Improving Domain Filtering using Restricted Path Consistency

P. Berlandier

SECOIA Project, INRIA–CERMICS, B.P. 93, 06902 Sophia Antipolis, FRANCE

Abstract

This paper introduces a new level of partial consistency for constraint satisfaction problems, which is situated between arc and path-consistency. We call this level restricted path-consistency (RPC). *Two procedures to enforce complete and partial* RPC *are presented. They both use path-based verifications to detect inconsistencies but retain the good features of arc-consistency since they only touch the variables domains and do not augment the connectivity of the constraint graph. We show that, although they perform a limited number of checks, these procedures exhibit a considerable pruning power.*

1 Introduction

Constraint satisfaction problems (CSP) have proved useful to encode various instances of combinatorial problems. A CSP is simply defined by giving a set of variables, each having a finite domain, and a set of constraints, each connected to a subset of the variables. Constraints are partial informations that restrict the values that can be assigned simultaneously to their variables.

Enforcing the *global consistency* of a CSP consists in finding a set of value assignments, one for each variable, so that all the constraints are simultaneously satisfied. This is a NP-complete problem that is usually attacked with now sophisticated [Tsang 93] but still exponential search procedures. It is thus crucial to narrow the search space as much as possible by enforcing some level of *partial consistency* with a polynomial time filtering procedure.

Currently, the best known levels of partial consistency are arc and path-consistency. The first one is used almost universally because it can be computed at low cost and its enforcement simply comes to the elimination of some value assignments (which is quite valuable since value elimination represents a maximal simplification operation with regard to the problem combinatorics).

Now, except for some temporal reasoning applications, path-consistency does not have the same good press. There are mainly three reasons for this. The first one is that the ratio between its complexity and the simplification factor it brings is far less interesting than the one brought by arc-consistency.

Then comes the fact that the action of a path-consistency procedure amounts to the elimination of *pairs* of values assignments [Han 88]. This imposes that constraints should have an extensive representation from which individual pairs can somehow be deleted (e.g. boolean matrices [Montanari 74]). Such a representation is often unacceptable for the implementation of real-world problems for which intensive representations are much more concise and efficient.

Finally, enforcing path-consistency has the major disadvantage of bringing some modifications to the connectivity of the constraint graph by adding some edges to this graph. These drawbacks are the reasons why path-consistency algorithms are almost never implemented in commercial CSP-solving systems.

In this paper, we present a level of partial consistency called *restricted path-consistency* (RPC) which is half-way between arc and path-consistency. The procedures that will enforce this consistency level will turn the above three problems of path-consistency by their incompleteness: path-consistency checking will be engaged for a given assignment pair if and only if the deletion of this pair implies the deletion of one of its element.

Such a situation occurs when a given assignment pair represents the *only* support for one of the two assignments with regard to some constraint. Below is a simple example of this state of affair. Figure 1.1 shows the compatibility graph (also called the *microstructure*) of a binary CSP with three variables i, j and k.

As it is defined here, the problem is initially arc-consistent. However, for example, the status of the assignment $\langle i, a\rangle$ is rather fragile since it has *only one* supporting value on j (i.e. value c).

If we now consider path-consistency between a and c, we observe that there are no existing path through k. Therefore, the assignment pair $(\langle i, a\rangle, \langle j, c\rangle)$ could be ruled out, leaving a without any support on j. This implies in turn that assignment $\langle i, a\rangle$ can be eliminated. From there, it is possible to reapply arc-consistency to propagate this elimination and delete

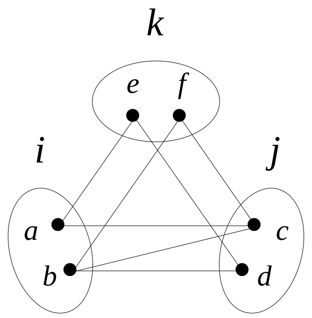

Figure 1.1: initial situation

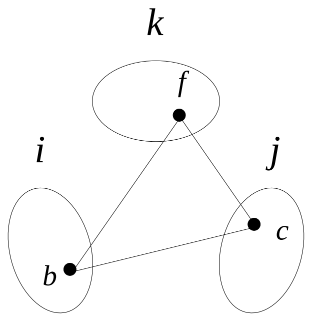

Figure 1.2: after enforcing RPC

successively the values e for k and d for j. The resulting problem (which, incidentally, is globally consistent) is shown on figure 1.2.

We thus can see that, focusing our attention on a limited number of correctly chosen assignment pairs, it is possible to operate a domain filtering that is more powerful than the one offered by arc-consistency alone. Due to its incomplete nature, RPC breaks away from the drawbacks of path-consistency:

- The complexity of the filtering process is lowered since the number of assignment pairs that are to be checked is greatly restricted.
- No particular representation for the constraints is necessary since no assignment pair is withdrawn from the constraints.
- No additional constraint is created by the filtering process since, once again, there is no pair to withdraw.

The rest of the paper shows how restricted path-consistency can be built on top of arc-consistency. In section 2, we recall some definitions about CSP, arc and path-consistency and we give a formal definition of RPC. In sections 3 and 4, we present the details of two RPC procedures that are developed using AC$_4$ [Mohr 86]. Finally, in section 5, we evaluate experimentally the merits of RPC on some randomly generated constraint problems.

2 Preliminaries

A binary constraint satisfaction problem P is a quadruple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ in which:

- $\mathcal{V}$ is a set of n variables $\{1, \ldots, n\}$;
- $\mathcal{D}$ is the set of domains $\{D_1, \ldots, D_n\}$ corresponding to each variable. A value assignment $\langle i, a \rangle$ is possible iff $a \in D_i$. We note d the maximum size of the domains.
- $\mathcal{C}$ is the set of constraints. Each constraint is a pair of distinct variables $\{i, j\}$ noted C_{ij} for convenience. We note e the size of $\mathcal{C}$.
- $\mathcal{R}$ is the set of relations associated to the constraints. The relation R_{ij} defines the set of legal pairs of values for i and j.

Reaching global consistency for a CSP means finding a value assignment for each variable in $\mathcal{V}$ so that all the constraints in $\mathcal{C}$ are satisfied. Partial consistency imposes some weaker conditions about the possible value assignments of the problem. For arc and path-consistency, the conditions are the followings:

- a variable i is arc-consistent iff:

  for all $a \in D_i$,
    $j \in \mathcal{V}$ such that $C_{ij} \in \mathcal{C}$,
  there exists $b \in D_j$ such that $(a, b) \in R_{ij}$.

- a pair of variables $\{i, j\}$ is path-consistent iff:

  both i and j are arc-consistent,
  for all $(a, b) \in D_i \times D_j$,
    $k \in \mathcal{V}$ such that C_{ik} and $C_{kj} \in \mathcal{C}$,
  there exists $c \in D_k$ such that $(a, c) \in R_{ik}$ and
          $(c, b) \in R_{kj}$.

Based on the above, we can derive the following definition for restricted path-consistency:

- a variable i is restrictedly path-consistent iff:

  i is arc-consistent,
  for all $a \in D_i$,
    $j \in \mathcal{V}$ such that $C_{ij} \in \mathcal{C}$ and
          $\exists! b \in D_j, (a, b) \in R_{ij}$,
    $k \in \mathcal{V}$ such that C_{ik} and $C_{kj} \in \mathcal{C}$,
  there exists $c \in D_k$ such that $(a, c) \in R_{ik}$ and
          $(c, b) \in R_{kj}$.

A CSP is said arc-consistent iff all of its variables are arc-consistent and none of them have an empty domain. Similarly, a CSP is said path-consistent iff all of the possible pairs of variables are path-consistent. Therefore, we will say a CSP is restrictedly path-consistent iff all its variables are restrictedly path-consistent. By definition, a CSP that is restrictedly path-consistent, is also arc-consistent (of course, the reciprocal is false). Now, a CSP that is path-consistent is also restrictedly path-consistent (and again, the reciprocal is false).

3 Enforcing RPC

To enforce restricted path-consistency in a binary CSP, we have to enforce arc-consistency and also to select the pairs of assignments that are eligible for path-consistency checking. To achieve the latter operation, we need to count the supports of all possible value assignments with respect to all possible constraints. Now, this accounting is precisely the keystone of the AC_4 algorithm [Mohr 86]. It thus make AC_4 a natural choice for the development of RPC.

Figure 2 shows the three procedures that implement complete RPC. Procedures *initialize* and *prune* are directly derived from AC_4. As in the original algorithm, *initialize* builds two data structures: S_{ia} which stores the set of assignments supported by value a at variable i and $counter[(i,j),a]$ which counts the number of supports on j of the assignment $\langle i,a\rangle$.

The main difference of *initialize* and *prune* from their original versions is the respective addition of lines 11 to 14 and 7 to 14. In the procedure *initialize*, those four lines allow, when the support counter of an assignment has just been initialized, to check whether the assignment in question has got only one support left. If this is the case, we enqueue the information in $list_{PC}$ under the form of a triple describing the assignment, the variable on which the one support remains and the third variable through which a path will be searched.

In the procedure *prune*, reaching line 7 means that the pair $(\langle i,a\rangle,\langle j,b\rangle)$ has just been deleted and that this deletion did not provoke the deletion of the value a for i. We are then faced with one of the following cases:

- There is only one remaining arc that supports the value a on i with respect to j. We are thus back to the case handled by lines 11 to 14 of *initialize* where a triple $(\langle i,a\rangle,j,k)$ is enqueued for every variable k connected to both i and j.
- The lately deleted arc might have been the only one that could compose a path from i through j to a third variable k such that a has got only one support on k. For every such k, we thus have to enqueue the triple $(\langle i,a\rangle,k,j)$. This case is handled by lines 11 to 14 of *prune*.

When *prune* ends, the procedure *check* is invoked with the $list_{PC}$ that has just been determined. For each triple $(\langle i,a\rangle,j,k)$ in this list, we determine in S_{ia} the value b that is the only support of a on j (cf. line 4). Then, given the variable k connected both to i and j, we check that there is a path from a to b passing through k. If this is not the case, we know that value a can be deleted from the domain of i. We thus decrement its support counter and propagate its suppression in the graph by reapplying the procedure *prune*.

```
{step 1: initializing the data structure}

procedure initialize(var $list_{AC}$, var $list_{PC}$);
1 forall $(i,j)$ such that $C_{ij} \in \mathcal{C}$ do
2    forall $a \in D_i$ do
3       $total \leftarrow 0$;
4       forall $b \in D_j$ do
5          if $(a,b) \in R_{ij}$ then
6             $total \leftarrow total + 1$;
7             $S_{ia} \leftarrow S_{ia} \cup \{\langle j,b\rangle\}$;
8       $counter[(i,j),a] \leftarrow total$;
9       if $counter[(i,j),a] = 0$ then
10         $list_{AC} \leftarrow list_{AC} \cup \{\langle i,a\rangle\}$; $D_i \leftarrow D_i \setminus \{a\}$
11         else if $counter[(i,j),a] = 1$ then
12                 forall $k \in \mathcal{V}$
13                 such that $C_{ik}$ and $C_{kj} \in \mathcal{C}$ do
14                    $list_{PC} \leftarrow list_{PC} \cup \{(\langle i,a\rangle,j,k)\}$.

{step 2: pruning inconsistent values}

procedure prune($list_{AC}$, var $list_{PC}$);
1 while $list_{AC} \neq \emptyset$ do
2    choose and delete $\langle j,b\rangle$ from $list_{AC}$;
3    forall $(i,a) \in S_{jb}$ do
4       $counter[(i,j),a] \leftarrow counter[(i,j),a] - 1$;
5       if $counter[(i,j),a] = 0$ and $a \in D_i$ then
6          $list_{AC} \leftarrow list_{AC} \cup \{\langle i,a\rangle\}$; $D_i \leftarrow D_i \setminus \{a\}$
7       else if $counter[(i,j),a] = 1$ then
8               forall $k \in \mathcal{V}$
9               such that $C_{ik}$ and $C_{kj} \in \mathcal{C}$ do
10                 $list_{PC} \leftarrow list_{PC} \cup \{(\langle i,a\rangle,j,k)\}$
11           else forall $k \in \mathcal{V}$
12                such that $C_{ik}$ and $C_{kj} \in \mathcal{C}$ do
13                   if $counter[(i,k),a] = 1$ then
14                      $list_{PC} \leftarrow list_{PC} \cup \{(\langle i,a\rangle,k,j)\}$.

{step 3: checking path existence}

procedure check(var $list_{PC}$);
1 while $list_{PC} \neq \emptyset$ do
2    choose and delete $(\langle i,a\rangle,j,k)$ from $list_{PC}$;
3    if $a \in D_i$ then
4       let $\{\langle j,b\rangle\} = \{\langle j,x\rangle \in S_{ia} \mid x \in D_j\}$;
5       if $\{\langle k,c\rangle \in S_{ia} \cap S_{jb} \mid c \in D_k\} = \emptyset$ then
6          $counter[(i,j),a] \leftarrow 0$; $D_i \leftarrow D_i \setminus \{a\}$;
7          prune($\{\langle i,a\rangle\}$, $list_{PC}$).
```

Figure 2: Computing complete RPC

Note that the triples that become eligible for a test by *check* after the successive applications of *prune* are added incrementally to $list_{PC}$ by *prune* itself.

In like manner of the arc-consistency algorithm, the termination of the RPC algorithm is guaranteed by the monotonic decrease of the number of eligible assignment pairs and the decrease of $list_{PC}$ ensured by line 2 of *check*.

3.1 Complexity

Compared to AC_4, our algorithm introduces the list of triples $list_{PC}$. This list reaches its maximal size when, for all the variables of the problem, each assignment has only one support with respect to each constraint. In this case, the size of the list is in $O(ned)$. The space complexity of AC_4 being $O(ed^2)$, this gives us an upper bound of $O(ed(n+d))$ for RPC.

Let us now deal with the time complexity. The updates that we brought to the procedures *initialize* and *prune* do not change the complexity of AC_4. Now, for procedure *check*, the loop at line 1 is executed at most *ned* times. Of course, the cost of the executions of *prune*, called at line 7, is covered by the worst-case cost calculated for the execution of arc-consistency. Therefore, the time complexity of *check* is in $O(ned)$.

This gives us a upper bound time complexity of $O(ed(n+d))$ for RPC. For the comparison, we recall that the time complexity of the best path consistency algorithm, PC_3 [Han 88], is in $O(n^3d^3)$.

4 Enforcing Partial RPC

The algorithm presented above is complete i.e. it stops only when the RPC property presented in section 2 is fully installed in the constraint problem. While testing this algorithm on examples, we noticed that ensuring this completeness usually requires some extra work that is of little profit from a domain filtering point of view.

A partial but fast enforcement of RPC can be achieved by checking only once every possible value assignment after arc-consistency has been enforced. To implement this, the simple *partial-check* procedure, presented on figure 3 can be used. The *prune* procedure that is invoked here is the basic AC_4 propagation procedure since we do not wish to maintain the revision list $list_{PC}$ any more.

The upper bound space and time complexities of AC_4 are not affected by the addition of the *partial-check* procedure. Indeed, *partial-check* does not introduce any new data-structure and its worst case time complexity is in $O(ed)$.

```
procedure partial-check;
1 forall $i \in \mathcal{V}$ do
2    forall $a \in D_i$ do
3       forall $j \in \mathcal{V}$ such that $C_{ij} \in \mathcal{C}$ do
4          if $counter[(i,j),a] = 1$ then
5             let $\{\langle j,b\rangle\} = \{\langle j,x\rangle \in S_{ia} \mid x \in D_j\}$;
6             if $\{\langle k,c\rangle \in S_{ia} \cap S_{jb} \mid c \in D_k\} = \emptyset$ then
7                $counter[(i,j),a] \leftarrow 0$; $D_i \leftarrow D_i \setminus \{a\}$;
8                $prune(\{\langle i,a\rangle\})$.
```

Figure 3: Computing partial RPC

5 Experimental Evaluation

In order to check out the benefits brought by our consistency procedures, we have experimented them on some random constraint problems.

As usual, the generation of random problems is based on four parameters: the number n of variables, the size d of the variables domain, the constraint density *cd* in the graph and the constraint tightness *ct*.

The constraint density corresponds to the fraction of the difference in the number of edges between a n-vertices clique and a n-vertices tree. A problem with density 0 will show $n-1$ constraints; a problem with density 1 will show $n(n-1)/2$ constraints.

The constraint tightness *ct* corresponds to the fraction of the number of tuples in the cross-product of the domain of two variables that will not be allowed by the constraint between these two variables. Tightness 0 stands for the universal constraint and tightness 1, the unsatisfiable constraint.

We have run two sets of experiments, one with $n = 16$ and $d = 8$ and the other with $n = 8$ and $d = 16$. For each set, we had the constraint density range from 0 to 1 by steps of 0.05 and for each density, we have repeated the execution of the filtering procedures on 100 different instances, reporting the average of the results. The performances of basic arc-consistency (AC), partial RPC and complete RPC are compared on three grounds:

- the execution time,
- the portion of the possible assignments that are ruled out after the execution of the procedure i.e.
$$1 - \frac{\sum_{i\in\mathcal{V}} \Delta_i}{\sum_{i\in\mathcal{V}} D_i}$$
where Δ_i is the domain of i after being filtered. This indicates the raw quantity of filtering work that has been achieved.
- the portion of the initial search space that is ruled out after the execution of the procedure i.e.
$$1 - \frac{\prod_{i\in\mathcal{V}} \Delta_i}{\prod_{i\in\mathcal{V}} D_i}$$

This indicates the impact of the filtering work on the problem simplification.

Figures 4 and 5 show the results that we have obtained for two sets of parameters that have been found relevant i.e. the problems that were generated from these parameters were correctly constrained: enough for arc-consistency to have some effect and not too much for inconsistencies to be detected right away.

Our main observations are:

- *The computation time added by the application of partial* RPC *is small with regard to the time consumed by* AC. On our examples, the application of partial RPC takes at most half more time than the application of AC. This turns out to be a small additional cost when it is weighted against the corresponding portion of the pruned search space. A reason why the additional cost of RPC is limited is that the most expansive part of the process is the initialization step, which is achieved by AC. So, in some sense, the application of RPC allows a better amortizing of the work that is done during this initialization phase.
- *The application of* RPC *offers a far better domain filtering power than the application of* AC *alone.* Figure 4 shows that RPC (be it partial or complete) can detect inconsistent problems very early, i.e. starting at a constraint density of 0.5 while, with the use of AC alone, inconsistent problems are never detected. Figure 5 shows that complete RPC performs a better filtering job than partial RPC on easy problems. However, complete RPC is much more expansive and this expanse is not rewarded since these problems are anyway easy to solve!
- *Partial* RPC *does almost the same job as complete* RPC *with far better performances.* Partial RPC appears as a good tradeoff between the time consumed and the filtering power. On the one hand, the execution time of partial RPC seems to be linearly related to the cost of AC whereas the cost of complete RPC is diverging for high constraint densities. On the other hand, the pruning capability of partial RPC stays close to the one of complete RPC especially for difficult problems (e.g. problems with a density between 0.4 and 0.5 in figure 4).

6 Conclusion

We have presented a new level of partial consistency and two procedures to enforce it. These procedures are based on the AC_4 algorithm and therefore, every arguments against the use of AC_4 also fall on them.

First, there has been some criticisms against the space greediness of the data structures used by AC_4. Then, in [Wallace 93] it was established that AC_3 algorithm [Mackworth 77] performs almost always better than AC_4 to enforce arc-consistency.

This is true when arc-consistency is considered as a preprocessing step but it is not any more when arc-consistency is maintained incrementally during the resolution of the problem as shown in [Sabin 94]; here again, the reason is that the cost of building the data structures for AC_4 is well amortized by their repeated use during the search. One future work is thus to explore the benefits of using RPC during the search on hard problems, as it was done with full arc-consistency in the MAC algorithm [Sabin 94].

Future developments of RPC are primarily two generalizations. The first one concerns the handling of n-ary constraints using the GAC_4 algorithm [Mohr 88], which is itself a generalization of AC_4.

The second one concerns the restriction that we have imposed on the maximum number of supports of an assignment beyond which we will not bother to trigger path-consistency checking. Here, this number was fixed to 1 because of the good property it induced. But it might be interesting to investigate the effect of using a variable bound and perhaps tune this bound according to the tightness of the constraints.

References

[Han 88] C. Han, C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[Mackworth 77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mohr 86] R. Mohr, T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Mohr 88] R. Mohr, G. Masini. Good old discrete relaxation. In *Proc. ECAI*, Munich, Germany, 1988.

[Montanari 74] U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7(3):95–132, 1974.

[Sabin 94] D. Sabin, E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. ECAI*, Amsterdam, Netherlands, 1994.

[Tsang 93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Wallace 93] R. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proc. IJCAI*, Chambéry, France, 1993.

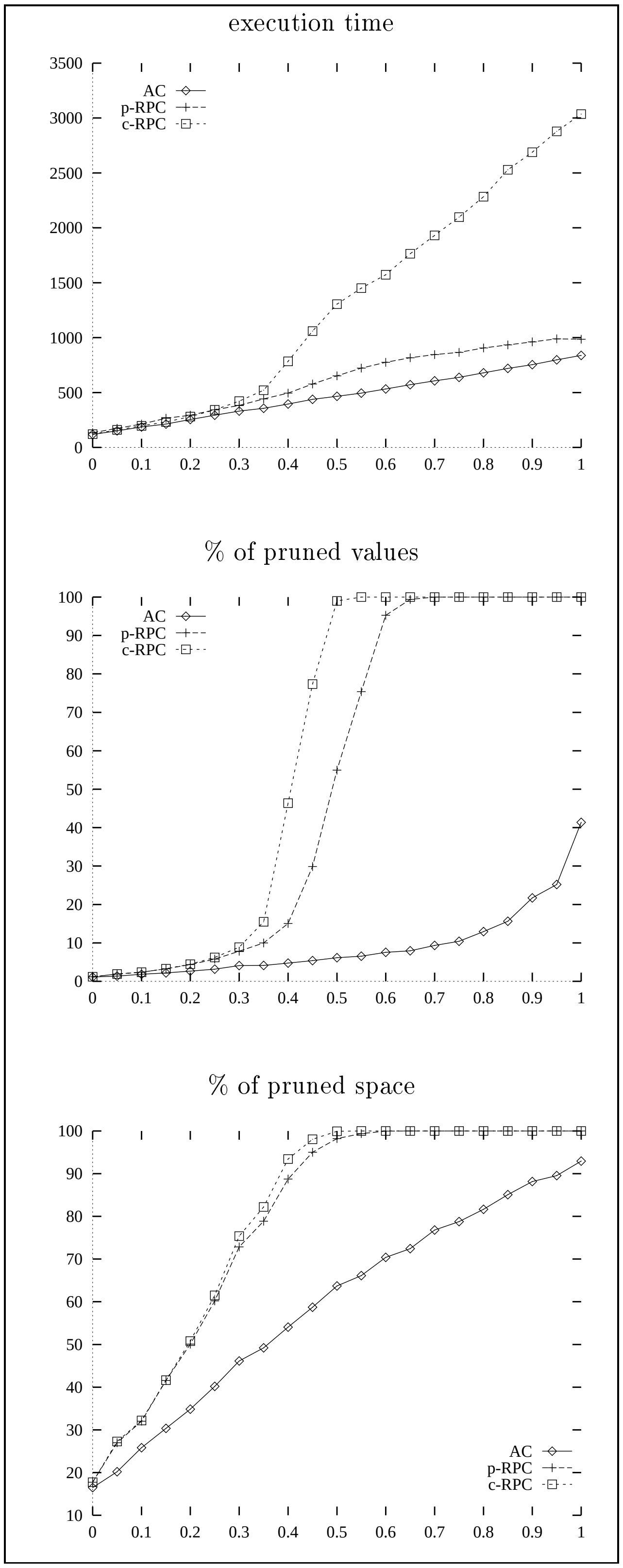

Figure 4: $n = 16$, $d = 8$, $ct = 0.5$

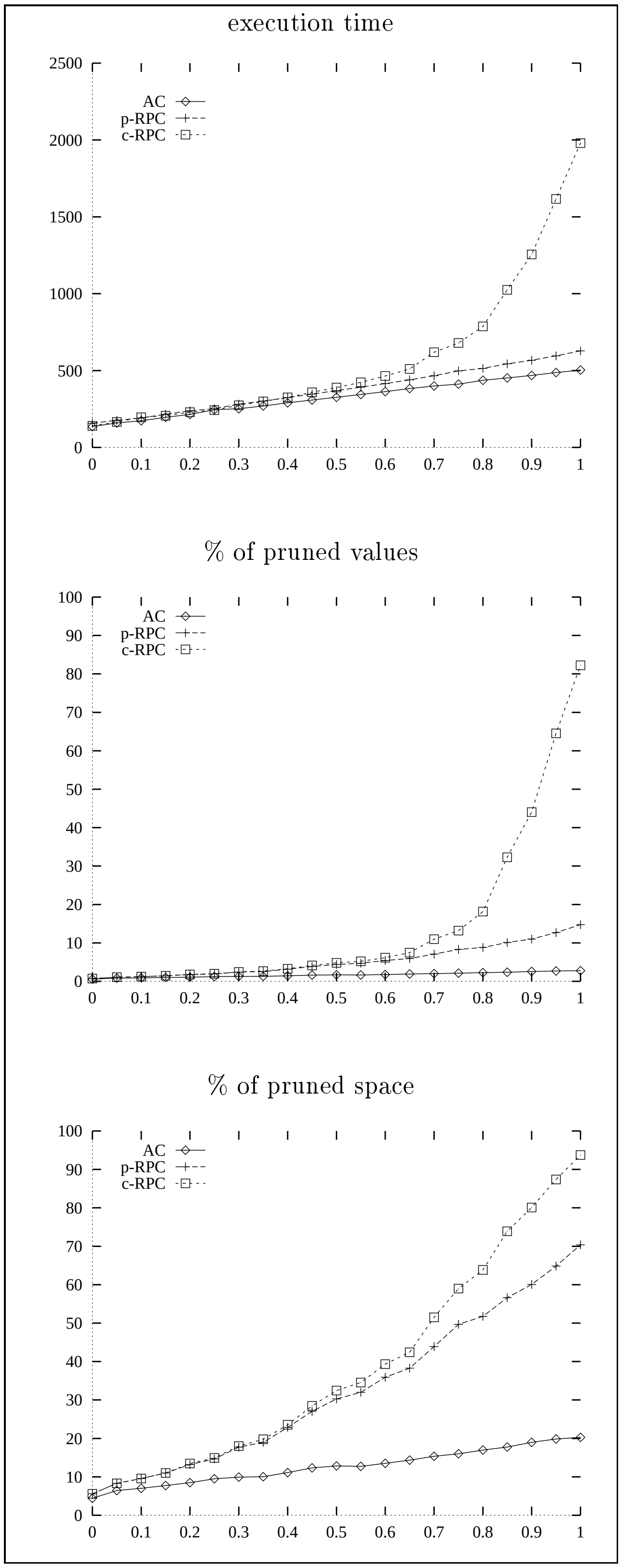

Figure 5: $n = 8$, $d = 16$, $ct = 0.7$