume
        ;; we can change values in them
24      FOR each X entry in SOLUTION-VC-1[I,X]
            ;; for a specific combination of I-1 variables, X, first get
            ;; all of the sub-solutions of order-i solutions that use
            ;; that combination
25          SUBSOLUTIONS <- SOLUTION-VC-1[I,X]
            ;; now get all of the order I-1 solutions that use that combination
26          VC-1-SOLUTIONS <- SOLUTION-VC[I-1,X]
            ;; now remove any VC-1-SOLUTIONS
            ;; that are not used as subsolutions in order-I solutions.
27          FOR each VC-1-SOLUTION in VC-1-SOLUTIONS
28              IF NOT-MEMBER(VC-1-SOLUTION,SUBSOLUTIONS) THEN
29a               REMOVE VC-1-SOLUTION from SOLUTION-SET[I-1]
29b               REMOVE VC-1-SOLUTION from SOLUTION-VC[I-1,X]
29c               REMOVE subsolutions of VC-1-SOLUTION from
                      SOLUTION-VC-1[I-1] if necessary
30                UPWARD-PROPAGATE(SOLUTION-SET,I,SOLUTION)
31                DOWNWARD-PROPAGATE(SOLUTION-SET,I-1)

32 PROCEDURE UPWARD-PROPAGATE(SOLUTION-SET,I,SOLUTION)
33      FOR each SOLUTION-I in SOLUTION-SET[I]
34          IF SOLUTION is a sub-solution of SOLUTION-I THEN
35a             REMOVE SOLUTION-I from SOLUTION-SET[I]
35b             VAR-COMBO <- the combination of variables used in SOLUTION-I
35c             REMOVE SOLUTION-I from SOLUTION-VC[I,VAR-COMBO]
35d             REMOVE subsolutions of SOLUTION-I from SOLUTION-VC-1[I] if necessary
36              UPWARD-PROPAGATE(SOLUTION-SET,I+1,SOLUTION-I)
36              DOWNWARD-PROPAGATE(SOLUTION-SET,I)
```

SS-FREUDER makes use of two arrays to store solutions for combinations of variables at each level. Besides this storage, performed in lines 7b-7g, the only major change to the basic algorithm is the addition of line 7h, where DOWNWARD-PROPAGATE is called. DOWNWARD-PROPAGATE determines if any sub-solutions of order I-1 need to be excluded on the basis of the order I solutions. Whenever a solution is removed (lines 29 and 35), two things happen. First, UPWARD-PROPAGATE is called to remove higher-level solutions which are based on it. Second, DOWNWARD-PROPAGATE is called to start the downward propagation process again, because the removal of a solution at level I might enable downward propagation again.

SS-FREUDER is inefficient as written. When a SOLUTION is removed, only those solutions which depend on it need to be rechecked, similar to the work in AC-4. However, it is not the intent here to optimize SS-FREUDER. Freuder himself admits that the algorithm is intractable.

SS-FREUDER is simply a step on the way to TSANG's algorithm, and, ultimately, to our own.

To conclude the description of SS-FREUDER, the computational semantic example presented above will be re-worked. The order-1 solutions would be calculated the same as before, and are repeated here for convenience:

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T - O >), (< A, OBT >)\}$
$N_J = \{(< J, HUM >), (< J, ORG >)\}$
$N_F = \{(< F, COST >), (< F, BEN >), (< F, PUR >), (< F, DUR >)\}$
$N_T = \{(< T, MON >)\}$

The order-2 nodes are then calculated:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{IJ} = {(ORG,HUM),(ORG,ORG)}
N_{IF} = {(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{IT} = {(ORG,MON)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} N_{AF} = {(T-O,COST),(T-O,BEN),(T-O,PUR),(T-O,DUR),
(OBT,COST),(OBT,BEN),(OBT,PUR),(OBT,DUR)}
N_{AT} = {(T-O,MON),(OBT,MON)}
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{JT} = {(HUM,MON),(ORG,MON)}
N_{FT} = {(COST,MON)}

In the DOWNWARD-PROPAGATE step, it will be determined that all assignments $< F, BEN >$, $< F, PUR > and < F, DUR >$ can never be combined with any values for T, and that the assignment $< J, HUM >$ is incompatible with any value of A. These facts will be propagated downward to the order-1 constraints:

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T - O >), (< A, OBT >)\}$
$N_J = \{(< J, ORG >)\}$;; removed $(< J, HUM >)$
$N_F = \{(< F, COST >)\}$;; removed $(< F, BEN >), (< F, PUR >), (< F, DUR >)$
$N_T = \{(< T, MON >)\}$

The removals can then be upward-propagated to order-2 nodes again:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{IJ} = {(ORG,ORG)} ;; removed (ORG,HUM)
N_{IF} = {(ORG,COST)} ;; removed (ORG,BEN),(ORG,PUR),(ORG,DUR)
N_{IT} = {(ORG,MON)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)}
N_{AF} = {(T-O,COST),(OBT,COST)}
;; removed(T-O,BEN),(T-O,PUR),(T-O,DUR),(OBT,BEN),(OBT,PUR),(OBT,DUR)
N_{AT} = {(T-O,MON),(OBT,MON)}
N_{JF} = {(ORG,COST)} ;; removed (HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),

(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{JT} = {(ORG,MON)} ;; removed (HUM,MON)
N_{FT} = {(COST,MON)}

No further downward propagation is possible, so the order-3 solutions are synthesized:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{IAF} = {(ORG,T-O,COST),(ORG,OBT,COST)}
N_{IAT} = {(ORG,T-O,MON),(ORG,OBT,MON)}
N_{IJF} = {(ORG,ORG,COST)}
N_{IJT} = {(ORG,ORG,MON)}
N_{IFT} = {(ORG,COST,MON)}
N_{AJF} = {(T-O,ORG,COST),(OBT,ORG,COST}
N_{AJT} = {(T-O,ORG,MON),(OBT,ORG,MON)}
N_{AFT} = {(T-O,COST,MON),(OBT,COST,MON)}
N_{JFT} = {(ORG,COST,MON)}

From here, order-4 and order-5 solutions are simple:

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,OBT,ORG,COST)}
N_{IAJT} = {(ORG,T-O,ORG,MON),(ORG,OBT,ORG,MON)}
N_{IAFT} = {(ORG,T-O,COST,MON),(ORG,OBT,COST,MON)}
N_{IJFT} = {(ORG,ORG,COST,MON)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

TSANG'S ALGORITHM

Tsang's algorithms, defined in [Tsang and Foster, 1990], also known as the Essex algorithms, improve upon SS-FREUDER by limiting the number of second-order solutions (and thus limiting the number of higher order solutions as well). SS-TSANG sets up an arbitrarily ordered list of the variables, and then constructs second-order solutions only for variable pairs that are adjacent in that list. Third-order solutions are then synthesized from "adjacent" second-order solution sets, etc.. Figure 8 displays the process for the computational semantic example.

SS-TSANG is exactly the same as SS-1, except that the number of order-2 solutions is restricted.[27] This restriction could be easily added by constructing a list of the variables, then only allowing order-2 solutions that involve adjacent variables in the list. Synthesizing higher-order solutions will proceed exactly as in SS-1.[28] Because they are so similar, with the changes fairly obvious, the algorithm will not be presented here.

The computational semantic example solutions would be synthesized in the following manner. Order-1 nodes would be constructed first, exactly the same as before:

[27]SS-TSANG can also be built upon SS-FREUDER if propagation is desired.

[28]Because synthesizing order-3 solutions occurs only between order-2 solutions with 3 distinct variables (SS-1 line 12a), this will only occur between "adjacent" (as shown in Figure 8) order-2 solutions, as all other combinations would lead to 4 distinct variables. Likewise, higher order solutions only are synthesized from adjacent solutions of the next lower order.

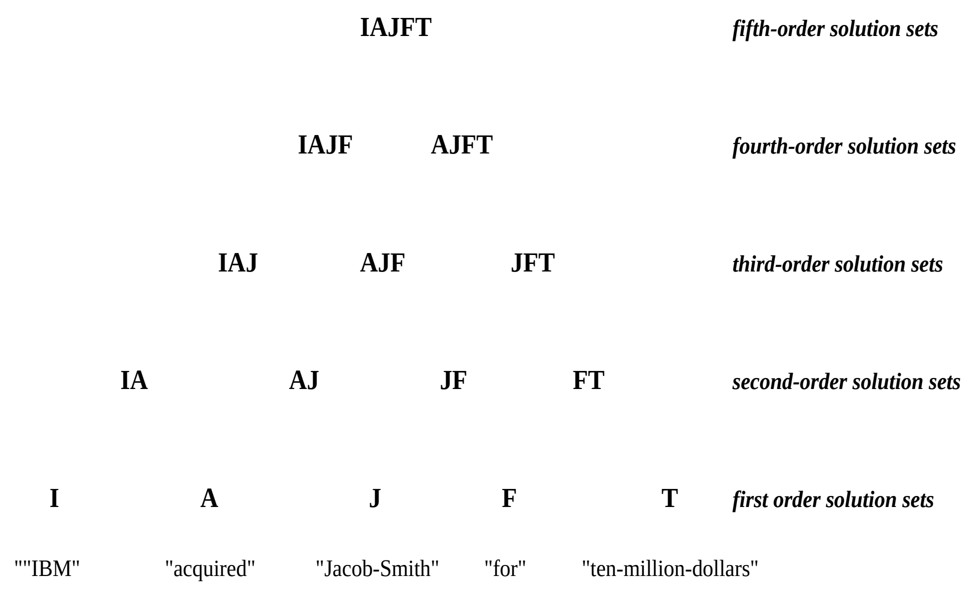
IAJFT
fifth-order solution sets
IAJF
AJFT
fourth-order solution sets
IAJ
AJF
JFT
third-order solution sets
IA
AJ
JF
FT
second-order solution sets
I
A
J
F
T
first order solution sets
""IBM"
"acquired"
"Jacob-Smith"
"for"
"ten-million-dollars"

Figure 8: SS-TSANG Solution Set Construction.

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T-O >), (< A, OBT >)\}$
$N_J = \{(< J, HUM >), (< J, ORG >)\}$
$N_F = \{(< F, COST >), (< F, BEN >), (< F, PUR >), (< F, DUR >)\}$
$N_T = \{(< T, MON >)\}$

Assuming a list of the variables as in Figure 8,[29] the order-2 solution set is constructed the same as for SS-1, except only solutions which utilize adjacent variables are allowed:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} ;; eliminate (T-O,HUM),(OBT,HUM)
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{FT} = {(COST,MON)} ;; eliminate (BEN,MON),(PUR,MON),(DUR,MON)

Level 3 solutions are then synthesized from these:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{AJF} = {(T-O,ORG,COST),(T-O,ORG,BEN),(T-O,ORG,PUR),(T-O,ORG,DUR),
(OBT,ORG,COST),(OBT,ORG,BEN),(OBT,ORG,PUR),(OBT,ORG,DUR)}
N_{JFT} = {(HUM,COST,MON),(ORG,COST,MON)}

Level 4 solutions follow:

[29]This ordering is arbitrary. The efficiency of SS-TSANG is greatly affected by the ordering chosen. This fact is recognized and expanded upon in our work.

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,T-O,ORG,BEN),(ORG,T-O,ORG,PUR),
(ORG,T-O,ORG,DUR),(ORG,OBT,ORG,COST),(ORG,OBT,ORG,BEN),
(ORG,OBT,ORG,PUR),(ORG,OBT,ORG,DUR)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

Finally, the correct solution set is generated:

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

Propagation, as in SS-FREUDER, can be added. For example, once the order-2 solutions are calculated above, it can be deduced that the assignment $< J, HUM >$ is incompatible with all assignments to variable A, and the assignments $< F, BEN >, < F, PUR > and < F, DUR >$ are incompatible with variable T. This information can be propagated downward to order-1 solutions to give:

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T - O >), (< A, OBT >)\}$
$N_J = \{(< J, ORG >)\}$
$N_F = \{(< F, COST >)\}$
$N_T = \{(< T, MON >)\}$

This can then be upward propagated to order-2 solutions to give:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} N_{JF} = {(ORG,COST)}
N_{FT} = {(COST,MON)}

Higher order solutions follow:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{AJF} = {(T-O,ORG,COST),(OBT,ORG,COST)}
N_{JFT} = {(ORG,COST,MON)}

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,OBT,ORG,COST)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

The major benefit of SS-TSANG is that the number of solution sets at each level is minimized. Also important is the fact that this algorithm is ideal for parallel implementations (see [Tsang and Foster, 1990]). There are two disadvantages:

1. Full downward propagation is impossible. Downward propagation in SS-FREUDER relies on having information about allowable combinations of sub-solutions. However, SS-TSANG only determines allowable combinations of adjacent nodes; therefore, propagation can only proceed from these nodes. A limited amount of propagation between adjacent nodes is possible. This drawback is related to the next problem.

2. Ambiguity is carried forward needlessly if two constrained variables are far apart in list. Consider what would happen in the computational semantic example if the ordering of variables was changed to (T I A J F). The following order-2 solutions would be obtained:

   N_{TI} = {(MON,ORG)} N_{IA} = {(ORG,T-O),(ORG,OBT)}
   N_{AJ} = {(T-O,ORG),(OBT,ORG)} ;; eliminate (T-O,HUM),(OBT,HUM)
   N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
   (ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}

   Because the variable T is no longer adjacent to F, it will not be able to disambiguate it. Propagation will be of no help either, since it is dependent on discovering that certain values of F are not compatible with T. Thus, the ambiguity, under this ordering, will be carried all the way up until the final solution, the order-5 solutions, are synthesized.

   Tsang states [Tsang, 1993] that the efficiency of his algorithm can be improved by giving the nodes a certain order. Preferred orderings, he suggests, can be obtained by applying some of the general variable ordering techniques discussed below in the "Other Strategies for CSPs" section. Specifically, he suggests using a Minimal Bandwidth Ordering (MBO). This idea is adopted and expanded upon in our work.

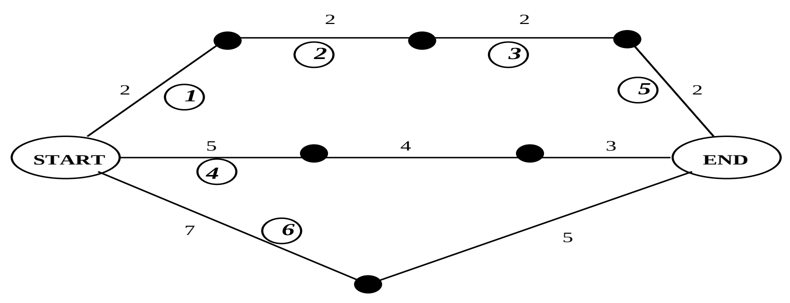

Figure 9: Branch-and-Bound

2.3. Branch-and-Bound

Branch-and-bound (BB) techniques can be used to reduce the amount of search needed to find the optimal solution. BB is based on a common-sense principle: do not keep trying a path that you already know is worse than the best answer.[30] One of the first articles on branch-and-bound was [Lawler and Wood, 1966]. A more readable introduction is in [Winston, 1984]. A simple algorithm describes the method:[31]

1. Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere. Let the optimal-path be an infinite-length path.
2. Until the queue is empty or the first path in the queue has length longer than the optimal-path.
   (a) Remove the first path from the queue.
   (b) Form new paths by extending the removed path one step, if possible.
   (c) If any of the new paths reach the goal, and the shortest of these is shorter than the optimal path, set the optimal path to it. Remove all paths in the queue with length greater than this new optimal-path.
   (d) add all the new paths (except any that reach the goal) with length less than the optimal-path length to the queue, sorting the queue with the smallest length paths in the front.
3. If optimal-path has non-infinite length, return it; otherwise announce failure.

The example given in the introduction will be repeated and explained here. Figure 4 is repeated as Figure 9. Paths would be created for each of the initial arcs from the start. The path with the arc labeled with the circled 1 would be expanded first, since it has the shortest total length. Arc 2 is added to it, giving a total length of 4. Because this is still shorter than any other path, it is extended again by adding arc 3 to it, yielding a total path length of 6. At this point, the path with arc 4, with a length of 5, is shorter than the 1-2-3 path. It is expanded to give a path of length 9.

[30]AKA The "hit-your-head-against-the-wall phenomena"

[31]Mostly from [Winston, 1984].

The first path is again the shortest, so it is extended to give path 1-2-3-5, which reaches the goal in a length of 8. Optimal-path would, at this point, be set to this path. The path including arc 4 would be discarded now, since it already has length greater than the optimal-path. The path with arc 6, however, only has length 7. It is extended, yielding a path of length 12. Because there are no more paths with lengths less than 8, the process is terminated and optimal-path is returned.

This, of course, is a simplified example. Most search problems do not have all possible paths ending up at the goal. Most nodes have multiple branches. Cyclical paths can cause problems. Nevertheless, the BB procedure can be used to find the optimal solution, despite these complications. Various other techniques, with constraint satisfaction being the most relevant here, can be used to further optimize.[32] Heuristics which estimate the distance from a node to the goal state can also be used.[33]

BB is useful for searches where the optimal solution is needed. If **any** valid solution is desired, a depth-first search is indicated. If a solution with the smallest number of arcs is desired, a breadth-first search would be more advantageous. Below, we will demonstrate how BB techniques can be used in combination with solution synthesis and constraint satisfaction to produce an extremely efficient computational semantic problem solver.

[32] As shown below.

[33] For example, a potential path with a high estimated distance to goal can be excluded even though its total current distance is lower than the optimal path.

2.4. Other Strategies for CSPs

CSP techniques can be used in conjunction with many other AI search strategies. The most basic of these strategies is called "lookahead." In fact, lookahead is not an additional strategy, but simply a full, dynamic application of consistency checking. At each decision point in a search, a value is assigned to a variable. This implicitly removes the other possible values of that variable; thus, other variables that depend on those values can be affected. By applying the consistency algorithm,[34] these effects can be automatically propagated at each decision point.

There are various strategies used to handle backtracking. The most advantageous of these, used in conjunction with constraint analysis, is called "dependency-directed backtracking." When the need for backtracking arises, constraint analyses can help indicate what the source of the current conflict is. The search can then be backtracked directly to the source of the problem. This can potentially eliminate large areas of search that will not "fix" the current bottleneck. Because backtracking is eliminated in solution synthesis methods, the various backtracking strategies are not relevant to this research.

Heuristics can be used to great advantage in search. At any branching point, each choice can be analyzed using heuristic knowledge to estimate how much closer to a solution the choice brings the search. Choices with higher heuristic value can be followed first, a technique generally referred to as "best-first" search. Optimal solutions cannot be guaranteed with heuristic search, however, because local optima can obscure longer-term solutions. For instance, turning south seems like a bad choice when your goal is north, unless there is a freeway one block to the south that can speedily take you on your way. Some types of problems require heuristic search to make them tractable. Any problem with an exponentially growing search space that is irreducible with CSP techniques will quickly become unmanageable. Prior to the current research, the Mikrokosmos project relied on heuristic search in its analysis of natural language semantics. It is the thesis of this report, however, that CSP techniques in combination with branch-and-bound and solution synthesis can deliver guaranteed optimal solutions in near-linear time for computational semantic problems. Therefore, heuristic search is not necessary. On the other hand, even with near-linear time speed, large problems, especially those involving discourse, are nowhere near "real-time." Best-first techniques can be added to those described below to improve this situation.

A different kind of heuristic can be used to optimally order the instantiations of variables and their values. These heuristics can be labeled more accurately as "strategies," because they involve general principles rather than world knowledge. These types of strategies are closely connected to constraint analysis; a good discussion of them can be found in [Tsang, 1993]. The first is called "minimal width ordering" (MWO). In general terms, this strategy seeks to instantiate more highly constrained variables first, in hopes that backtracking will be reduced. For example, if variable A constrains variable B to value X and variable C to Y, it would be advantageous to instantiate variable A first. This would reduce the number of choices in B and C, which in turn might restrict choices elsewhere. If variable C was instantiated first, values for it might be tried that conflict with variable A.

Alternatively, a "minimal bandwidth ordering" (MBO) can be used. This ordering seeks to place constrained variables close together so that when backtracking is necessary, only a small distance will have to be backtracked, minimizing the amount of work that might have to be repeated. For

[34] For us, arc consistency will be applied.

example, if variable A constrains B to X, but B is instantiated first, followed by C, D, E and F, then when A is finally instantiated, the search will need to backtrack to B and then recalculate values for C, D, E and F as well. If, on the other hand, A was instantiated directly after B, backtracking would proceed directly to B, with no intervening variables affected.

Of course, a dynamic implementation of arc consistency reduces the importance of these types of orderings. First of all, before search even began, conflicting values will be removed from any variables domain. In addition, during search, when a value for a variable is chosen, all variables affected by the choice can be dynamically processed, with all those effects recursively propagated, etc.. Thus, a dynamic implementation of arc-consistency inherently gives the benefits of MWO and MBO. This notwithstanding, these techniques can still give some advantage. If two or more variable instantiations work together to eliminate certain values of other variables,[35] it would be advantageous to process these "partners" early. Since ordering more constrained variables first maximizes this potential, a minimal width ordering would be helpful.

Solution synthesis, however, alters the picture somewhat. Solution synthesis combines small sub-solutions together to form larger and larger solutions. Interactions outside of the subsets being combined are not noticed. Because of this, it would be helpful to group variables together in such a way as to minimize the amount of ambiguity carried along in the sub-solutions. By grouping variables (and later sub-solutions) together that constrain each other maximally, ambiguity can be eliminated as early as possible. This type of variable grouping is an outgrowth of MBO. This project uses MBO concepts to group variables and synthesized solutions together in order to minimize ambiguity within the sub-solution. This goes well beyond a simple linear ordering of variables on which an SS-TSANG-like algorithm would work.

MWO and MBO help determine which order to instantiate variables. There are also techniques which help decide, given a variable, which values to try first. While variable ordering techniques seek to maximally constrain so that backtracking can be identified early, value ordering techniques seek to eliminate backtracking by trying the most likely values first. For problems in which all possible solutions are required (or the most optimal), these techniques are not helpful. All values that result in solutions must be attempted; it does not matter which order they are found.

[35] Which is not detected by arc consistency until those variables are instantiated.

3. Hunters and Gatherers in Computational Semantics

Implied information, background knowledge, ellipsis, coreference, figurative speech, ambiguity; these are a few of the immense challenges a natural language semantic system faces. And yet, humans process language in real time every day with very little misunderstanding. How can a computer do the same?

By constraining the problem. Sixty six million and some odd amount of thousands is, indeed, a large number. Two hundred and thirty five billion is much larger. These two numbers represent the number of choices an computational semantic system faces for a medium size and a slightly larger size problem. Come across a truly long sentence and the numbers soar past 10^{18}. And that only to determine basic semantic dependencies; add in ellipsis and coreference resolution possibilities and they increase even faster. Such exponential growth in the size of the problem must be constrained if serious work is to be accomplished.

In a "blocks" world, CSP techniques and solution synthesis are powerful mechanisms. Many "real-world" problems, however, have a more complex semantics: constraints are not "yes" or "no" but "maybe" and "sometimes." In computational semantics, certain word-sense combinations might make sense in one context but not in another. We need a method as powerful as CSP for this more complex environment. Our proposal is to 1) use constraint dependency information to partition problems into appropriate sub-problems, 2) combine (gather) results from these sub-problems using a new solution synthesis technique, and 3) prune (hunt) these results using, not constraint satisfaction, but branch-and-bound techniques. These three issues are discussed next. Following this is a description of how a means-end planner for text generation can be implemented using HUNTER-GATHERER and constraint satisfaction techniques. The section concludes with a discussion of the place of natural language semantics in the HUNTER-GATHERER control paradigm. In particular, it will be argued that natural language semantic problems can be naturally broken into relatively independent sub-problems. These are exactly the types of problems that benefit most from HUNTER-GATHERER.

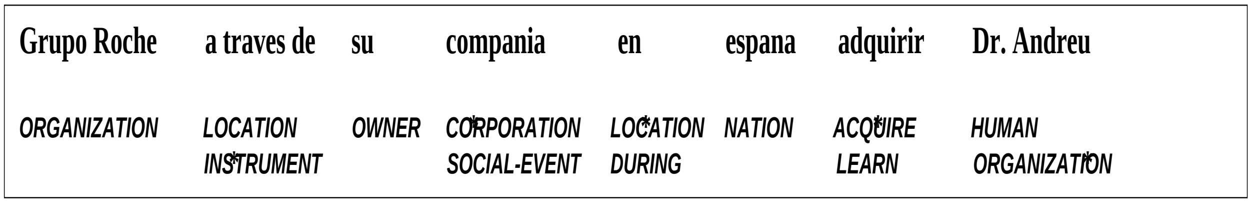

Figure 10: Example Sentence

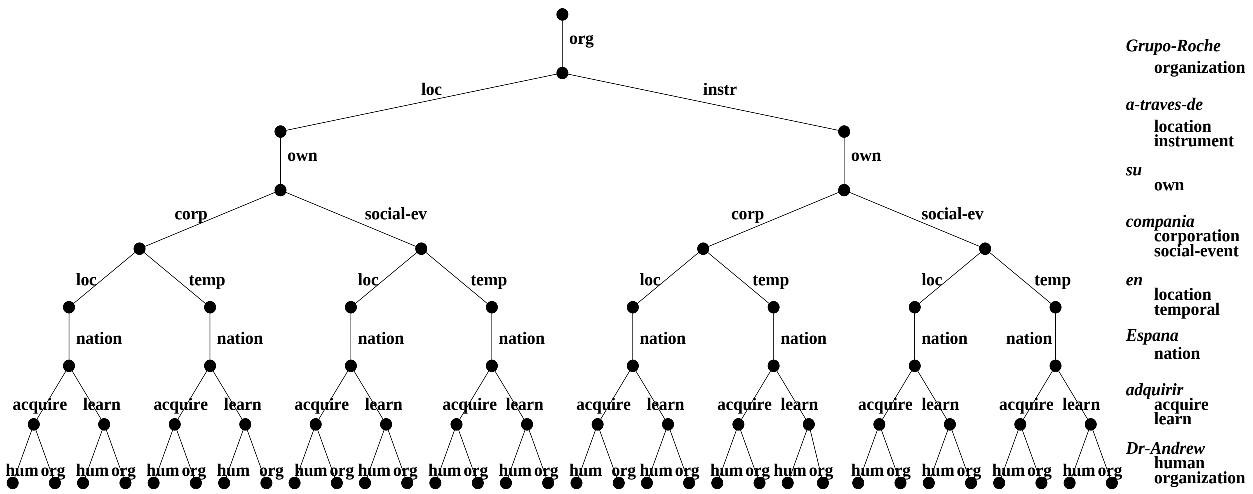

Figure 11: Decision Tree

3.1. Constraint Satisfaction Hunters in Computational Semantics

This section will examine constraint analysis in computational semantic problems. As will be demonstrated in section 3.5, these types of problems typically consist of bundles of tightly constrained sub-problems, each of which combine at higher, relatively constraint-free levels to produce the complete solution. Algorithms that identify these bundles, or "circles," along with their constraint dependencies, will be given here. A simple, "literal," semantic problem will be solved using CSP consistency algorithms. The problem of non-literal meanings will be presented, with the solution to this problem to be presented in the "Using Branch-and-Bound in an Uncertain World" section.

Consider Figure 10, an example of a very simple Spanish sentence analyzed by the Mikrokosmos semantic analyzer. The Spanish sentence along with possible word senses are shown. Figure 11 shows the decision tree for this sentence. The paths leading to leaves at the bottom of the tree represent possible solutions. In this case, there are 32 such solutions.

Figure 12 shows the results when some fairly obvious "literal" semantic constraints are imposed and propagated using arc consistency. Only one valid path remains. Obviously, CSP techniques help out immensely in this literal-interpretation paradigm. Some of the constraints that were used

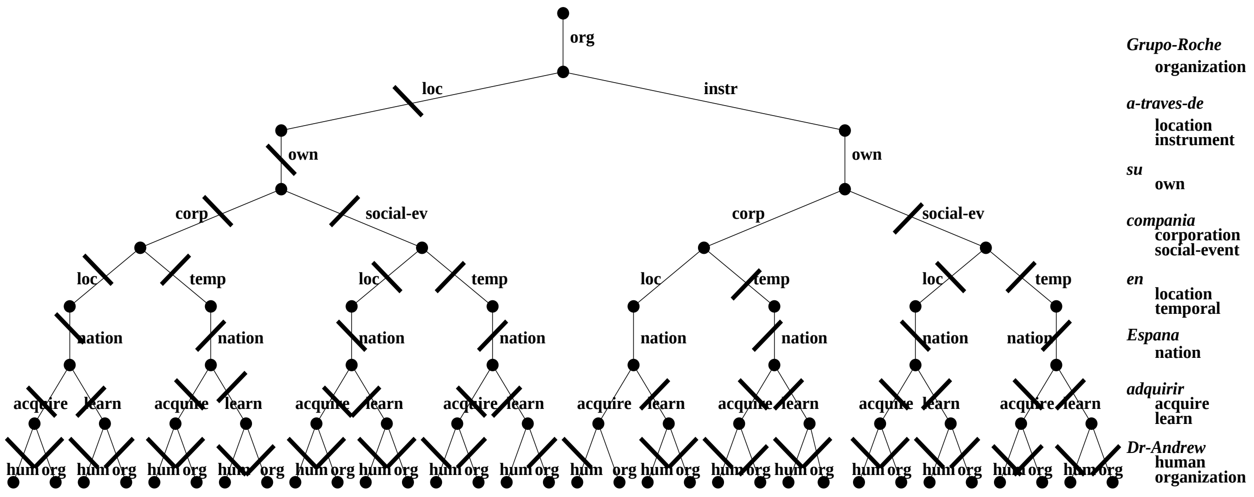

Figure 12: Decision Tree after CSP

to eliminate sub-trees in Figure 11 are as follows:[36]

- the **location** sense for *a traves de* is ruled out because neither of the two senses of *compania* – {**corporation,social-event**} - are descendants of **place** in the ontology.
- *compania* cannot be a **social-event** because the meaning of the prepositional object of *a traves de* in its {**instrument**} sense must be a descendant of **object**, not an **event**.
- *adquirir* is not used in its **learn** sense because **learn** requires an object of type **knowledge**, which *Dr. Andreu* = {**human, organization**} is not.
- *Dr. Andreu* is not **human** because only **organizations** can be **acquired** in our ontology.

Unfortunately, a literal imposition of constraints does not work in computational semantics. For example, *a-traves-de* could very well be **location**, because corporation names are often used metonymically to stand for "the place of the location:"

I walked to IBM.
I walked to where IBM's building is.

Therefore, the fact that *compania* is not literally a **place** only biases the result towards a different interpretation. In fact, under certain contexts, the **location** interpretation might be preferred; certainly it cannot be ruled out completely, as was done in Figure 12. Constraint satisfaction techniques such as arc-consistency, therefore, will be of limited value. The "Using Branch-and-Bound in an Uncertain World" section will describe how to combine branch-and-bound techniques with solution synthesis and knowledge of constraint dependencies. Here, we prepare for that section by describing how constraint dependencies can be identified, and how they typically look in computational semantic problems.

[36]Please bear with us if some of these constraints do not make complete sense; they are for demonstration purposes only.

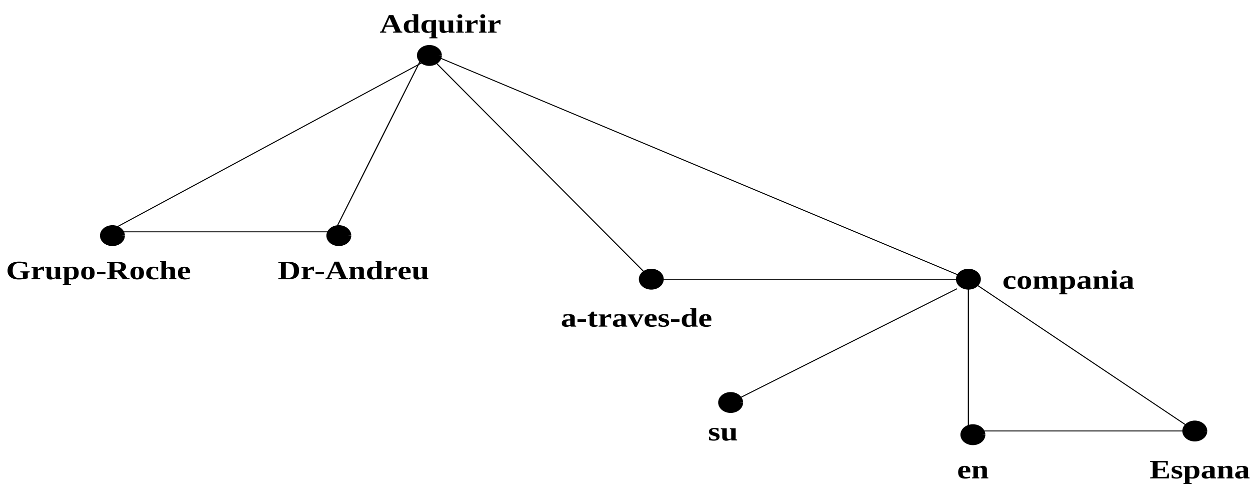

Figure 13: Constraint Dependencies in Sample Sentence

Figure 13 gives a different view of this same problem by graphically displaying the constraint dependencies present in Figure 10. These constraint dependencies can be identified simply by iterating through the list of constraints[37] and connecting with an arc any words involved in the same constraint.

3.1.1. Identifying "Circles" of Dependency

Figure 13 clearly shows three sub-problems, or circles, that are relatively independent. If these circles could be identified, the processing involved in finding a complete solution could be decomposed into three sub-problems. Figure 14 displays the results of such a decomposition. The three circles represent sub-problems with 4, 8 and 4 possible solutions, respectively. Notice that there are still dependencies between the circles. The first two circles are connected because each contains the word *adquirir*; the second two are connected because they both contain the word *compania*. In the next section, we will show how to combine results from circles to form larger and larger solutions, the largest of which will be the solution to the whole problem.

We have taken a multi-phase approach to creating the circles that will guide the solution synthesis mechanism. The approach used is described next. The five phases of circle creation are as follows:

1. Create constraint-based circles. This is accomplished by identifying all cycles in a dependency graph such as Figure 13. As will be shown, the purpose of these constraint-based circles is to eliminate as early as possible in solution synthesis inconsistent variable assignments.

2. Create tree-based circles. Because dependencies are generally limited to structures in a governing relationship (section 3.5), the input tree can be used as a guide to the combination of

[37]The semantic constraints are drawn from the lexicon and ontology. See [Beale, Nirenburg & Mahesh, 1995] for a description of this process.

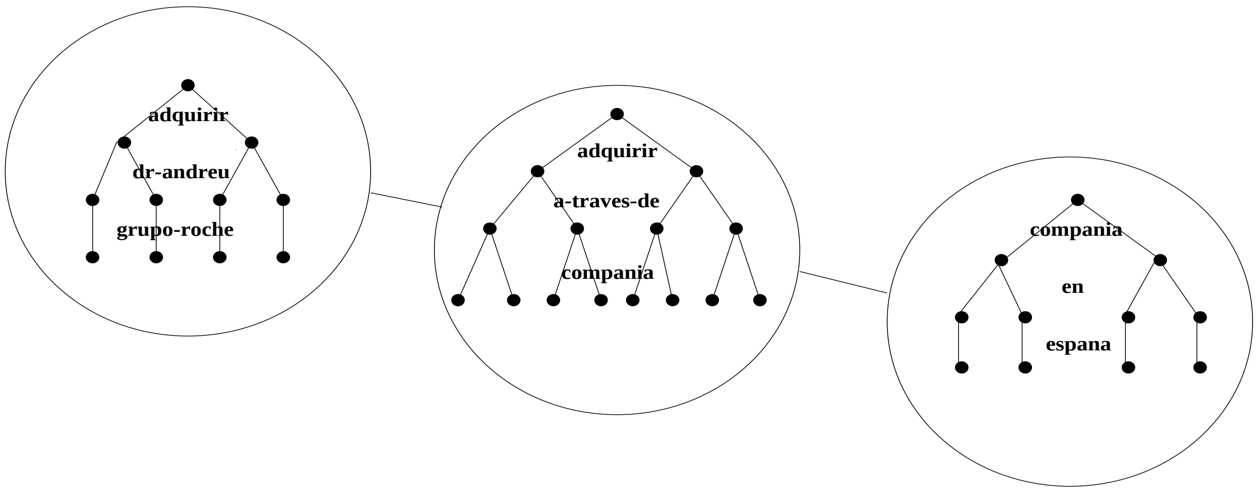

Figure 14: Problem Decomposition in Sample Sentence

smaller structures.[38]

3. Decompose larger circles into a sequence of smaller circles plus individual nodes. Delete any "orphan" circles.

4. Identify which nodes in each circle are constrained outside the circle. These are the nodes for which branch-and-bound techniques can **not** choose optimal values. Eliminate any circle that does not have at least one node not effected outside the circle; it will not contribute toward the efficiency of the solution.[39]

5. Some of the "base" circles can be further decomposed into groups of second-order nodes. The reason for this is intimately tied to the solution synthesis methodology, and thus will be described in the next section.

Step 1, identifying constraint-based circles, is relatively straightforward. Assuming we have a graph such as Figure 13 in the form of an array:[40]

effected[adquirir] = (grupo-roche, dr-andrew, a-traves-de, compania)
effected[grupo-roche] = (adquirir, dr-andrew)
effected[dr-andrew] = (adquirir, grupo-roche)
effected[a-traves-de] = (adquirir, compania)
effected[compania] = (adquirir, a-traves-de, su, en, espana)
effected[su] = (compania)
effected[en] = (compania, espana)
effected[espana] = (compania, en)

[38]Consult section 3.2 for a discussion of why this is beneficial.

[39]See the branch-and-bound section for the reason why this is true.

[40]Recall that such a graph can be created with one pass through the list of all constraints.

Then an algorithm for identifying circles in the graph is as follows:

```
1 PROCEDURE Constraint-Circles
2     Circles <-- nil
3     FOR each Word
4         Circle <-- (Word)
          ;; create all the circles starting with Word and add to Circles
5         Circles <-- Circles + Extend-Path(Circle)
6     Circles <-- remove all identical circles.
7     RETURN Circles

8 PROCEDURE Extend-Path(Circle)
9     Last-Node-Added <-- LAST(Circle)
10    Start-Node <-- FIRST(Circle)
      ;; find all the nodes that can extend the current path
11    Effected-Nodes <-- effected[Last-Node-Added]
12    Circles <-- nil
      ;; if Start-Node is in Effected-Nodes, then Circle is a circle
13    IF MEMBER(Start-Node,Effected-Nodes) THEN
14        Circles = Circles + Circle
      ;; remove any nodes already in Circle
15    New-Nodes <-- Effected-Nodes - Circle
16    FOR each New-Node, Node
17        New-Circle <-- Circle + Node
18        Circles = APPEND(Circles,Extend-Path(New-Circle))
19    RETURN Circles
```

Each time Extend-Path is called by Constraint-Circles, Circle is length 1. In line 15, then, New-Nodes can have length N-1 if the Start-Node affects every other node. If this is the case, Extend-Path will be recursively called N-1 times. Each of these calls can, in turn, produce N-2 calls to Extend-Path, etc.. Thus, Extend-Path can theoretically be called N! times for each of the N Words, for a total complexity of O(N*N!). Actually, this complexity can be reduced by recognizing that any potential circle that contains a Word already processed by the FOR loop in line 3 will already have been identified. New-Nodes in line 15 can therefore also subtract any Words already processed. Thus, by the time the last Word is sent to Extend-Path by Constraint-Circles, no New-Nodes will be present.

This still leaves a potential complexity of O(N!) for Extend-Path when it is called for the first Word. This, however, assumes that every Word can affect every other Word, which fortunately is not the case in computational semantics. On the average, each word constrains less than two other words. In addition, most of the paths created in Extend-Path quickly do one of two things: either they loop back on themselves, creating a circle, or they end at a leaf node which cannot be extended to another Word not already in the current path (or already processed by Constraint-Circles). This is the nature of tree-like graphs. Only experimentation can determine if the time complexity is acceptable. In our experience, the computation of Constraint-Circles is almost instantaneous.

It can be debated whether step 1 is really necessary for computational semantics constraint problems. Given the tree-like nature of its inputs, and the general “governing” principle described in section 3.5, step 2, to be described next, seems to create most, if not all, of the relevant circles, without any question of exponential time difficulties. Identifying constraint-based circles, however, gives the advantage during solution synthesis that all co-dependent nodes are analyzed together as soon as possible, reducing the amount of ambiguity carried along. Because the practical time limits do not seem to be a problem, we have chosen to include step 1; however, further research may prove either that this step is, practically speaking, unnecessary, or that for larger problems (i.e. for whole texts with discourse relations, etc.), the time complexity becomes too great.

Step 2, creating tree-based circles is much easier. It can be accomplished by starting at the top of the input tree, making a circle of it and all the nodes below it, then progressing to each child node, making a circle of it and all the nodes below it, and so on down to the bottom of the tree. Figure 5 in the introduction shows an example of tree-based circles combined with constraint-based circles. .

In Step 3, larger circles are decomposed into smaller circles plus individual nodes. The solution synthesis algorithms will follow these progressions from small circles to large circles. This step is partly accomplished during step 2. During construction of the tree-based circles, a record is kept of which smaller circles the larger circles contain. All that is left is to use any of the constraint-based circles to further decompose any of the tree-based circles. At present, we decompose the smallest tree-shaped circle that has the constraint-based circle as a subset. If no such tree-based circles exist, the constraint-based circle is eliminated.

As will be shown below, the main savings gained from the combination solution synthesis / branch-and-bound methodology comes from synthesizing circles in which there is at least one node[41] that is not constrained outside the circle. Thus, for each circle, these nodes are identified, and, if none exist, the circle is eliminated.

3.1.2. Setting Up Constraint Dependencies

The following section will briefly outline how the dependency information is gathered and recorded. The main goal of this section is to set up the data structures that will be used by the algorithms in the following sections. As shown in the discussion about AC-4 above, it is advantageous to store, for each possible value of each variable, those values in other variables that will satisfy constraints. We also find it advantageous to store, for each value of each variable, values of other variables that conflict with constraints.

Figure 15, a copy of Figure 6B, shows a simple constraint problem. Data structures can be constructed such that the arrays Needed and Fails[42] produce the following results:

$Needed[< A, X >, C] - - > (< B, Y_1 >, < B, Y_2 >, ...)$
$Fails[< A, X >] - - > (< B, Y_1 >, < B, Y_2 >, ...)$

where A is the name of a variable, X is a value from its domain, and $< B, Y_i >$ represent variable-value pairs that will satisfy (for Needed) or conflict with (for Fails) constraint C given the assign-

[41]The node should not already have been "reduced" in a smaller circle. See below.

[42]In previous CSP work, only the Needed information was kept. Fails provides an extra level of automation. The same information is inherent in the Needed array, but must be accessed through a potentially complex arc-consistency application.

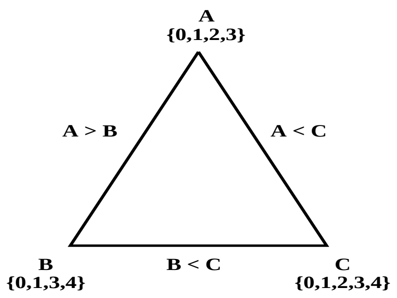

Figure 15: Constraint Example

ment X to A. The constraint involved, C, must be recorded for Needed because for each constraint, the list of Neededs will need to be consulted separately. If any of the constraints has a list of Neededs that are all failed, then the value assignment is invalid. For Fails, the constraint information is not needed, since the $< A, X >$ value assignment will automatically fail all of the B variable assignments listed.

For example,

$Needed[< A, 3 >, [A > B]] - - > (< B, 0 >, < B, 1 >)$
$Failed[< A, 3 >] - - > (< B, 3 >, < B, 4 >, < C, 2 >, < C, 3 >)$

It is also convenient to have a simple array noting which constraints apply to which variables:

$Constraints[A] - - > (C_1, C_2, ...)$

A simple algorithm[43] that performs these tasks would be:

```
1 PROCEDURE SET-UP-CONSTRAINTS
2   FOR each Constraint
3     ;; determine which variables, A and B are involved
4     Constraints[A] $<--$ Constraints[A] + Constraint
5     Constraints[B] $<--$ Constraints[B] + Constraint
6     FOR each value, X, in A's domain
7       FOR each value, Y, in B's domain
8         IF (X,Y) SAT Constraint THEN
9           Needed($< A, X >, Constraint$) $<--$
              Needed($< A, X >, Constraint$)+ $< B, Y >$
10          Needed($< B, Y >, Constraint$) $<--$
              Needed($< B, Y >, Constraint$)+ $< A, X >$
11        ELSE
```

[43]Actually, a slightly more complex algorithm is needed for the types of constraints used by the Text Planner described in a later section. There, one variable will put forward a constraint that some task be accomplished. It is not known which other variable(s) might accomplish the task. Thus, before this algorithm is performed, a pass through the effects of each value instantiation needs to be done so that needs can then be cross-indexed with effects. After this, an algorithm similar to this can be run. The added preprocessing phase has time complexity *O(an)*, where a is the maximum size of the domains.

```
12            $Fails[< A,X >] < - - Fails[< A,X >]+ < B,Y >$
13            $Fails[< B,Y >] < - - Fails[< B,Y >]+ < A,X >$
```

If the maximum size of any domain is a and the number of constraints, or arcs, is e,[44] then this algorithm has time complexity of $O(ea^2)$.

[44]Recall that e is proportional to n^2 for general constraint problems but is proportional to n in computational semantics.

3.2. Solution Synthesis Gatherers in Computational Semantics

As described above, Freuder introduced solution synthesis as a means to "gather up" all solutions for a CSP without resorting to traditional search methods. Freuder's algorithm created a set of two-variable nodes that contained combinations of **every** two variables. These two variable nodes were then combined into three variable nodes, and so on, until a node containing all the variables, i.e. the solution, was synthesized. At each step, constraints were propagated down and then back up the "tree" of synthesized nodes.

Tsang improved greatly on this scheme with the Essex Algorithms. These algorithms assumed that a list of the variables could be made, after which two-variable nodes were created only between adjacent variables in the list. Higher order nodes were then synthesized as usual, starting from the two-variable nodes. Tsang noted that some orderings of the original list would prove more efficient than others, most notably a Minimal Bandwidth Ordering" (MBO), which seeks to minimize the distance between constrained variables.

This work extends the concept of MBO and carries it to a higher level. The whole concept of synthesizing solution sets one order higher than their immediate ancestors is thrown out. Synthesis, here, is aimed at maximally interacting groupings of variables, of any order. Furthermore, this process of using maximally interacting groupings, or circles, extends to the highest levels of synthesizing. Tsang only creates second order nodes from adjacent variables in a list, with the list possibly ordered to maximize second order interactions. After that, third and higher order nodes are blindly created from combinations of second order nodes. In this work, in a sense, MBO is continued on to the higher levels. The circles of co-constrained variables described in the previous section guide the synthesis process from beginning to end.

The main improvement of this approach comes from a recognition that much of the work in SS-FREUDER and SS-TSANG was wasted on finding valid combinations of variables which were not related. For example, in the computational semantic example given above, the words "IBM" (I), "Jacob-Smith" (J) and "ten-million-dollars" (T) (among others) are not directly related by any constraints. Despite this, valid combinations for $N_{IJ}, N_{IT}, N_{JT} and N_{IJT}$ are calculated by SS-FREUDER and, depending on the ordering used, could also be calculated by SS-TSANG. Furthermore, the SS-TSANG algorithm tends to carry along unneeded ambiguity. As shown above, if two related variables are not adjacent in the original list, their disambiguating power will not be applied until they happen to co-occur in a higher-order synthesis. It should be noted that Freuder's algorithm does not have this disadvantage, because **all** combinations of variables are created.[45] The current work combines the efficiency of Tsang's algorithms with the early-disambiguation power of Freuder.

The SS-GATHERER Algorithm

The SS-GATHERER algorithm to be defined below only expends energy on variables directly related by constraints. For instance, a constraint graph of the example computational semantic problem would look like Figure 16. Only two lower order circles, IAJ and AFT would be synthesized:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{AFT} = {(T-O,COST,MON),(OBT,COST,MON)}

[45] But only at great expense.

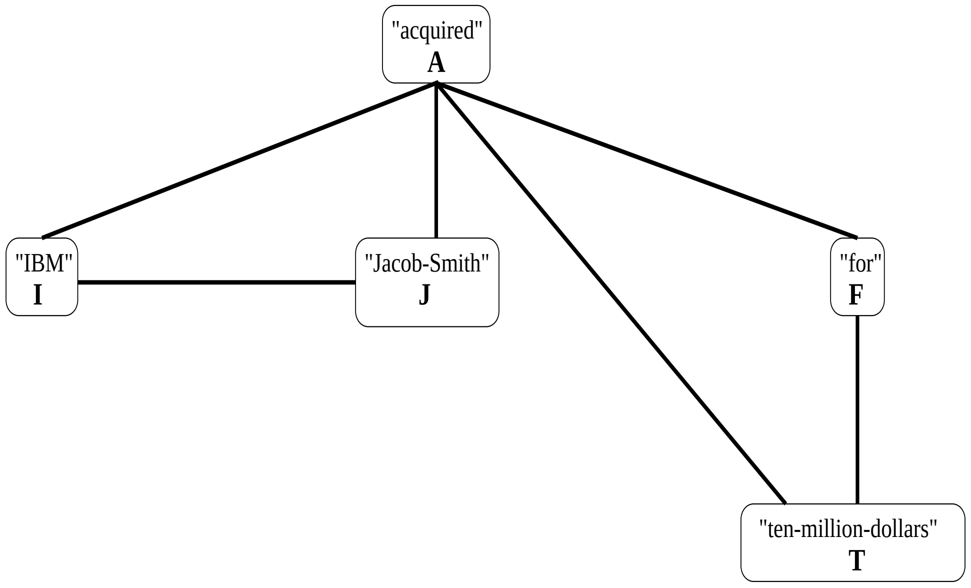

Figure 16: Constraint Graph.

The answer is then directly synthesized from these:

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

Notice that the bulk of disambiguation occurs in the lower order circles which were chosen to maximize this phenomena. Because the same solution was obtained in only three steps, it goes without saying that all of the other combinations of variables used in SS-Freuder (24 extra nodes, not including first order nodes) and SS-TSANG (8 extra nodes) were superfluous. Focusing the synthesizer on circles that will maximally disambiguate produces huge savings while still guaranteeing the correct solution.

One objection that could be raised to this process is that, in spurning the second order nodes, more work is needed to create higher level nodes. For instance, if each variable had three possible values, one needs to test 9 (3^2) combinations for each second-order node, but 27 (3^3) combinations for third order nodes.[46] This argument only is valid, however, if disambiguation is possible at the lower nodes. For example, if two second-order nodes could be created that would form the third-order node, and each second order node could be completely disambiguated to a single solution, then the third order node could be created without any combinatorics, yielding a total of 18 combinations (9 + 9) that were searched. Directly creating the third-order node requires the 27 combinations to be searched. However, if the second order nodes do **not** disambiguate, nothing is gained from them. For this reason, initial[47] circles can be further sub-divided into groups of

[46]It should be pointed out that sometimes second order nodes are used in SS-GATHERER, for example if a variable has only one other variable which constrains it, a tree-based circle with two members will be created. Also, there is nothing sacred about "third" order nodes in SS-GATHERER, although computational semantic constraints seem to produce them the most. It is quite possible that even higher-order nodes could be the starting point.

[47]Initial circles are those that have no sub-circles identified.

second-order nodes, **if** those second-order nodes are connected in the constraint graph.[48]

The algorithm below accepts a list of Circles, created as shown in the previous section (with the above addition). Each circle is in the form:

(List-of-Vars List-of-Sub-Circles)

The list is ordered from smaller circles to larger circles. For the example in the previous section, graphically displayed in Figure 13,[49] the following would be the list of input circles:

```
(
 ((adquirir grupo-roche) nil)
 ((adquirir dr-andrew) nil)
 ((adquirir a-traves-de) nil)
 ((a-traves-de compania) nil)
 ((compania su) nil)
 ((en compania) nil)
 ((en espana) nil)
 ((adquirir grupo-roche dr-andrew)
     ((adquirir grupo-roche) (adquirir dr-andrew)))
 ((adquirir a-traves-de compania)
     ((adquirir a-traves-de) (a-traves-de compania)))
 ((en compania espana)
     ((en compania) (en espana)))
 ((adquirir a-traves-de compania en espana su)
     ((adquirir a-traves-de compania) (en compania espana)))
 ((adquirir grupo-roche dr-andrew a-traves-de compania en espana su)
     ((adquirir grupo-roche dr-andrew)
      (adquirir a-traves-de compania en espana su)))
)
```

An algorithm that implements this methodology follows. In it, a "plan" refers to a specific assignment of values to some number of variables. Section 3.3 will work out a complete example demonstrating SS-GATHERER used in conjunction with branch-and-bound techniques.

```
1 PROCEDURE SS-GATHERER(Circles)
;; This procedure returns value of last PROCESS-CIRCLE.
;; The last circle processed should be the entire problem.
2     FOR each Circle in Circles
3         PROCESS-CIRCLE(Circle)
```

[48]Tree-based circles, for example, can contain some variable pairs that are non-constrained, and fourth-order and higher constraint-based circles can have some variable pairs that are unrelated.

[49]A slight "instructional" fudge in the figure must be pointed out. *Grupo-Roche* and *Dr-Andrew* are not actually connected by a constraint. I connected them so that this simple example would illustrate constraint-based circles. Ditto for *compania / Espana* and *adquirir / compania*. However, the same circles would have been created by the tree-based method. This explains why second-order nodes, as explained immediately above, are not created for these variable pairs immediately below. Be assured that constraint-based circles do occur in more complex sentences.

```
4 PROCEDURE PROCESS-CIRCLE(Circle)
5     Output-Plans < -- nil
6     Name < -- FIRST(Circle)
      ;; The circle "name" is referenced by the (consistently)
      ;; ordered list of variables in it
7     Incoming-Circles < -- SECOND(Circle)
      ;; Incoming-Circles is a list of sub-circles that make up Circle
8     Incoming-Non-Circles < -- REMOVE all variables in Incoming-Circles from Name
9     Non-Circle-Combos < -- Get-Combos(Incoming-Non-Circles)
10    Circle-Combos < -- Combine-Circles(Incoming-Circles)
11    FOR each Non-Circle-Combo in Non-Circle-Combos
12        FOR each Circle-Combo in Circle-Combos
              ;; each incoming circle has consistency info stored in arrays:
13            AC-Info < -- access arc constraint info from input circles
14            Plan < -- add Non-Circle-Combo to Incoming-Plan
              ;; Plan is a potential solution for this Circle
              ;; with a value assigned to each variable.
15            IF Arc-Consistent(Plan,AC-Info) THEN
16                Output-Plans < -- Output-Plans + Plan
17                ;; update AC-Info for this circle
18    RETURN Output-Plans
```

The Get-Combos procedure (line 9) simply produces all combinations of value assignments for the input variables. This procedure has complexity $O(a^x)$, where x is the number of variables in the input Incoming-Non-Circles, and a is the maximum number of values for a variable. In the worst case, x will be n; this is the case when the initial circle contains all the variables and no sub-circles. Of course, this is simply an exhaustive search, not Solution Synthesis. In practice, the circles usually contain no more than two variables not involved in input sub-circles, the exceptions almost always pertaining to the base circle, in which case the combine-circles procedure does not add complexity.

The combine-circles procedure (line 10) combines all consistent[50] plans already calculated for each input Incoming-Circle. In the worst case, where each Incoming-Circle contained a single variable, and Incoming-Circles contained every variable, then the time complexity would be $O(a^n)$,[51] where a is the maximum size of a variable domain. This is unavoidable, and is the nature of CSPs. However, if the number of circles in Incoming-Circles is limited to c, and each circle has at most p possible Plans, then the complexity of this step is $O(p^c)$. This step, without the branch-and-bound modifications presented in the next section, dominates the time complexity of SS-GATHERER. Section 3.3 illustrates how this number can be reduced to a "near" constant value.

The FOR loops in lines 11 and 12 combine the possible Plan-Combos from the Incoming-Non-Circles with the Circle-Combos calculated for the Incoming-Circles. The worst-case time complexity is no worse than the worst-case time complexity for either Combine-Circles or Get-Var-Combos. If Get-Var-Combos produces a^n combinations, then Combine-Circles will produce

[50]If one circle has a Plan1 with the assignment $< A, X >$ and another Circle has a Plan2 with the assignment $< A, Y >$, then Plan1 and Plan2 are not consistent and cannot be combined. See Section 3.3 for a detailed example.

[51]Combining n variables each with a possible values.

none, and vice-versa. In practice, Combine-Circles produces p^c combinations while Plan-Combos produces a constant[52] number of combinations. The total complexity of PROCESS-CIRCLE is therefore *O(p^c)*. Again, this number can be reduced to a "near" constant. The complexity of SS-GATHERER, then, is *O(p^c)* times the number of circles, which is proportional to the number of variables, n. If *O(p^c)* can be shown to be a "near" constant, then SS-GATHERER has time complexity that is "near" linear with respect to the number of variables.

For each synthesis, arc consistency may be performed (line 15). As discussed above, however, un-modified CSP techniques such as arc-consistency are not usually helpful in problems with non-binary constraints. The next section presents a computational substitute that will produce similar efficiency for these kinds of problems. Arc consistency does play a role, however, in the text planning system, PICARD, described in section 3.4. It also comes into play when THRESH is set to a non-zero value, which can create straightforward constraints. Arc-consistency techniques described in section 2.1 are used. These techniques are further discussed in section 4.2 below.

[52] a^x, where x is the number of variables in Incoming-Non-Circles, usually 1 or at most 2, except for base circles.

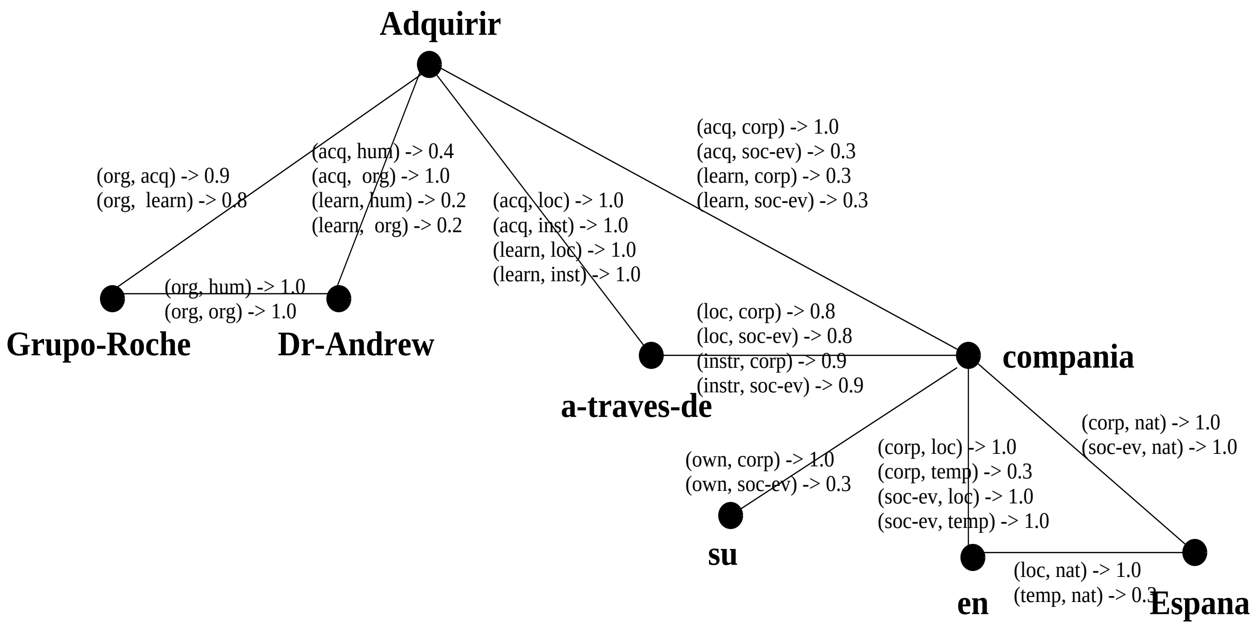

Figure 17: Constraint Dependencies in Sample Sentence

3.3. Using Branch-and-Bound in an Uncertain World

So far, we have modified the solution synthesis algorithms to use maximally interacting circles of variables in order to exploit their disambiguating power as early as possible. Combined with CSP techniques, these algorithms give fast and completely reliable answers to simplified computational semantic problems. The difficulty is that these problems **are** simplified. The constraints used in the example above used yes-no constraints. In computational semantics, however, semantic constraints must be used only as **tendencies**, not sure-and-fast answers. Therefore, for computational semantics, bare CSP techniques are not that helpful.[53] In this section, we will demonstrate that branch-and-bound techniques can be combined with solution synthesis to overcome this problem. We will see that the circles created above to aid constraint satisfaction are also optimal with regard to branch-and-bound techniques.

3.3.1. Implementing Branch-and-Bound

A more complex version of Figure 13 is repeated here as Figure 17. In this figure, constraint "tendencies" are given for each possible combination of value assignments on an arc. Each tendency is rated on a scale of 0 to 1, with 1 being a perfect (literal) semantic match. Some of the values assigned in Figure 17 are explained below:[54]

[53]The next section will show how they **can** be helpful: by using them to restate a means-end planning problem using constraints.

[54]Again, the main point here is not **why** the scores given were assigned. See [Beale, Nirenburg & Mahesh, 1995] for a discussion of constraints and how scores are determined. These scores are only a fairly intuitive assignment of values that will be used to demonstrate the algorithms below.

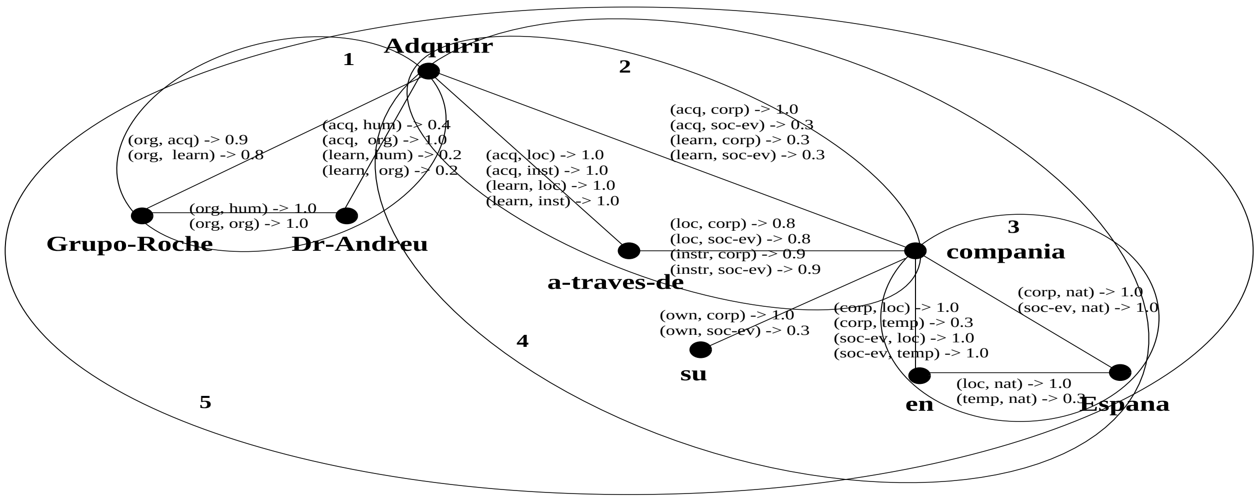

Figure 18: Constraint Circles in Sample Sentence

- Grupo-Roche - Adquirir: Grupo-Roche has only one meaning, organization. As the AGENT of an ACQUIRE event (the (org, acq) arc), it is given a rating of 0.9. This reflects the fact that ORG is not literally a HUMAN (the expected AGENT of an ACQUIRE event), but through metonymy processing, it can be matched. As the AGENT of a LEARN event (the (org, learn) arc), it receives a slightly lower score, reflecting a slightly less promising metonymic relationship in that context.
- Grupo-Roche - Dr-Andrew: These two nouns do not, in this case, constrain each other. They are both given a score of 1.0. In some cases, the grammatical subject and grammatical object of a clause could constrain each other through collocational constraints.
- Adquirir - Dr-Andrew: Each of these words have two meanings, yielding the four possible combinations given. LEARN usually takes some sort of INFORMATION as its THEME; neither HUMAN nor ORG fit this very well, resulting in low scores. ACQUIRE generally takes an OBJECT for its THEME, which ORG is. ACQUIRE generally does not take HUMAN as its THEME, which results in a lower score for the (acq, hum) arc.
- Compania - En: Again, each word can have two meanings. The CORP meaning of Compania can have a LOC specified, but not usually a time (TEMPORAL). On the other hand, SOCIAL-EVENT can easily have either a LOC or a time specified.

The key observation that enables the application of branch-and-bound to solution synthesis problems is that some variables in a synthesis group, or circle, are uneffected by variables outside the circle. For example, in Circle 1 of Figure 18, (Adquirir, Grupo-Roche, Dr-Andrew), both *Grupo-Roche* and *Dr-Andrew* are not connected (by constraints) to any other variables outside the circle. This will allow us to optimize, or reduce, Circle 1 with respect to these two variables. Branch-and-bound techniques are used in this reduction.

Implementing this type of branch-and-bound is quite simple once the apparatus created in previous sections has been produced. Recall that when creating circles, all nodes constrained outside the circle were identified. To implement SS-GATHERER with branch-and-bound, we add this list of constrained variables to the inputs. That is, SS-GATHERER will now accept as input a list of Circles. Each circle is in the form:

(List-of-Vars List-of-Sub-Circles List-of-Constrained-Vars)

Again, the list is ordered from smaller circles to larger circles. For the example sentence, the following would be the list of input circles:[55]

```
(
 ((adquirir grupo-roche dr-andrew)
     NIL
     (adquirir))
 ((adquirir a-traves-de compania)
     NIL
     (adquirir compania))
 ((en compania espana)
     NIL
     (compania))
 ((adquirir a-traves-de compania en espana su)
     ((adquirir a-traves-de compania) (en compania espana))
     (adquirir))
 ((adquirir grupo-roche dr-andrew a-traves-de compania en espana su)
     ((adquirir grupo-roche dr-andrew)
         (adquirir a-traves-de compania en espana su))
     NIL)
)
```

In the last circle, no variables are identified in the List-of-Constrained-Vars. This last circle contains the entire problem and thus has no variables outside of it. The result of processing it will be the optimal solution for all variables. The second to last circle identifies only *adquirir* in the List-of-Constrained-Vars. Its solution set will contain plans for each possible value of *adquirir*, with each plan being optimized with respect to all its other variables.

All that is needed to upgrade SS-GATHERER as presented earlier is the addition of one procedure call (and the procedure, of course). It should be pointed out, however, that SET-UP-CONSTRAINTS must be modified somewhat. Instead of setting up the NEEDED and FAILS arrays based on yes-no constraints, these arrays must recognize constraint scores from 0 to 1. The best approach is to set a threshold below which a constraint score should be considered "not satisfied." That is, line 8 in Set-Up-Constraints should be changed to something like:

IF $C_{XY} >$ THRESH THEN

This will allow the CSP solving mechanism to eliminate combinations with low-scoring constraint

[55]For clarity, we omit the second-order nodes introduced earlier. In fact, using the second order nodes does not improve performance in this case.

scores. All other combinations will be allowed to go forward, and will be reduced by REDUCE-PLANS:

```
1 PR)CEDURE PROCESS-CIRCLE(Circle)
...
18a  REDUCE-PLANS(Output-Plans List-of-Constrained-Vars)
18   RETURN Output-Plans
19 PROCEDURE REDUCE-PLANS(Plans Constrained-Vars)
20     FOR each Plan in Plans
21         Effected-Assignments < -- all possible value
              assignments from Plan that involve a Constrained-Var
22         IF Effected-Assignments is NIL THEN
               ;;This will only happen for the topmost circle
23             Effected-Assignments < -- TOP
24         This-Score < -- Get-Score(Plan)
25         Best-Score < -- Best-Score[Effected-Assignments]
26         IF (This-Score > Best-Score) THEN
27             Best-Score[Effected-Assignments] < -- This-Score
28             Best-Plan[Effected-Assignments] < -- Plan
29     RETURN the list of all Best-Plans stored in Best-Plan array
```

Why does this work? First of all, each previously processed circle has been reduced, so that the input Circle-Combos will only contain reduced plans. In REDUCE-PLANS, then, we want to retain all combinations of value assignments for variables that are effected outside the circle. Line 21 calculates what these effected combinations are for the input plan. The Best-Score and Best-Plan arrays are then indexed by this (consistently ordered) list of value assignment combinations. The goal is that, for each possible combination of assignments of variables effected outside the circle, find the Plan that maximizes that combination. Because all of the other, Non-Constrained-Vars, are **not** effected outside the circle, we can find the Plan that maximizes each of the combinations that **are** effected outside the circle.

This is how SS-GATHERER works for the example sentence. The first Circle input to PROCESS-CIRCLE is

```
((adquirir grupo-roche dr-andrew)
     NIL
     (adquirir))
```

That is, the Circle Name is (adquirir grupo-roche dr-andrew), there are no Incoming-Circles, and the List-of-Non-Constrained-Vars is (adquirir). There will be no Circle-Combos, but Non-Circle-Combos will be produced for each combination of value assignments possible:[56]

```
((<A,acq>,<G,org>,<D,hum>)
```

[56]In the following, A = adquirir, G = grupo-roche, D - dr-andrew, A-T = a-traves-de, C = compania, S = su, E = en and ES = espana.

```
 (<A,acq>,<G,org>,<D,org>)
 (<A,learn>,<G,org>,<D,hum>)
 (<A,learn>,<G,org>,<D,org>))
```

Assuming that the THRESH in SET-UP-CONSTRAINTS is 0 so that all possible combinations will be passed through to REDUCE-PLAN, this entire list of plans will be sent to REDUCE-PLAN. Each plan will be scored[57] as follows (consult Figure 18 to see where constraint scores come from):

```
 (<A,acq>,<G,org>,<D,hum>)   [0.9 * 0.4 * 1.0 = 0.36]
 (<A,acq>,<G,org>,<D,org>)   [0.9 * 1.0 * 1.0 = 0.9]
 (<A,learn>,<G,org>,<D,hum>) [0.8 * 0.2 * 1.0 = 0.16]
 (<A,learn>,<G,org>,<D,org>) [0.8 * 0.2 * 1.0 = 0.16]
```

Because *adquirir* is the only Constrained-Vars, only combinations of value assignments involving it will be in Effected-Assignments. Only two such possible assignments exist: $< A, acq >$ *and* $< A, learn >$. The plan $(< A, acq >, < G, org >, < D, org >)$ maximizes $< A, acq >$ with a score of 0.9. The two possible plans for $< A, learn >$ have identical scores. Both could be kept; the algorithm above simply chooses the first. Therefore, the list of Output-Plans for the first circle is:

```
((<A,acq>,<G,org>,<D,org>)
 (<A,learn>,<G,org>,<D,hum>))
```

The other plans are discarded. It must be stressed here that discarding the other plans in no way incurs risk of finding sub-optimal solutions. The only variable that could be effected further outside this circle is *adquirir*, so plans that maximize each of its possible assignments were chosen. Nothing can happen later on to cause, for instance, the $(< A, acq >, < G, org >, < D, hum >)$ plan to become better than the $(< A, acq >, < G, org >, < D, org >)$ plan, since all interactions involving G and D have been accounted for.

The other circles proceed along the same lines. The (adquirir a-traves-de compania) circle produces the following list of Output-Plans (in this case, *adquirir* and *compania* are both effected outside the circle, so we need to find optimal plans for all combinations of their possible values):

```
((<A,acq>,<A-T,instr>,<C,corp>)
 (<A,acq>,<A-T,instr>,<C,soc-ev>)
 (<A,learn>,<A-T,instr>,<C,corp>)
 (<A,learn>,<A-T,instr>,<C,soc-ev>))
```

The (en compania espana) circle produces:

```
((<E,loc>,<C,corp>,ES,nat>)
 (<E,loc>,<C,soc-ev>,ES,nat>))
```

[57]Combining scores for a list of constraints is complicated. To simplify, we assume here that all individual constraint scores are multiplied together.

It becomes more interesting when smaller circles are synthesized into larger ones. For the following circle:

```
((adquirir a-traves-de compania en espana su)
 ((adquirir a-traves-de compania) (en compania espana))
 (adquirir))
```

there are two input Circles, (adquirir a-traves-de compania) and (en compania espana), and one input Non-Circle, su. Line 9 of SS-GATHERER calculates the Non-Circle-Combos, which amounts to a single plan because *su* has only one word sense:

$((< S, own >))$

Line 10, where Combine-Circles is called, synthesizes compatible plans for the smaller circles into larger circles. Recall that the Output-Plans for the two circles were:

```
((<A,acq>,<A-T,instr>,<C,corp>)    and ((<E,loc>,<C,corp>,<ES,nat>)
 (<A,acq>,<A-T,instr>,<C,soc-ev>)        (<E,loc>,<C,soc-ev>,<ES,nat>))
 (<A,learn>,<A-T,instr>,<C,corp>)
 (<A,learn>,<A-T,instr>,<C,soc-ev>))
```

"Compatible" plans are then synthesized. "Compatible" plans include all those for which like-variables have the same assignment. For instance, $(< A, acq >, < A-T, instr >, < C, corp >)$ and $(< E, loc >, < C, soc-ev >, < ES, nat >)$ are **not** compatible because a different value is assigned for C. The result, Circle-Combos, is showed below:

```
((<A,acq>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>)
 (<A,acq>,<A-T,instr>,<C,soc-ev>,<E,loc>,<ES,nat>)
 (<A,learn>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>)
 (<A,learn>,<A-T,instr>,<C,soc-ev>,<E,loc>,<ES,nat>))
```

Each of these Circle-Combos is then combined with the single Non-Circle-Combo in the FOR loops in lines 11 and 12. This will, in effect, add the assignment $< S, own >$ onto each of the Circle-Combos:

```
((<A,acq>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>,<S,own>)
 (<A,acq>,<A-T,instr>,<C,soc-ev>,<E,loc>,<ES,nat>,<S,own>)
 (<A,learn>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>,<S,own>)
 (<A,learn>,<A-T,instr>,<C,soc-ev>,<E,loc>,<ES,nat>,<S,own>))
```

Arc-Consistency is then checked in line 15, but in our case, since THRESH was set to 0, it will have no effect and all possible combinations will be sent through to REDUCE-PLANS.

The input circle $(< A, acq >, < A-T, instr >, < C, corp >)$ only had A-T as a Non-Constraind-Var, while $(< E, loc >, < C, soc-ev >, < ES, nat >)$ had both E and ES as Non-Constrained-Vars.

In the plans input to REDUCE-PLANS for this synthesis, those NON-Constrained-Variables are already reduced to optimal values for the plans they are in. For this synthesis, only A (adquirir) is identified in the input Circle description as a Constrained-Var. Therefore, for the plan $(< A, acq >$ $, < A - T, instr >, < C, corp >, < E, loc >, < ES, nat >, < S, own >)$, $(< A, acq >)$ will all be identified as Effected-Assignments in line 21 of REDUCE-PLANS. The Get-Score procedure is then called for the plan, with the result stored in an array indexed by this Effected-Assignments.

The Get-Score procedure in line 24 calculates the combined score for each of the constraints in the circle. In practice, a list of scores for input Circles should be maintained so that each test is not repeated for each Plan. Only the constraints involving Non-Circle variables and constraints **between** input circles should have to be calculated. In this case, all constraints involving S (su) need to be calculated since it was not involved in any input circles. If there were any cross-constraints between circles, such as between A (adquirir) and E (en), those constraints would need to be added. In summary, PROCESS-CIRCLE, and thus REDUCE-PLANS, should only need to calculate constraint scores for constraint interactions new to the circle.

The score for the first plan is calculated as:

```
(<A,acq>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>,<S,own>)
  = (0.9 * 1.0 * 1.0) = 0.9
```

where 0.9 is the score for the $(< A, acq >, < A - T, instr >, < C, corp >)$ circle (calculated above), the first 1.0 is the score for the $(< C, corp >, < E, loc >, < ES, nat >)$ circle, and the second 1.0 is the score of the constraint added by S (su). This score is stored in the Best-Score as the high score (since it is the first) indexed on the Effected-Assignments $(< A, acq >)$.

The second input plan to REDUCE-PLANS is $(< A, acq >, < A - T, instr >, < C, soc - ev >$ $, < E, loc >, < ES, nat >, < S, own >)$. This also has an Effected-Assignments = $(< A, acq >)$.

The score for this plan is calculated as

```
(<A,acq>,<A-T,instr>,<C,soc-ev>,<E,loc>,<ES,nat>,<S,own>)
  = (0.27 * 1.0 * 1.0) = 0.27
```

where the score 0.27 is the score of the $(< A, acq >, < A - T, instr >, < C, soc - ev >)$ circle, the first 1.0 is the score of the $(< C, soc - ev >, < E, loc >, < ES, nat >)$ circle, and, again, the second 1.0 is the score of the constraint added by S (su). Because this plan has the same Effected-Assignments and it has a lower score than the previous plan, it is discarded.

The third input plan, $(< A, learn >, < A - T, instr >, < C, corp >, < E, loc >, < ES, nat >$ $, < S, own >)$, has Effected-Assignments = $(< A, learn >)$. A total score of 0.27 is calculated and stored in the Best-Score array indexed on this Effected-Assignments.

The last input plan, $(< A, learn >, < A - T, instr >, < C, soc - ev >, < E, loc >, < ES, nat >$ $, < S, own >)$, has the same Effected-Assignments, and receives a total score of 0.27. It is, therefore, discarded, since it has a lower score than the previous plan with the same Effected-Assignments.

The output of REDUCE-PLANS, and therefore the output of PROCESS-CIRCLE for this circle, is:

```
((<A,acq>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>,<S,own>)
 (<A,learn>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>,<S,own>))
```

The only variable that was not explicitly maximized in these plans is *adquirir*. This makes sense because *adquirir* is the only variable that interacts outside the circle.

The final circle, the solution to the whole problem, is then synthesized from Circle 1 and Circle 4, the circle just created. PROCESS-CIRCLE, in this case, combines all the compatible plans from Circle 1 and Circle 4, then, because the arc consistency will do nothing with THRESH set to 0, sends them on to REDUCE-CIRCLE. Because all of the variables except *adquirir* were optimized for the plans they were in, only *adquirir* must be reduced here. This produces a single optimal plan:

```
(<A,acq>,<G,org>,<D,org>,<A-T,instr>,<C,corp>,<E,loc>,<ES,nat>,<S,own>)
```

In this example, THRESH was set to 0. In effect, this by-bassed the arc consistency checking process, unless there were some specific yes-no constraints present. The whole process can be further optimized by picking a reasonable value for THRESH such as 0.5. If no solutions can be found using that THRESH, it could be lowered further.

3.3.2. Branch-and-Bound Results

To illustrate how branch-and-bound dramatically reduces the search space, consider the results of applying it to the sample sentence.

Circle	Input-Circles	Input-Combos	Reduced-Combos
1	none	2*2*1 = 4	2
2	none	2*2*2 = 8	4
3	none	2*2*1 = 4	2
4	2 and 3	synth only	2
5	1 and 4	synth only	1

The total number of combinations examined is the sum of the input combos; in this case 4+8+4=16. Compare this to an exhaustive search, which would examine (2*1*2*2*2*1*2*1) = 32 combinations. As the input problem size increases, the savings are even more dramatic. This happens because the problem is broken up into manageable sub-parts; the total complexity of the problem is the **addition** of the individual complexities. Without these techniques, the overall complexity is the **product** of the individual complexities.

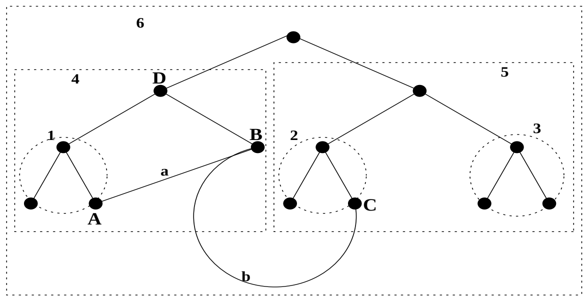

Figure 19: Cross Dependencies

The only way multiplicative growth can occur is when there are constraints across trees, as in Figure 19. In that Figure, several of the circles cannot be fully reduced due to interactions outside the circle. Variable A in Circle 1 cannot be fully reduced[58] because of Arc a. Note, however, that when Circle 4 is synthesized, Variable A **can** be reduced because, at that point, it does not interact outside the larger circle. In Circle 4, Variable B cannot be reduced because it interacts with Variable C. Likewise, Variable C cannot be reduced in Circle 2 because of its interaction with Variable B. In all of these cases, ambiguity must be carried along until no interactions outside the circle exist. For Variables B and C, that does not occur until Circle 6, the entire problem, is processed.

As is argued below, in computational semantic problems interactions such as Arc a and Arc b generally do not occur.[59] "Governed" interactions such as Variable D directly constraining Variable A can occasionally occur, but these only delay reduction to the next higher circle. Thus, some local multiplicative effects can occur, but over the whole problem, the complexity is additive.

To illustrate this point, consider what happens as the size of the problem increases. The following table shows actual results of analyses of various sized problems.

# plans	79	95	119
exh. combos	7,864,320	56,687,040	235,092,492,288
SS-GATHERER	179	254	327

It is interesting to note that a 20% increase in the number of total plans[60] (79 to 95) results in a 626% increase (7.8M to 56M) in the number of exhaustive combinations possible, but only a 42% increase (179 to 254) in the number of combinations considered by SS-GATHERER. As one moves on to even more complex problems, a 25% increase (95 to 119) in the number of plans catapults the exhaustive complexity 414,600% (56M to 235B) and yet only increases the SS-GATHERER complexity 29% (254 to 327). As the problem size increases, the minor effects of "local multiplicative" influences diminishes with respect to the size of the problem. We expect,

[58]By "fully reduced" we mean all child variables maximized with respect to a single parent, which cannot be reduced because it connects higher up in the tree.

[59]"Long distance" dependencies do exist, but are relatively rare.

[60]The total number of plans corresponds to the total number of words senses for all the words in the sentence.

therefore, the behavior of this algorithm to move even closer to linear with larger problems (i.e. discourse). And, again, it is important to note that SS-GATHERER is guaranteed to produce the same results as an exhaustive search.

Although time measurements are often misleading, it is important to state the practical outcome of this type of control advancement. Prior to implementing SS-GATHERER, all computational attempts to process larger sentences failed. The largest sentence above was analyzed for more than a **day** with no results. This is the nature of exponential search space. Using SS-GATHERER, on the other hand, the same problem was finished in 17 **seconds**. It must be pointed out as well that this is not an artificially inflated example. It is a real sentence occurring in natural text; and not an overly large sentence at that. Techniques such as SS-GATHERER must be employed to process real-life problems. Knowledge-based semantics has been severely limited until now, subject to arguments that it only works in "toy" environments. SS-GATHERER will enable large-scale investigations in the knowledge-based paradigm.

3.4. Constraint Satisfaction in Natural Language Generation: The PICARD Text Planning System

Many text planning systems are being used quite successfully today. Their success, however, has come about as a result of several compromises. Constraining the domain and text types are the most obvious. Related to this, however, are several control issues that have been hidden by the simplified nature of previous systems but are now becoming important as those simplifications are lifted:

> " Most current discourse planners ... rely on customized planning algorithms with procedural semantics for the purpose of solving specific text-planning problems. ... careful analysis of these programs show that there is nothing in their semantics to prevent them from generating incorrect plans, generating plans with redundant steps, or failing to find plans in situations where they exist. To the extent that these planners have been able to avoid these problems, they have done so by severely limiting the expressive power of action descriptions and/or requiring the designer of action descriptions to handcraft each description to fit correctly into the ad hoc semantics of the specific plan for which the action is intended." (Young & Moore, 1994)

> "With simple state-based representations, complete search strategies will generally be exponential as a function of solution length. With more expressive representations ... determining if solutions to arbitrary problems exist is an undecidable problem. Such disconcerting results have led several researchers to abandon the use of explicit or declarative problem representations. However, it appears that doing so requires that the goals of the agent be within a narrow range that are hard-coded into the problem representation." (Tenenberg, 1991).

> "Time to impact?" (Captain J.L. Picard)

The last quote above graphically illustrates what the first two quotes are talking about. When Capt. Picard asks how long his spacecraft has until it is obliterated by alien fire, he needs to know **NOW**. Furthermore, he needs to know **CORRECTLY**. Unfortunately, the current generation of text planners are not able to process real-life problems quickly, nor are they guaranteed to process them correctly.

Tenenberg states the obvious problem for all AI applications: basic search strategies have exponential time complexity. Young and Moore point out that most text planning systems currently are neither **sound** (guaranteed to give correct answers) nor **complete** (guaranteed to find correct answers). Both citations agree that current approaches sidestep these problems by abandoning declarative knowledge in favor of ad hoc procedures. Young and Moore go on to argue that the proliferation of procedural knowledge leads to unsoundness and incompleteness, and both papers conclude that such an approach can only be successful on a narrow range of limited problems.

Young and Moore, in their paper, go on to introduce the DPOCL text planning system. The main goal of that research was to ensure soundness and completeness. In this they no doubt succeeded; however, no claims were made concerning the efficiency of their work. Tenenberg, on the other hand, addressed efficiency in his discussion of abstraction in planning. We agree with Young and Moore's conclusion that ad hoc procedures contribute to unsoundness and incompleteness. Declarative knowledge which clearly marks preconditions and effects must be used, along with a control mechanism that ensures soundness and completeness. PICARD can be viewed as an attempt to add efficiency to this type of control paradigm by applying techniques similar to Tenenberg's abstraction. The work builds on the HUNTER-GATHERER analysis system described above.

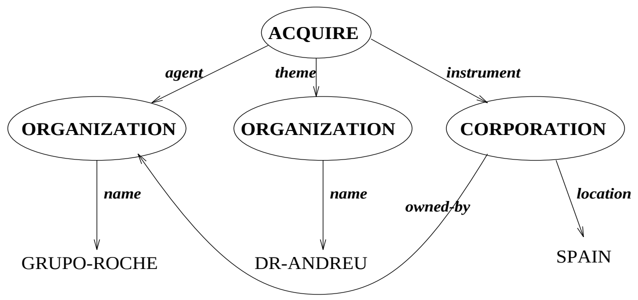

Figure 20: Example Semantic Representation

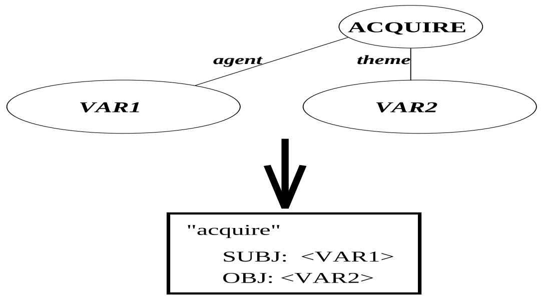

Figure 21: Lexicon Entry for *acquire*

That system employs constraint satisfaction, branch-and-bound and solution synthesis techniques to produce near linear-time processing for knowledge-based semantic analysis in the Mikrokosmos Machine Translation Project. PICARD enables similar results for text planning by recasting localized means-end planning instances into abstractions[61] connected by usage constraints that allow HUNTER-GATHERER to process the global problem as a simple constraint satisfaction problem.

3.4.1. Text Planning for Machine Translation

Figure 20 is a representation of the semantic content of a simple natural language sentence. In English the sentence could be rendered "Grupo Roche acquired Dr. Andreu through a subsidiary in Spain." The node names are semantic concepts taken from a language-independent ontology. Arc labels correspond to relations between concepts. The ontology defines for each concept the set of arcs that are allowed/expected, as well as the appropriate filler concepts. For simplicity, additional semantic information such as temporal relationships are not shown. Please consult (Beale, Nirenburg & Mahesh, 1995; Onyshkevych & Nirenburg, 1994 and Mahesh & Nirenburg, 1995) for more information about semantic representation in the Mikrokosmos system. For our purposes, the details of the semantic representation and generation lexicon entries to follow are unimportant; they serve only as simple examples of control concepts that will apply to more complex problems.

Generation lexicon entries attempt to match parts of the input semantic structures and map

[61]Or macros, or circles, or sub-groups, depending on your background.

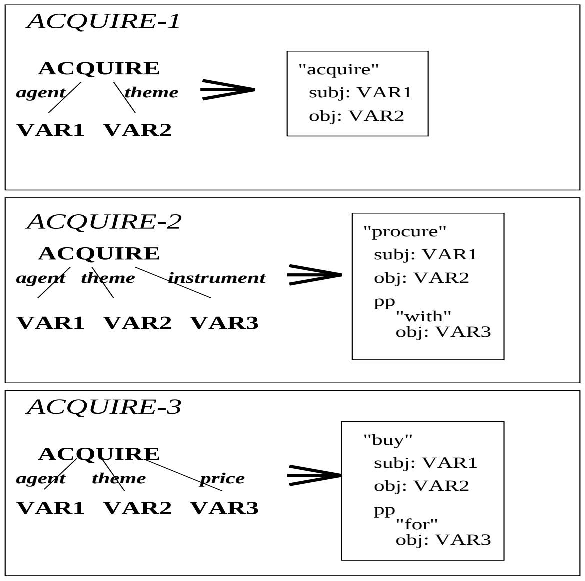

Figure 22: Three entries for ACQUIRE

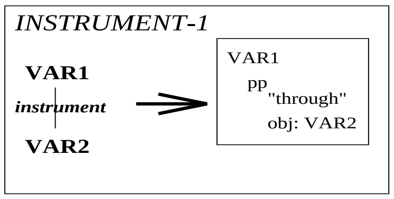

Figure 23: An entry for INSTRUMENT

them into target language surface structures. For instance, a lexicon entry for the concept **ACQUIRE** might look like Figure 21. The VARs in the entry will be bound to the corresponding semantic structures in the input, and their target realization will be planned separately and inserted into the output structure as shown. Typically, lexicon entries also contain semantic and pragmatic constraints. For instance, VAR1 might be constrained to be HUMAN. The entry could also be constrained to apply only to texts with certain stylistic characteristics. Collocational constraints are also important in generation. Any of these constraints can apply locally or can be propagated down to the VARs. The interplay of constraints is a major factor in determining the best overall plan.

Planning for Machine Translation comes in when we try to combine information in various lexicon entries to best match the input semantics with as little redundancy as possible and maximal adherence to the constraints. Figures 22, 23 and 24 represent some possible lexicon entries that might be used in planning target English sentences for the structure in Figure 20.

A typical means-end, hierarchical planner[62] uses the following algorithm:

[62]A similar algorithm can be used for non-hierarchical inputs. PICARD does not require hierarchical semantic

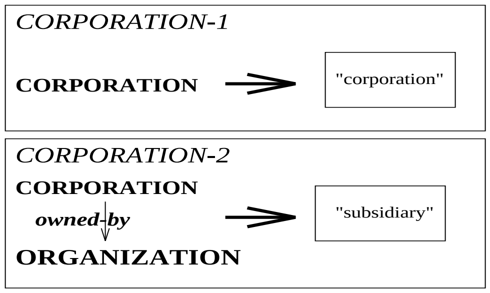

Figure 24: Two entries for CORPORATION

```
PROC PLAN(SEMANTIC-CONTENT)
1  Pick one PLAN that implements base
   meaning in SEMANTIC-CONTENT
2  FOR each PRECONDITION in PLAN not
   already satisfied
3      PLAN(PRECONDITION)
4  FOR each unrealized VAR in PLAN
5      PLAN(VAR)
6  FOR each unplanned RELATION in
       SEMANTIC-CONTENT
7      PLAN(RELATION)
8  IF FAILURE THEN BACKTRACK
```

For example, to generate text for Figure 20, the **ACQUIRE** concept would be passed to PLAN. Three possible entries exist for **ACQUIRE** (Figure 22). The first, *ACQUIRE-1*, expects a semantic environment with *agent* and *theme* relations, both of which are in the input semantics. There are no preconditions in these simplified entries, so the procedure skips to line 4. There are two VARs that are not realized in the first entry. These VARs are bound to the input semantics, VAR1 to a **GRUPO-ROCHE** instance of an **ORGANIZATION** concept, VAR2 to a **DR-ANDREU** instance of **ORGANIZATION**, and PLAN is called recursively for each, leading eventually to surface strings like *Grupo Roche* and *Dr. Andreu.* There is still an unplanned relation of the input **ACQUIRE** node, namely the *instrument* link to a **CORPORATION** concept. This relation is therefore recursively planned in line 7. "FAILURE" in line 8 can refer to a number of possible outcomes, such as over-generation of semantic content or an inability to plan one of the VARs or RELATIONs. This type of planner can be made to find all possible solutions by storing successful overall plans and then backtracking.

Depending on which lexicon entries are used, plans will be more or less complex. The second entry that realizes an **ACQUIRE** concept, *ACQUIRE-2*, has a "built-in" *instrument* relation. Because of this, there will be no unplanned RELATIONs in line 6. Similarly, the second **CORPORATION** entry encorporates the *ownership* relation. The first entries for both **ACQUIRE** and **CORPORATION** need to specifically plan for those links. *ACQUIRE-3* is an example of over-generation. It expects a *price* relation in the input semantics. This can either be made to

inputs.

cause FAILURE (line 8) or simply penalize any plans that utilize it.

The optimal overall plan can be determined by scoring all of the constraints present in each of the sub-plans, adding in penalties for over-generation, and adding in rewards for shorter plans. The best English sentence using the entries given on the example semantic input would combine *ACQUIRE-2* and *CORPORATION-2* to give something like "Grupo Roche procured Dr. Andreu with a subsidiary in Spain." [63] This would be better than using *ACQUIRE-1*, which requires the extra *INSTRUMENT-1* plan and *CORPORATION-1*, which requires an extra plan for the *ownership* link. *ACQUIRE-3* contains unwanted semantics (a *price* relation) and would thus be penalized.

There are two problems with PLAN. First, it cannot be guaranteed to be sound. Preconditions satisfied at a higher level of processing can be undone by side effects at lower levels. This is the problem that Young and Moore tackled with DPOCL. Second, PLAN is horribly inefficient. Local solutions are planned again and again as backtracking moves up and down the input semantic tree. Preconditions and constraints must be continually rechecked because each combination of sub-plans may be different. This is the problem tackled by Tenenberg with abstractions. PICARD identifies local areas of dependency and plans them separately. It uses constraint satisfaction techniques to ensure soundness. It recasts the means-end planning paradigm into an abstract system of independent sub-plans connected by usage constraints, so that efficient solution synthesis procedures can combine them. It is this last concept that is explored in the remainder of this paper.

It must be noted that text planning for machine translation is somewhat easier than for many Natural Language text planning problems, primarily because the semantic content is given. The main goals of an MT text planner are lexical choice and word and sentence ordering. In general, communicative goals are inherent in the input semantic content, although pragmatic features must be taken into account to a greater or lesser degree. In addition, appropriate generation of discourse structure, figurative language, anaphora and ellipsis serve to complicate matters. Text planners for question-answering systems have the added complexity of starting from communicative goals. This makes for more complex planning; nevertheless, the PICARD principles to be explained below can be applied in exactly the same manner.

3.4.2. Using Constraint Satisfaction to Enable Abstractions

It would be useful if we could divide text planning problems into relatively independent sub-problems and use HUNTER-GATHERER's solution synthesis to efficiently combine the smaller solutions. The problem is that solution synthesis requires an unchanging, orderly set of variables to start with. In Figure 3, there are 4 variables, A, B, C and D. Each one of these variables has a set of possible solutions. Three second order nodes are created, AB, BC and CD. From these, the ABC and BCD third order nodes are created and, finally, the answer, ABCD, is synthesized.

In text planning, as in all types of means-end planning systems, there is no fixed number of variables. "Variable," in this context, refers to a set of possible plans from which one **must** be chosen. A variable can be set up for **ACQUIRE**, which has three possible plans. One of them must be chosen. On the other hand, sometimes a plan for *instrument* is needed, and sometimes not. For instance, if *ACQUIRE-1* (Figure 22) is used, a separate sub-plan must be made for the *instrument*

[63]We emphasize that this is the best **computationally**, given the lexicon and semantic inputs in this simplified example.

relation. Two "variables" would be needed, one for **ACQUIRE** and one for *instrument.* If the *ACQUIRE-2* is used, the *instrument* plan and variable are unnecessary. Lexicon entries which have different set of VARS, different preconditions and/or contain more or fewer relations all create differing amounts of sub-plans. These differences are compounded as different paths through the space of possible plans are taken.

PICARD solves this problem in a simple way. Means-end planning is carried out **locally** to determine, for each lexicon entry, the additional sub-plans that are needed. Again, these sub-plans correspond to VARs and missing relations or preconditions in the lexicon entry. For instance, the *ACQUIRE-1* entry requires a sub-plan for the missing *instrument* relation. For each needed sub-plan, a "usage constraint" is added to the lexicon entry that will "request" some "non-dummy"[64] sub-plan to be used that fulfills the need. The *ACQUIRE-1* entry, for example, would receive a usage constraint that requires it to use one of the sub-plans for *instrument.* In addition, for each of the sub-plans that can fill the need, a usage constraint is added such that those entries can only be used if "requested" by some other plan.

For each semantic concept and relation that **is** included in the lexicon entry, a dummy sub-plan is created. For instance, in the *ACQUIRE-2* entry, a dummy *instrument* sub-plan is created and added to the list of other *instrument* sub-plans. The *ACQUIRE-2* entry then receives a usage constraint that "requests" the use of the dummy sub-plan. The dummy sub-plan receives a usage constraint that it be used only if "requested." The fact that ACQUIRE-2 does not "request" one of the non-dummy *instrument* plans will prevent them from being used.

The main benefit this gives is that a stable set of "variables" can be created. There will be an **ACQUIRE** variable, from which one of the three lexicon entries must be selected. There will be an *instrument* variable, from which either the entry shown in Figure 23 will be used or the newly created dummy entry. These variables can then be processed by a solution synthesis algorithm. Whenever a choice is made, for instance selecting *ACQUIRE-1* for **AQUIRE**, the constraint satisfaction mechanism in HUNTER-GATHERER will eliminate all conflicting sub-plans. Picking entry *ACQUIRE-1* will eliminate the dummy entry for *instrument.* Choosing entry *ACQUIRE-2* will eliminate all of the non-dummy *instrument* plans, as well as all the sub-plans that are created by the *instrument* plans. In this way, local means-end plans can be linked together, but can be processed globally by an efficient solution synthesis control. Figure 25 graphically displays the usage constraints for a portion of the example problem. The dotted lines connecting plans indicate compatible usage constraints.

Usage constraints are implemented by adding a series of preconditions and effects to each lexicon entry. For instance, for *ACQUIRE-1* to "request" that an *instrument* slot be filled, the following

[64]"Dummy" plans are explained next.

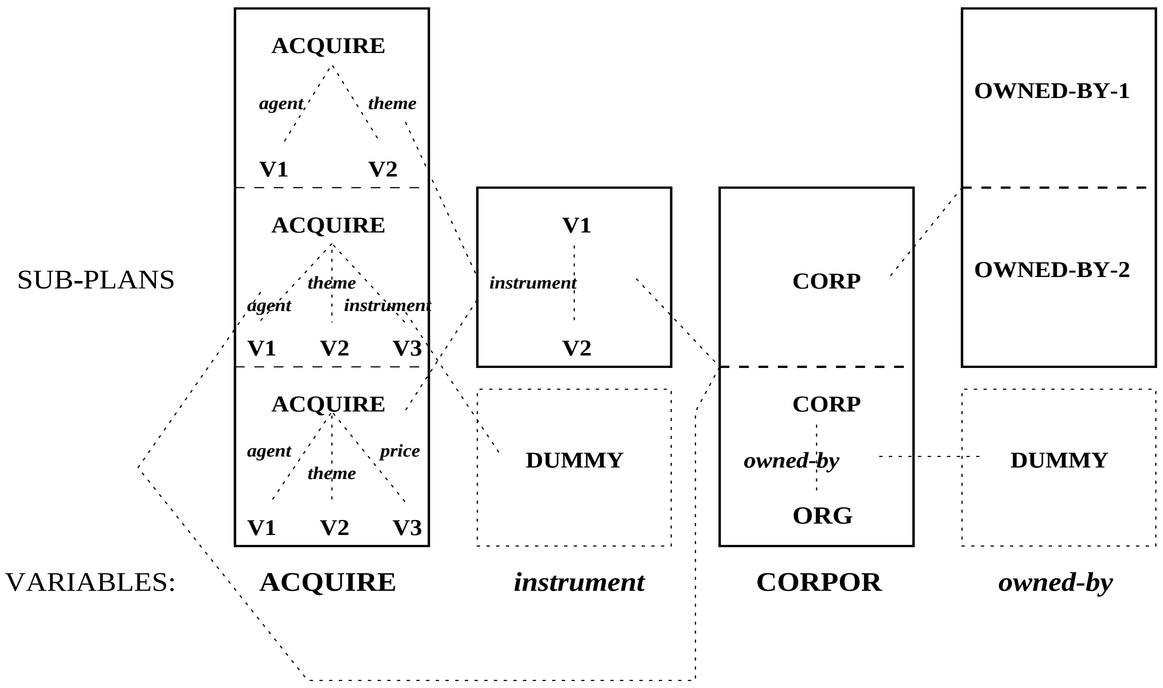

Figure 25: Usage Constraints

effect is added to it:

EFFECT: (FILL (ACQUIRE *instrument*))

Each of the non-dummy plans for *instrument* - only one in this case - then receive a precondition:

PRECONDITION: (FILL (ACQUIRE *instrument*))

This precondition cannot be fulfilled unless another plan with the corresponding effect is used.

To request a dummy filler, an effect like this is added:

EFFECT: (FILL (ACQUIRE *instrument-dummy*))

The dummy *instrument* then is given this precondition:

PRECONDITION: (FILL (ACQUIRE *instrument-dummy*))

Each concept (like **ACQUIRE**) and relation (like *instrument*) is linked to actual, uniquely-named structures in the input semantic representation. For example, (FILL (ACQUIRE *instrument*)) would actually look like (FILL (ACQUIRE-21 *instrument-23*)). This prevents confusion when more than one of the same concepts or relations is present.

Effects and preconditions can also be added to prevent redundant planning. For instance, if there was a **CORPORATION** lexicon entry that had a "built in" *instrument* link, combining this with *ACQUIRE-2* would over-generate the *instrument* meaning. Constraints can be added to ensure no input relation or concept is used twice.

To summarize, a means-end planner is used **locally** to set up possible sub-plans. The sub-plans are connected with a system of usage constraints that inhibit or allow usage depending on the other sub-plans being used. The HUNTER-GATHERER system can then efficiently process the collection of sub-plans to find the best overall plan. Constraint satisfaction techniques described in (Beale, 96) automatically control the combination of sub-plans. Constraint satisfaction also ensures the soundness of **all** preconditions used in the lexicon entries, including those which are not related to the ideas presented above. Efficiency is gained by restricting the means-end planning component to local sub-problems. Solutions to these sub-problems are then combined, utilizing solution synthesis, branch-and-bound and constraint satisfaction, by HUNTER-GATHERER.

Generation in the Mikrokosmos project is a relatively new development. Currently we are developing methods to reverse multilingual analysis lexicons (Viegas & Beale, submitted). PICARD has been used to back-translate the semantic analyses of the Mikrokosmos analyzer using these reversed lexicons. Efficiency results similar to those reported for HUNTER-GATHERER above were obtained.

The HUNTER-GATHERER algorithms are complete with respect to the set of monotonic solutions. Currently, solutions with plans that temporarily violate preconditions of other plans (with the "violation" corrected by a later plan) will not be allowed. Besides this limitation, HUNTER-GATHERER is guaranteed to find the same solution(s) as an exhaustive search. In addition, the constraint satisfaction component of HUNTER-GATHERER ensures soundness. By converting means-end planners into a format that can be used by HUNTER-GATHER, PICARD achieves efficient processing with guaranteed soundness and completeness without sacrificing the generality of means-end planning.

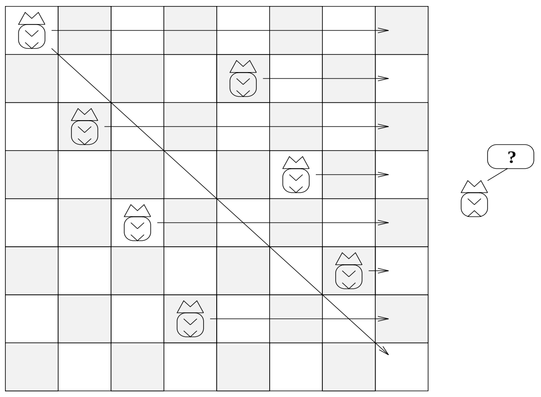

Figure 26: The 8-Queens Problem

3.5. Natural Language - a "Natural" CSP

There are two kinds of problems in the AI world:[65] the naughty and the nice. The naughty problems are those where everything depends on everything else. The N-Queens problem in Figure 26 is a good example. Even if some poor unfortunate is able to place N-1 Queens legally,[66] placement of the N^{th} Queen can be foiled by any of the N-1 already placed. These naughty problems typically are what people have in mind when they think about CSPs. Massive amounts of constraints going every which way. Look at just about any CSP reference (for example, [Tsang, 1993]) and you will see a preponderance of problems for which there will be no presents under the tree.

Fortunately (or perhaps unfortunately - see below), real world problems are not naughty, but nice. Often a real world problem gains its complexity not from dense interconnections of constraints, but simply from the large number of possible solutions. Natural Language is a perfect example of the latter. Given a sentence of length 23 (an average-looking sentence), and assuming each word can have two meanings, this yields 2^{23} possible combinations of word senses, or over eight million.

As strange as it may seem, though, in the CSP world "naughty" problems are actually "nice," and "nice" problems can be extremely "naughty." Figure 27 reproduces a table from [Tsang, 1993]. "Naughty" problems are those on the right-hand side, tightly constrained. For a human, keeping track of a large number of interacting constraints makes the problem difficult, that is why we tend to think of them as "naughty." For a computer, though, the constraints actually help, using CSP, to make the problem easier. The loosely constrained problems on the left side for which all solutions are required from among a large set of possible solutions are described in the table as "hard by nature."

The first major point of this research is to show that computational semantic problems are, in fact "naughty," and thus are "nice" in the CSP world. More precisely, we will demonstrate that computational semantic problems **are** tightly constrained **locally**, and CSP techniques can be used to great advantage in identifying these local interactions and determining their solutions. Perhaps

[65]As in Santa's.

[66]The object of N-Queens is to place N Queens on an N X N chessboard such that no Queen attacks any other Queen.

Solutions Required	Tightness of the Problem	
	Loosely Constrained	Tightly Constrained
Single Solution Required	Problem is easy by nature; brute force search (e.g. simple backtracking) would be sufficient	Problem reduction (i.e. CSP) helps to prune off search space, hence could be used to improve search efficiency
All Solutions Required	When the search space is large, the problem is hard by nature.	Problem reduction helps to prune off search space; solution synthesis has greater potential in these problems than in loosely constrained.

Figure 27: CSP Problem Types

even more important, we can use the fact that certain parts of the problem are **not** constrained[67] to guide solution synthesis most efficiently.

Another aspect of computational semantics that can cause problems with regard to applying CSP techniques is the fact that the constraints often do not have "yes" or "no" answers. CSP relies on definite answers to prune away inconsistent solutions. In natural language semantics, however, nothing is ever straightforward. Is the "White House" a HUMAN? No, it is a building, and yet we can say "The White House said today that ..." without even thinking about calling the Ghost Busters. Does this eliminate computational semantics from the domain of CSP problems? In one sense, it does. A straightforward application of CSP consistency algorithms would yield little, since constraints in are only tendencies; metonymy and figurative language often override these tendencies. However, branch-and-bound techniques combined with solution synthesis can be used to efficiently drive an computational semantic search engine. The efficiency, though, as shown above, is derived from information gained by analyzing constraint dependencies. This is the second major point of this research: that the branch-and-bound / solution synthesis paradigm is a type of constraint analysis problem, with its efficiency coming from analysis of dependencies within the problem.

Finally, we demonstrated that one of the central tools in Natural Language Generation, the means-end planner, can be more efficiently implemented by framing the planning process in terms of a CSP, and then using straightforward CSP consistency algorithms to determine possible solutions. This is the third, and last, major point of this work. Thus, the computational semantic problem itself, as well as one of the major tools used in solving it, can best be seen as being a type of CSP. It's a "natural" fit.

3.5.1. Local Dependency in Computational Semantics

It is evident from a quick look at Figure 13, repeated here as Figure 28, that constraints in that sentence are locally bundled. *Grupo-Roche* has nothing to do with *Espana*. The meaning of the one can be determined independently of the other. That is not to say that they cannot influence

[67]Determining "unconstrainedness" is a peculiar, but potentially powerful outcome of constraint analysis.

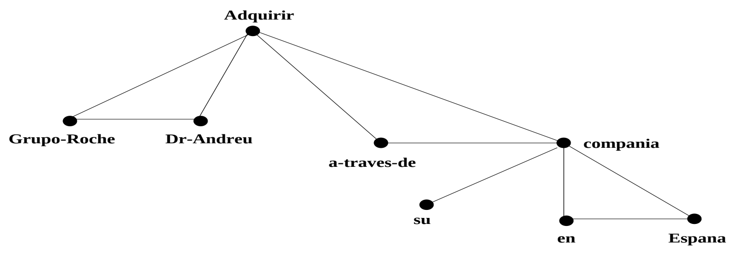

Figure 28: Constraint Dependencies in Sample Sentence

each other. For instance, *Espana* helps the analyzer choose the **location** meaning of *en*, which through a series of other interactions, could possibly influence the choice of meaning for *adquirir*, which, finally, could influence *Grupo-Roche*. However, this chain of dependencies is exactly what constraint-based mechanisms handles. As far as direct dependencies, though, the two words are not linked.

This is the general state of affairs in Natural Language. Government and Binding theory [Haegeman, 1991] is built on the assumption that one part of a text **governs** another, and interactions can only occur under this relationship. Government is restricted, among other things, to nodes that syntactically **command** other nodes. A node commands another node (again, among other things), if it is higher in the syntactic tree, and both are on the same path to the topmost node. This constraining property of syntax excludes non-governing relationships, which, in effect, partitions sentences into independent bundles. Smaller bundles can be combined into larger bundles as one moves higher up in the syntactic tree, where government domains become larger. This fact will be used to great advantage when constructing the "circles" for solution synthesis, as described below.

It is fairly obvious that, at the sentence level, computational semantics tends to bundle dependencies into these circles. What about larger sections of text? This research claims to be a step toward implementation of practical, large-scale, "real" computational semantic systems. Such systems eventually will address discourse issues. Can the claim that dependencies are locally bundled be maintained?

Yes, and no. [Grosz and Sidner, 1986] identify three aspects of discourse: the linguistic structure, the intentional structure and the attentional state. The first two seek to identify segments of text and give their purpose. In function, they are very similar to Rhetorical Structures [Mann and Thompson, 1983]. The attentional state, on the other hand, is an abstraction of the participants' focus of attention. This can be either global, or local [Grosz, 1981]. Knowledge of the attentional state is needed in reference resolution (and generation).

The rhetorical structure of a text links together chunks of text and identifies the function of the composition. In Rhetorical Structure Theory (RST), a nuclear section of text is joined to a satellite. Constraints between the nucleus and satellite are typically constraints between the main events of the main clauses of the sub-texts involved. For a constraint-based analyzer/planner, this simply adds an extra constraint link between the two sub-texts. In practice, this may cause final decisions at the sentence level to be delayed[68] until later in the discourse. This, of course, is a desirable situation. Often decisions cannot be made until the global purpose of a text or sub-text

[68]Unless all other decisions can be ruled out locally using the techniques described above.

has been determined. In fact, this discourse oriented processing is a main driving force behind a constraint-based approach. If a 23 word sentence produces millions of combinations, how many combinations would a thousand word text produce? Constraints must be used to intelligently prune the space of possibilities to a minimum, limiting interaction between sentences only to the bare minimum.

Attempts at processing the attentional state have concentrated on local focus. "Centering" theory [Grosz, et al. 1983] is an attempt to constrain reference resolution to the immediate context. Such efforts have proven to be effective for many texts; however, it is recognized that local focus alone cannot solve all reference problems.

[Elhadad, 1990] argues that conversations are locally constrained. He uses a constraint-based[69] paradigm to generate turns in a conversation. Each turn is linked to the previous turn by five local constraints. He cites [Sacks, et al, 1978] to support the contention that the most important characteristic of dialog is that it is locally managed.

In practice, local focus and local dialog constraints can be tracked independently of the main analyzer/planner. Before each sentence is processed, the focus and dialog constraints can be calculated for that sentence. These constraints can then be added to the local processing of each sentence. Tracking global focus can also be added to this independent mechanism. Thus, these phenomena do not pose a problem for CSP techniques.

On the other hand, certain aspects of Text Generation such as planning sentence length are heavily influenced by global considerations. What has come before and what comes next, the complexity of the preceding text, the surface length of the realizations of sub-parts of the current sentence, as well as global considerations of style; all these impact on sentence boundary decisions. Some of these factors can be tracked independently, similar to focus and local dialog constraints. The complexity and surface length of the current text, and of the text yet to be processed, however, are difficult to measure until the surface forms have been generated. For instance, a precondition such as "Sentence length < 25" cannot be satisfied by a single effect, but only from the combination of many effects. Effects could be created that increment a global variable, which is then referenced by the precondition; however, this creates a situation where one sub-plan is constrained by every other sub-plan, which destroys the computational efficiency of HUNTER-GATHERER (see the "Classes of Problems" section below.) In these cases, constraint-based planners hold little advantage.

The best solution to this problem probably lies in a post-processor that can examine the output text and suggest revisions based on measures of global surface features (Robin, 1994; Inui et al., 1992). Another possible solution would be to eliminate decisions based on surface features, such as the number of words planned, and replace these constraints with semantically based ones, such as the number of concepts. Or, perhaps, with a little more thought, using the global variable approach described above might be implemented in a way that does not impact efficiency. We leave these matters of handling surface constraints for future research.

[69]Elhadad uses functional unification to enforce constraint satisfaction. Although this certainly works, it has none of the efficiency advantages presented here.

3.5.2. Classes of Problems for which HUNTER-GATHERER is Beneficial

Several different features must come together to produce a problem for which HUNTER-GATHERER is the preferred processing methodology:

- Constraint-based. HUNTER-GATHERER gains its efficiency by identifying and processing independently parts of the problem that are tightly constrained. A knowledge of constraints between various parts of the problem is therefore essential.

- Constraints are tendencies. Traditional CSP techniques are adequate for problems whose constraints are of the straightforward yes/no variety. The N-Queens problem is a good example of such a problem. Many "real-world" problems, however, do not have such a simple semantics. Context often decides whether a certain decision is preferred (or even relevant) or not. The branch-and-bound methods used by HUNTER-GATHERER can be viewed as constraint satisfaction for "fuzzy" constraints. Constraints are combined in context, with value assignment combinations that are guaranteed to be sub-optimal removed. Solution synthesis provides for efficient combination of partial solutions, while knowledge of constraint dependencies recorded in the "circles" guides the synthesis.

- Relatively independent circles. Imagine a new N-Queens' problem where the constraints were "fuzzy." For instance, consider setting up constraints such that queens in odd-numbered columns attacking queens in other odd-numbered columns was relatively bad (possibly a score of 0.2), odd-numbered attacking even-numbered was not quite so bad (0.5), even-numbered combatants were not bad at all (a score of 0.9), and queens not attacking any other queens (pacifists) were given a score of 0.1. This violent version of N-Queens could then have the goal of finding the highest scored placement of queens. Obviously, straight-forward CSP techniques would be useless in this problem. HUNTER-GATHERER, although able to find the solution, cannot take advantage of its solution synthesis mechanism. A simple branch-and-bound algorithm would be enough to handle this, but it might take several centuries to process a 20-Queens' problem.

  The key reason the new N-Queens' problem is so difficult is because it is impossible to divide it into relatively independent sub-problems. HUNTER-GATHERER gains its efficiency in its ability, at each level of synthesis, to find at least one variable that is not effected outside the current synthesis circle. This variable can be optimized in all the possible combinations of variables that are effected outside the circle. In the new N-Queens' problem, every variable affects every other variable, making it impossible to perform this optimization.

  In terms of constraint graphs, the prototypical form of suitable problems will be tree-shaped. Circles are formed from the leaves up, by combining all children with their parent into a circle. At each synthesis, then, all of the children nodes will be optimized, since only the parent node is effected outside the circle.

  The prototypical form of unsuitable constraint graphs is the clique. In a clique, each variable affects every other variable. Circles cannot be constructed in this situation which will enable optimization at any level of synthesis.

  Of course, there is a wide spectrum of problems in between these two extremes. The "constraint-based" circles described in section 3.1 is a beginning towards identifying and exploiting non-tree-based constraint groups. Again, the key to making such circles beneficial is finding ones that contain variables that are not effected outside the circle. Further research in this area may enable the application of HUNTER-GATHERER to a wider spectrum of problems.

In summary, constraint satisfaction finds the circles, branch-and-bound optimizes "fuzzy" constraints, and solution synthesis combines together partial solutions. These functions, in turn, depend on, or take advantage of, the availability of constraints, the "fuzzy" nature of the constraints, and the localized nature of the interactions.

4. Discussion

In this section, we discuss two central issues: island processing and the formal properties of HUNTER-GATHERER.

4.1. Planning and Island Processing

One of the central characteristics of almost any natural, complex problem is that parts of its solution are fixed by available resources, while other parts may have a wide variability in possible solutions. For instance, in planning a cross-country trip, a traveler might have some general goals such as "visit as many places as possible", "enjoy the vacation", etc.. There may be some specific goals such as "see the Grand Canyon". Unfortunately, there will be some general constraints as well: "spend less than 1000 dollars", "get back home in two weeks". There probably will be some very specific constraints as well. For example, "go to the meeting in Phoenix on Monday, 4-13, at 10 AM (so you can write-off the vacation)" and "visit Aunt Millie on her birthday". Some other constraints are fairly restrictive when combined with other constraints: "visit Cousin Fred while I am in Phoenix".

When planning, the smart traveler will first determine the areas where he has no choices. The other decisions will then be made in relation to the fixed **islands of certainty** in the schedule. Upon making an initial assessment of the island constraints, other islands may appear, such as "visit Fred while in Phoenix" in the context of "go to the meeting in Phoenix on 4-13...". Islands constraints can pop up at even with the most general of constraints. For instance, if the piggy bank is empty after the Grand Canyon, there is only one place to go.

Considering that island driving is such a central component of human planning, it is surprising that it receives so little emphasis in the planning literature. This project seeks to remedy that situation with respect to computational semantic systems. HUNTER-GATHERER automatically uses its constraint satisfaction engine to take advantage of islands. Whenever a variable's plan, or value, is failed for any reason, constraint satisfaction will fail any other plans that critically[70] depend on effects from the failed plan, If a variable has all of its possible plans failed except one, forming an island, then only plans that do not critically rely on those failed plans will remain. In addition, when the non-failed plan that forms the island is processed, all plans that conflict with it will also be failed. Thus, by dynamically implementing constraint satisfaction techniques, islands are automatically identified and their effects propagated.

The second aspect to island driving in this project concerns the artificial creation of islands by the solution synthesis processor. In the solution synthesizer, valid combinations of plans are created and tested. For each combination, one of the plans, or values, is chosen and instantiated for each variable in the combination. This, in effect, creates artificial islands at each of these variables. The "island effects" can then be propagated out for each. Again, this might cause other plans to fail, creating other islands that are also artificial, in the sense that they were created by a combination of constraints imposed by planning choices rather than by the problem itself.

An interesting sidepoint in this discussion is that we treat text generation and semantic analysis equivalently; that is, they are both instances of planning. The "variables" in analysis are words,

[70] No other non-failed plans exist that could also satisfy its constraints.

the “plans” are word senses. The analyzer then tries to plan the combination of word senses that best describes the semantics of the input. In generation, the “variables” are semantic concepts or relations that need to be realized, the “plans” are textual directions for implementing those “variables,” and the generator plans the combination of textual directions that best implements the input semantics. With the exception of the “usage constraints” introduced by PICARD which enable HUNTER-GATHERER’s solution synthesis mechanism in the slightly more “fluid” world of generation, analysis and generation are processed equivalently.

4.2. Formal Properties of HUNTER-GATHER: Soundness and Completeness

4.2.1. Soundness

[Chapman 87] discusses precondition “clobbering”, the state where a rule that had been instantiated previously on the basis of some precondition, later had that precondition removed, invalidating the rule. Many previous text planners were not “sound”, in that they did not detect this kind of situation. To prove more formally that this system is sound, let me state the soundness criterion which it claims to adhere to:

Soundness Criterion: At all times, every rule that is not marked as currently failed must, for each of its preconditions, have at least one active, non-failed plan that has an effect that satisfies the precondition if it is a positive constraint (i.e. some state **must** exist for the precondition to be satisfied, identified by a “check-con” in the precondition field), or have no plans active that have an effect equal to the precondition if it is a negative constraint (i.e. the state may **not** exist, identified by a “check-not-con”).

Thus, if plan 1 has a precondition that states that constraint A must be true, then whenever plan 1 is not marked failed, there must be at least one other active plan that has an effect that produces A. On the other hand, if plan 1 has a precondition that states that precondition B must not be true, then there can be no plans active that produce B while rule 1 is not failed.

Note that this is slightly stronger than a general soundness condition, which might allow a planner to go through unsound states as long as the final result is sound. These “non-monotonic” plans, though valid, are currently excluded by HUNTER-GATHERER. We do not feel that this is a big drawback, since, in practice, a planner that is guaranteed to be sound at the end most likely will be sound throughout. This is all the more true for computational semantic planners which do not have complex preconditions and effects.

This system meets the soundness criterion given above. After some basic precondition application, which removes any rules that do not meet unary constraints (for example, “only apply this rule in formal contexts”), the dependencies between all the remaining rules are analyzed. The following information is recorded:

1. for each precondition of each plan, the plans are recorded that, if instantiated, would produce constraints that conflict with the precondition [71]

[71] A constraint A is considered conflicting when the precondition requires that constraint A **not** be set. Also judged as conflicting is the plan that sets constraint A when the precondition requires constraint -A

2. the inverse of the above: for each plan, if it were instantiated, record all the preconditions that conflict with it.
3. for each precondition that requires a constraint A, record all of the plans that, if instantiated, would set constraint A.
4. the inverse of the above: for every plan, if it were instantiated, record all the preconditions that would be satisfied by it.

Before solution synthesis is initiated, every plan that has a precondition for which there were no plans found in number 3 above is failed. Every time a plan is failed, both during this initial process and during the subsequent syntheses, all of the preconditions it can satisfy (from number 4 above) are retrieved. For each of these preconditions, all of the plans that could satisfy the precondition (from number 3 above) are retrieved, and it is checked that at least one of these plans is still active. If none are, then the plan corresponding to the precondition is failed. This process ensures that no plans are ever deleted that are the sole suppliers of preconditions of other active plans (or more accurately, that any such active plans are failed if their sole supplier is failed).

This does not yet fail preconditions that conflict with constraints. Whenever a variable has only one valid plan left[72], all of the preconditions that conflict with the plan (from number 2 above) are retrieved and the corresponding plan(s) are failed. The plans affected by these failures are then checked as described above.

During the solution synthesis, valid combinations of plans are chosen for each variable in the synthesis. In a combination, each variable has one plan chosen and the others are failed. The effects of failing the other plans are checked, and the effects of instantiating the plan that is used are checked. In general, whenever a plan is failed, the plans for which it supplied preconditions are checked. Whenever an island is created, the plans that conflict with the island are failed.

The fact that the system is sound is good in itself. But this is not where the benefits end. The whole process of ensuring soundness as described above, combined with island processing, creates a very efficient text planner. All conflicting rules and impossible rules are eliminated at the earliest point possible. Circles are created for the solution synthesizer that will maximize this effect. A plan is not ever failed, and then, after further processing, a different plan found to be invalid as a result. As soon as a plan is failed or an island created, all the other plans that can no longer be valid are removed immediately. Removing these plans then may lead to further removals by the same process. This process feeds on itself to remove as many of the possible plans as possible.

4.2.2. Completeness

HUNTER-GATHERER is complete[73] because it guarantees the same results as an exhaustive planner. The solution synthesizer, at every step, exhaustively calculates all of the valid combinations of plans for that synthesis. The only combinations that are removed are:

[72]This can happen because there was only one plan to start with, or because the soundness procedure above eliminated one or more plans in a node, or the process being described here eliminated one or more plans in a node, or a combination of the last two processes left only one valid plan, or, finally, the solution synthesis mechanism chose one plan and "failed" all the others

[73]Except for the non-monotonic plans and surface constraint types described below.

1. Combinations that can be guaranteed, by branch-and-bound techniques, to produce complete solutions that are not optimal.
2. Combinations that contain constraint conflicts.

Exhaustive planners are obviously complete, so HUNTER-GATHERER must be as well.

Two exceptions exist, however. The first has to do with the types of constraints allowed by HUNTER-GATHERER. One of the drawbacks inherent in a text generator like Penman is the inability of the modules to communicate with each other. One of the main characteristics of driving this project is the inter-action of choices available at different levels. HUNTER-GATHERER specifically allows for these inter-actions. There is, however, a class of constraints that this system cannot at present address. It is not able to use constraints that arise from the combination of plan effects. Thus a constraint such as "if the clause is already 30 words long, make a sentence boundary" cannot be used. This limitation exists because simple precondition-effect pairs cannot be set up. A precondition like "sentence 30 words long?" is satisfied by combining the effects of many plans together. Section 3.5 addresses this problem and suggests a number of possible alternative methods for handling it.

The second limitation with regards to completeness has already been mentioned. Non-monotonic plans are not allowed. Non-monotonic plans are those which at intermediate stages contain constraint conflicts which are later resolved in the overall plan. While we recognize that this may be a severe limitation in generalized AI planning, we do not feel it presents much of a problem for computational semantics, where preconditions and effects are generally fairly simple. As we look into using HUNTER-GATHERER in other applications, we plan on investigating this limitation further.

5. Conclusion

We have presented a new control environment for processing computational semantics. By combining and extending the AI techniques known as constraint satisfaction, solution synthesis and branch-and-bound, we have reduced the search space from billions or more to thousands or less. We have argued that the search problems encountered in computational semantics fit nicely into the class of problems that this control paradigm handle well.

In the past, the utility of knowledge-based semantics has been limited, subject to arguments that it only works in "toy" environments. Recent efforts at increasing the size of knowledge bases, however, have created an imbalance with existing control techniques which are unable to handle the explosion of information. We believe that this methodology will enable such work. Furthermore, because this work is a generalization of a control strategy used for simpler binary constraints, we believe that it is applicable to a wide variety of real-life problems. We intend to test this control paradigm on problems outside NLP.

Acknowledgements

I would like to thank Sergei Nirenburg for his continual encouragement and contributions to this work. Also appreciated are Kavi Mahesh and Evelyne Viegas for their work on the Mikrokosmos project as well as their helpful suggestions on this paper. Thanks to Remi Zajac for his thoughtful review of this material. And finally, I would also like to acknowledge and thank David Evans and Barbara DiEugenio, who, along with Sergei Nirenburg, were my advisors at Carnegie Mellon where research on an earlier version of this work was done.

References

[1] Stephen Beale. 1994. Dependency-Directed Text Generation. Technical Report, MCCS-94-272, Computing Research Lab, New Mexico State Univ.

[2] Stephen Beale and Sergei Nirenburg. 1995. Dependency-Directed Text Planning. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence, Workshop on Multilingual Text Generation*, 13-21. Montreal, Quebec.

[3] Stephen Beale and Sergei Nirenburg. Submitted. HUNTER-GATHERER: Three Search Techniques Integrated for Natural Language Semantics. Submitted to the Thirteenth National Conference on Artificial Intelligence (AAAI96), Portland, Oregon.

[4] Stephen Beale, Sergei Nirenburg and Kavi Mahesh. 1995. Semantic Analysis in the Mikrokosmos Machine Translation Project. In *Proceedings of the 2nd Symposium on Natural Language Processing*, 297-307. Bangkok, Thailand.

[5] D. Chapman. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32.

[6] E. Charniak, C.K. Riesbeck, D.V McDermott and J.R. Meehan. 1987. *Artificial Intelligence Programming.* Erlbaum, Hillsdale, NJ.

[7] Michael Elhadad. 1990. Constraint-based Text Generation. Technical Report, CUCS-003-90, Dept. of Computer Science, Columbia Univ.

[8] E.C. Freuder. 1978. Synthesizing Constraint Expressions. *Communications ACM* 21(11): 958-966.

[9] Barbara J. Grosz and Candace L. Sidner. 1986. Attentions, Intentions, and the Structure of Discourse. *Computational Linguistics* 12(3): 175-204.

[10] Liliane Haegeman. 1991. *An Introduction to Government and Binding Theory.* Blackwell Publishers, Oxford, U.K.

[11] K. Inui, T. Tokunaga and H. Tanaka. 1992. Text Revision: a Model and its Implementation. In *Aspects of Automated Natural Language Generation*, R. Dale, E. Hovy, D. Roesner and O. Stock, editors. Springler-Verlag.

[12] E.W. Lawler and D.E. Wood. 1966. Branch-and-Bound Methods: a Survey. *Operations Research* 14: 699-719.

[13] A.K. Mackworth. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8(1): 99-118.

[14] A.K. Mackworth and E.C. Freuder. 1985. The Complexity of Some Polynomial Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence* 25: 65-74.

[15] Kavi Mahesh and Sergei Nirenburg. 1995. A situated ontology for practical NLP. In *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing, International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada.

[16] Boyan Onyshkevych and Sergei Nirenburg. 1994. The Lexicon in the Scheme of KBMT Things. Technical Report MCCS-94-277, Computing Research Lab, New Mexico State University.

[17] K.R. McKeown. 1985. *Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.* Cambridge University Press.

[18] R. Mohr and T.C. Henderson. 1986. Arc and Path Consistency Revisited. *Artificial Intelligence* 28: 225-233.

[19] Allen Newell and Herbert Simon. 1972. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, N.J.

[20] Jacques Robin. 1994. Revision-Based Generation of Natural Language Summaries Providing Historical Background. Technical Report CUCS-034-94, Columbia University.

[21] Roger Schank and R. Abelson. 1977. *Scripts, Plans, Goals and Understanding.* Erlbaum, Hillsdale, N.J.

[22] Josh D. Tenenberg. 1991. Abstraction in Planning. In *Reasoning about Plans.* James A. Allen, Henry A. Kautz, Richard N. Pelavin and Josh D. Tenenberg, eds. Morgan Kaufmann Publishers, San Mateo, Ca.

[23] Edward Tsang. 1993. *Foundations of Constraint Satisfaction.* Academic Press, London.

[24] Edward Tsang and Nigel Foster. 1990. Solution Synthesis in the Constraint Satisfaction Problem. Technical Report, CSM-142, Dept. of Computer Science, Univ. of Essex.

[25] Evelyne Viegas and Stephen Beale. Submitted. Multilinguality and Reversibility in Computational Semantic Lexicons. Submitted to the 8th International Workshop on Natural Language Generation, Sussex, UK.

[26] R. Michael Young and Johanna D. Moore. 1994. DPOCL: A Principled Approach to Discourse Planning. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, Kennebunkport, ME.