Comparison of Several Path-Consistency Algorithms

Christopher Thiel

August 19, 2008

1 Introduction

This paper compares several path-consistency algorithms experimentally on uniformly distributed random constraint satisfaction problems (CSPs). Like other consistency methods, path-consistency algorithms use inference to make constraint networks more explicit. CSPs are made more explicit by adding constraints that are unstated in the original problem. In general, this reduces the search space of partial solutions [Dechter(2003)]. Furthermore, we can sometimes detect inconsistent problems using consistency methods.

We will show some of the tradeoff between completeness and resource usage. We will also compare the efficiency of different implementations of the same idea. The algorithms we compare are PATH-CONSISTENCY-2 (PC-2), DIRECTIONAL-PATH-CONSISTENCY (DPC), PARTIAL-PATH-CONSISTENCY (PPC), TRIANGLE-PARTIAL-PATH-CONSISTENCY ($\triangle$PPC), and a proposed modification of $\triangle$PPC called $\triangle$PPC-2.

We begin by presenting the algorithms in Section 2. Then we discuss our experiments, including problem generation and performance metrics, in Section 3. The results of the experiments are presented in Section 4. Finally, we present some conclusions in Section 5.

2 Algorithms

This section presents each of the path-consistency algorithms compared in this paper. The REVISE-3 function [Dechter(2003)] is used in all of the path-consistency algorithms. REVISE-3$((x, y), z)$ is equivalent to the composition

$$R_{xy} \leftarrow R_{xy} \cap \pi_{xy}(R_{xy} \bowtie D_z \bowtie R_{zy})$$

where $\pi_{xy}(R)$ is the projection of R onto $\{x, y\}$ and where $\bowtie$ is the join operator, except that REVISE-3 additionally returns true when R_{xy} is modified. We abstract this composition into a function since we use the number of times it is called as a performance metric when comparing the various path-consistency algorithms. The REVISE-3 procedure is given in Algorithm 1.

Algorithm 1: Revise-3((x, y),z)

Input: A three-variable subnetwork over (x, y, z), R_{xy}, R_{yz}, R_{xz}.
Output: Revised R_{xy} path-consistent with z. Returns true if and only if R_{xy} was modified.

changed $\leftarrow$ false
foreach pair $(a, b) \in R_{xy}$ **do**
  if no value $c \in D_z$ exists such that $(a, c) \in R_{xz}$ and $(b, c) \in R_{yz}$ **then**
    remove (a, b) from R_{xy}
    changed $\leftarrow$ true
  end
end
return changed

All of our algorithms try to detect inconsistent problems. If a problem is determined inconsistent, then the consistency procedures return immediately. Thus, inconsistent problems may not be path-consistent when a consistency procedure terminates.

2.1 Path-Consistency-2

Path-Consistency-2 (PC-2) [Mackworth(1977)] is a classical algorithm for achieving path-consistency. PC-2 is shown in Algorithm 2. Our version of PC-2 returns false if a binary network $\mathcal{R}$ is detected as inconsistent; otherwise it returns true. Note that failure to detect an inconsistent problems does not imply that the problem is consistent. Inconsistent problems are revealed by an empty constraint, that is, a constraint in which all tuples have been filtered out.

2.2 Directional Path-Consistency

We next look at Directional Path-Consistency (DPC) [Dechter(2003)]. Unlike Dechter's DPC, the definition given here does not produce a *strong* directional path-consistent network. We do without arc-consistency here since the other path-consistency algorithms we present also perform no arc-consistency. The DPC procedure is shown in Algorithm 3. DPC does not complete the constraint graph as PC-2 does. Instead DPC considers only one specific variable order to make path-consistent. Achieving path-consistency relative to one ordering may be done more efficiently than when considering any ordering as PC-2 does, but if arbitrary ordering are to be considered, there is no guarantee of path-consistency for other orderings. Furthermore, DPC cannot detect inconsistent problems as well as PC-2. This is shown later in Section 4.

Algorithm 2: PC-2($\mathcal{R}$)

Input: A binary network $\mathcal{R} = (X, D, C)$.
Output: A path-consistent network $\mathcal{R}'$ equivalent to $\mathcal{R}$ when possible. Returns false if $\mathcal{R}$ is determined inconsistent.

$Q \leftarrow \{(i, j, k) \mid 1 \leq i < j \leq n, 1 \leq k \leq n, k \neq i, k \neq j\}$
while Q is not empty **do**
  select and delete a triple (i, j, k) from Q
  changed $\leftarrow$ Revise-3($(i, j), k$)
  if R_{ij} is empty **then**
    return false
  end
  if changed **then**
    $Q \leftarrow Q \cup \{(l, i, j), (l, j, i) \mid 1 \leq l \leq n, l \neq i, l \neq j\}$
  end
end
return true

Algorithm 3: DPC($\mathcal{R}$)

Input: A binary network $\mathcal{R} = (X, D, C)$, its constraint graph $G = (V, E)$, and variable ordering $d = (x_1, \ldots, x_n)$.
Output: A directional path-consistent network $\mathcal{R}'$ and its graph $G' = (V, E')$. Returns false if $\mathcal{R}$ is determined inconsistent.

$E' \leftarrow E$
for $k = n$ **to** 1 by -1 **do**
  foreach $i, j < k$ such that $(x_i, x_k), (x_j, x_k) \in E'$ **do**
    Revise-3($(x_i, x_j), x_k$)
    if R_{ij} is empty **then**
      return false
    end
    $E' \leftarrow E' \cup (x_i, x_j)$
  end
end
return true

2.3 Partial Path-Consistency

Partial-Path-Consistency (PPC) is an algorithm to make triangulated graphs path-consistent [Bliek and Sam-Haroud(1999)]. PPC is shown in Algorithm 4.

There exist several methods to triangulate an arbitrary constraint graph. We use the Min-Fill algorithm to triangulate all graphs when a consistency algorithm requires it.

Algorithm 4: PPC($\mathcal{R}$)

Input: A binary network $\mathcal{R} = (X, D, C)$ and its constraint graph $G = (V, E)$.

Output: A partial path-consistent network network $\mathcal{R}'$ and its graph $G' = (V, E')$. Returns false if $\mathcal{R}$ is determined inconsistent.

$Q \leftarrow E$
while Q is not empty **do**
    $q \leftarrow$ Front(Q)
    foreach $(i, k, j) \in$ Related-Triplets(q) **do**
        changed $\leftarrow$ Revise-3($(i, j), k$)
        if R_{ij} is empty **then**
            return false
        end
        if changed **then**
            Push($(i, j), Q$)
        end
    end
    Pop(Q)
end
return true

2.3.1 Min-Fill

Min-Fill is a graph triangulation heuristic. The Min-Fill procedure given in Algorithm 5 is based on that given in [Dechter(2003)], but the variable ordering proceeds forward rather than backward. The ordering produced is a perfect elimination order [Gogate and Dechter(2004)]. Also, we use the triangulated graph produced by the Min-Fill procedure, not just the variable ordering.

Note that the triangulated graph produced by Algorithm 5 is $G' = (V, E')$; the variable V' is used only locally to track eliminated nodes.

2.4 Triangle-Partial-Path-Consistency

Triangle-Partial-Path-Consistency ($\triangle$PPC) [Xu(2003)] is the proposed generalization of the $\triangle$STP algorithm given in [Xu(2003)]. As its name suggests, $\triangle$PPC intends to perform the same filtering as PPC, and in our experiments

Algorithm 5: MIN-FILL(G)

Input: A graph $G = (V, E)$.
Output: A triangulated graph $G' = (V, E')$ and a perfect elimination order σ.

$V' \leftarrow V$
$E' \leftarrow E$
for $j = 1$ **to** n **do**
  $r \leftarrow$ a node in V with the smallest number of fill edges for its parents
  put r is position j of σ
  $E' \leftarrow E' \cup \{(v_i, v_j) \mid (v_i, r), (v_j, r) \in E'\}$
  $V' \leftarrow V' \setminus \{r\}$
end

$\triangle$PPC and PPC do perform the same filtering for problems that are not detected as inconsistent. Unlike the edge queue used by PPC, $\triangle$PPC uses a queue of triangles. The main difference between PPC and $\triangle$PPC is that when $\triangle$PPC revises a constraint, it places all triangles onto the triangle queue that contain the revised edge.

2.5 Triangle-Partial-Path-Consistency-2

The TRIANGLE-PARTIAL-PATH-CONSISTENCY-2 ($\triangle$PPC-2) algorithm requires that we find a join-tree. This in turn requires us to find the maximal cliques in a constraint graph. Algorithm 7 finds the maximal cliques given a perfect elimination order σ, which we get from MIN-FILL. The presentation of CLIQUES is based on [Golumbic(2004)]. We use the notation $\sigma(i)$ to denote the element at position i of σ, and we use the notation $\sigma^{-1}(x)$ to denote the position of element x in σ.

The JOIN-TREE procedure [Dechter(2003)] given in Algorithm 8 finds a join-tree from the vector of maximal cliques given by CLIQUES.

$\triangle$PPC-2 differs from $\triangle$PPC by considering one clique at a time. After finding a join-tree, $\triangle$PPC-2 first looks at the cliques from the leaves to the root using a depth-first search postordering of the join-tree's cliques. $\triangle$PPC-2 calls the $\triangle$PPC procedure once for each clique using a constraint graph containing only the variables within the current clique. After processing the root node of a join-tree, $\triangle$PPC-2 then processes the cliques in the reverse order. Since we are working with a tree (the join-tree), this is a preorder of the cliques. This processing from the root to the leaves happens in an identical manner.

The idea behind this approach is to consider local triangles as a group. The triangles of a clique are necessarily connected to each other.

Algorithm 6: TRIANGLE-PARTIAL-PATH-CONSISTENCY($\mathcal{R}$, G)

Input: A binary network $\mathcal{R} = (X, D, C)$ and its constraint graph $G = (V, E)$
Output: A partially path consistent network and its constraint graph $G' = (V, E')$

$Q_T \leftarrow$ all triangles in G
while Q_T is not empty **do**
    $Q_E \leftarrow \emptyset$
    $\{i, j, k\} \leftarrow$ FRONT(Q)
    changed $\leftarrow$ REVISE-3((i, j),k)
    if $C_{i,j}$ is empty **then return** false
    if changed **then**
        PUSH((i, j),Q_E)
    end
    changed $\leftarrow$ REVISE-3((i, k),j)
    if $C_{i,k}$ is empty **then return** false
    if changed **then**
        PUSH((i, k),Q_E)
    end
    changed $\leftarrow$ REVISE-3((j, k),i)
    if $C_{j,k}$ is empty **then return** false
    if changed **then**
        PUSH((j, k),Q_E)
    end
    foreach $(m, n) \in Q_E$ **do**
        $\mathcal{T}_{m,n} \leftarrow$ all triangles containing (m, n)
        foreach $\{l, m, n\} \in \mathcal{T}_{m,n}$ such that $\{l, m, n\} \notin Q_T$ **do**
            PUSH($\{l, m, n\}$, Q_T)
        end
    end
    POP(Q_T)
end
return true

Algorithm 7: CLIQUES(G, σ)

Input: A triangulated graph $G = (V, E)$ and a perfect elimination order σ.

Output: A vector of cliques $\mathcal{C}$.

```
j ← 0
foreach v ∈ V do S(v) ← 0
for i ← 1 to n do
    v ← σ(i)
    X ← {x ∈ Adj(v) | σ^{-1}(v) < σ^{-1}(x)}
    if Adj(v) = ∅ then
        C(j) ← v
        j ← j + 1
    end
    if X = ∅ then return C
    u ← σ(min{σ^{-1}(x) | x ∈ X})
    S(u) ← max{S(u), |X| − 1}
    if S(v) < |X| then
        C(j) ← {v} ∪ X
        j ← j + 1
    end
end
return C
```

Algorithm 8: JOIN-TREE($\mathcal{C}$)

Input: A vector of cliques $\mathcal{C} = (C_1, \ldots, C_r)$.

Output: A join-tree T.

```
foreach C_i ∈ C do
    Connect C_i to a C_j (j < i) with whom it shares the largest subset of
    variables.
end
return Join-tree T such that the cliques in C are its nodes and the edges
created above are its edges.
```

Algorithm 9: TRIANGLE-PARTIAL-PATH-CONSISTENCY-2($\mathcal{R}$, G, T)

Input: A binary network $\mathcal{R} = (X, D, C)$, its constraint graph $G = (V, E)$, and a join-tree T.
Output:
$O_{\text{post}} \leftarrow$ a postorder of the cliques (nodes) of T
foreach ordered clique K in O_{post} **do**
    $V' \leftarrow$ the set of variables in K
    $\triangle$PPC($\mathcal{R}$, (V', E))
end
$O_{\text{pre}} \leftarrow$ a preorder of the cliques (nodes) of T
foreach ordered clique K in O_{pre} **do**
    $V' \leftarrow$ the set of variables in K
    $\triangle$PPC($\mathcal{R}$, (V', E))
end

3 Experiments

This section discusses our problem generation method and our performance metrics.

3.1 Random Problem Generation

The problems used to compare the algorithms are uniformly distributed CSPs. The generated problems are in the XCSP 2.1 format [XCSP()] using conflict relations for all constraints; thus, all constraints are given in extension. Problems with disconnected constraint graphs were discarded. The CSP parameters are similar to those used in [Chmeiss and Jegou(1998)] for path-consistency. The parameters are specified as a 4-tuple (n, a, t, p) where n is the number of variables, a is the domain size of each variable, t is the constraint tightness, and p is the constraint density. Constraint tightness is the ratio of conflict (disallowed) tuples in a given constraint out of all possible tuples:

$$t = \frac{|\text{conflict tuples}|}{|\text{all tuples}|} = \frac{|\text{conflict tuples}|}{a^2}.$$

Constraint density describes the ratio of constraints in a given problem out of the maximum number of constraints possible. It is defined as

$$p = \frac{e}{n(n-1)/2}$$

where e is the number of constraints in a problem. This assumes that between any two variables, there is at most a single constraint. Our generated problems follow this assumption.

All generated problems use $n = 32$ and $a = 8$ with tightness varying from $t = 0.1$ to $t = 0.9$ in 0.1 increments. We then compare the path-consistency algorithms at $p = 0.2$ and $p = 0.5$. For each (n, a, t, p) tuple, we generate 100 problems and present the average results. In some cases the values of t and p are only approximate. For example, with $n = 32$, an exact tightness of 0.2 requires 99.2 constraints. Since the number of constraints must be an integer, we round e to 99. This results is an actual tightness of approximately 0.1996.

3.2 Performance Metrics

We compare the path-consistency algorithms by four metrics:

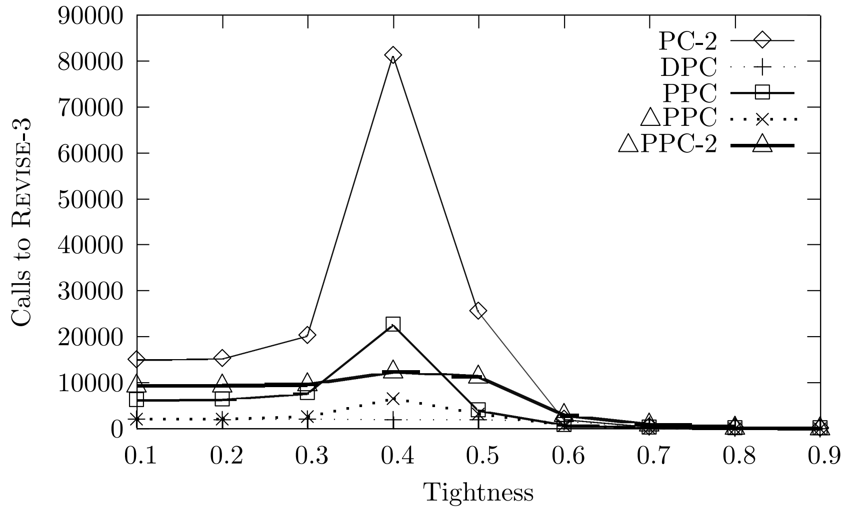
1. the number of calls to REVISE-3,
2. the number of tuples eliminated,
3. the CPU time, and
4. the number of inconsistent problems detected.

The number of times an algorithm calls REVISE-3 is simply a tally of such calls. These calls appear only in the path-consistency algorithms themselves, and not in any dependent algorithms, such as graph triangulation. Similarly, the count of eliminated tuples is incremented only in the path-consistency algorithms themselves. The count is comprised of the tuples filtered by a path-consistency algorithm. The CPU time, however, includes the time taken to perform any preprocessing of the problem by dependent algorithms. PC-2 and DPC require no preprocessing. PPC, $\triangle$PPC, and $\triangle$PPC-2 each require triangulation, so the CPU time of MIN-FILL is included. Furthermore, $\triangle$PPC-2 uses the CLIQUES and the JOIN-TREE procedures, so the CPU time of these procedures are included as well.

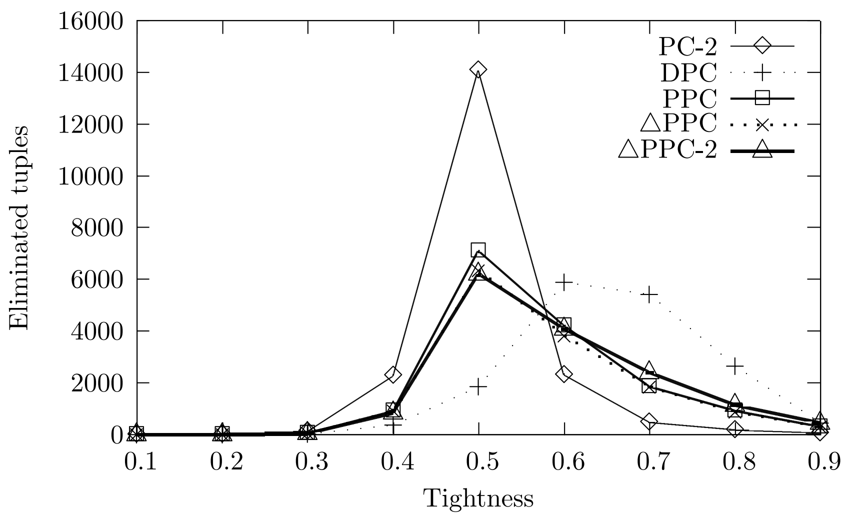
Since detecting inconsistent problems is important when one wants to actually solve a CSP, we keep track of the number of inconsistent problems detected. These numbers are compared to those determined inconsistent by PC-2 since no other pure path-consistency algorithm can detect more inconsistencies than it can. By pure path-consistency, we mean consistency algorithms that are concerned solely with determining 3-consistency.

4 Results

Figure 1 shows the revision results for each path-consistency algorithm with varying tightness and $n = 32$, $a = 8$, and $p = 0.2$. We group the calls to REVISE-3 and the number of eliminated tuples together since all filtering (elimination of tuples) occurs by way of REVISE-3. Given this, it is interesting to note that the number of eliminated tuples does not correspond closely with the number of calls to REVISE-3. From Figure 1a, we see that the phase transition occurs at around $t = 0.4$ — each algorithm makes more calls to REVISE-3 at $t = 0.4$ than at any other tightness. However, the number of eliminated tuples (Figure 1b) is maximum at $t = 0.5$ for all algorithms except DPC.



(a) Number of calls to REVISE-3.



(b) Number of eliminated tuples.

Figure 1: Average revision results for $n = 32$, $a = 8$, and $p = 0.2$.

tightness	PC-2	DPC	PPC	△PPC	△PPC-2
0.1–0.4	0/0	0/0	0/0	0/0	0/0
0.5	100/100	0/100	100/100	100/100	100/100
0.6–0.9	100/100	100/100	100/100	100/100	100/100

Table 1: Ratio of problems detected as inconsistent for $n = 32$, $a = 8$, and $p = 0.2$.

Table 1 shows the ratios of problems detected inconsistent by each algorithm compared to what PC-2 detected with $n = 32$, $a = 8$, and $p = 0.2$. We see that all problems with a tightness of 0.5 and greater are inconsistent, and that no problems with tightness less than 0.5 were detected as inconsistent. All algorithms performed similarly in this regard except for DPC. DPC failed to detect any of the 100 inconsistent problems when $t = 0.5$.

In Figure 1b, we see that before the phase transition ($t = 0.1$ to 0.3), all algorithms perform a relatively small number revisions. At the phase transition and after, the PPC-based algorithms perform roughly the same number of revisions. Interestingly, at $t = 0.5$, PC-2 does the most filtering by far, but at $t = 0.6$ and beyond it does the least, yet in both cases, all problems are inconsistent. DPC has a unique curve, peaking later than the others. Its peak is also broader than the others. This is not entirely surprising, however, since it is fundamentally different from the other algorithms. In any case, the increase in eliminated tuples does not cause a corresponding increase in the number of calls to Revise-3. Thus, DPC filters many more tuples per call to Revise-3 than the other algorithms on the tighter problems.

The average CPU times are shown in Figure 2. The CPU times for each algorithm and the calls to Revise-3 (Figure 1a) are very similar in relative terms.

Overall, DPC runs the fastest but does the least filtering on the set of problems not detected as inconsistent. Of the PPC-based algorithms, △PPC is the most efficient. It is faster than PPC and △PPC-2 in all cases while performing the same filtering for the problems not detected as inconsistent.

We now compare how the consistency algorithms perform when the density is increased to $p = 0.5$. By looking at Figure 3a we see that the phase transition occurs at $t = 0.3$, earlier than when $p = 0.2$. Again, the CPU times (Figure 4) correspond closely with the number of calls to Revise-3.

From Table 2, we can see that for $t = 0.4$ to 0.9 all problems are inconsistent, and that for $t = 0.1$ to 0.2 no problems were detected as inconsistent. Unlike when $p = 0.2$, we have a data point where the PPC-based algorithms detect less inconsistent problems than PC-2. This occurs at $t = 0.3$, the phase transition. PPC and △PPC find the same number of problems inconsistent (16/21) as expected, while △PPC-2 finds still fewer (7/21). This demonstrates that the PPC-based algorithms can fail to detect inconsistent problems as well as PC-2. It also shows that △PPC-2 is deficient in this regard against PPC and △PPC.

As with $p = 0.2$ △PPC is more efficient than PPC for the same filtering on

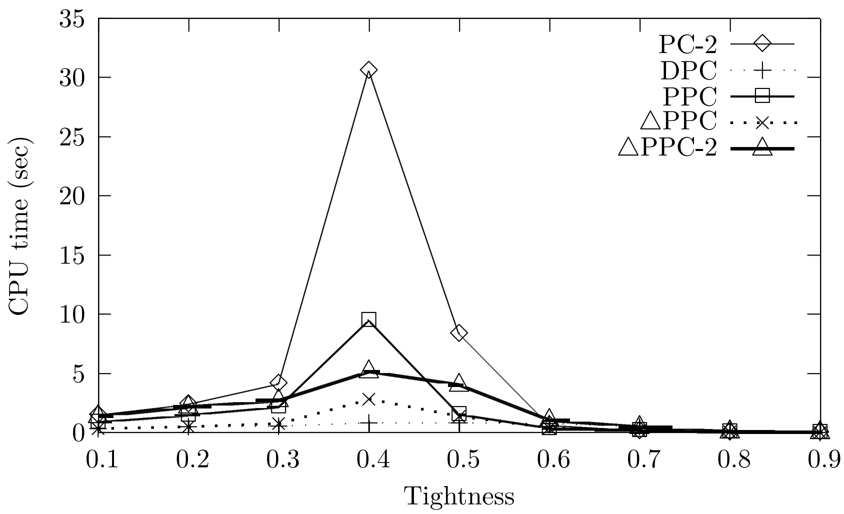


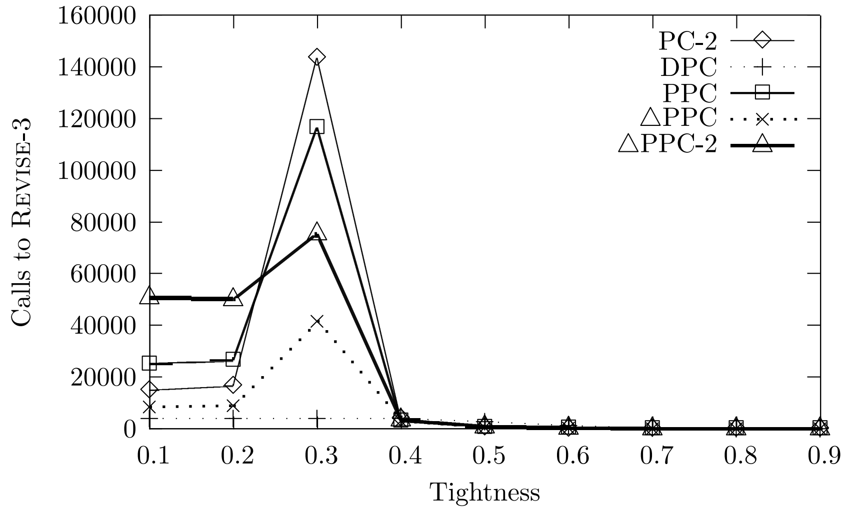
Figure 2: Average CPU time for $n = 32$, $a = 8$, and $p = 0.2$.

tightness	PC-2	DPC	PPC	△PPC	△PPC-2
0.1–0.2	0/0	0/0	0/0	0/0	0/0
0.3	21/21	0/21	16/21	16/21	7/21
0.4	100/100	69/100	100/100	100/100	100/100
0.5–0.9	100/100	100/100	100/100	100/100	100/100

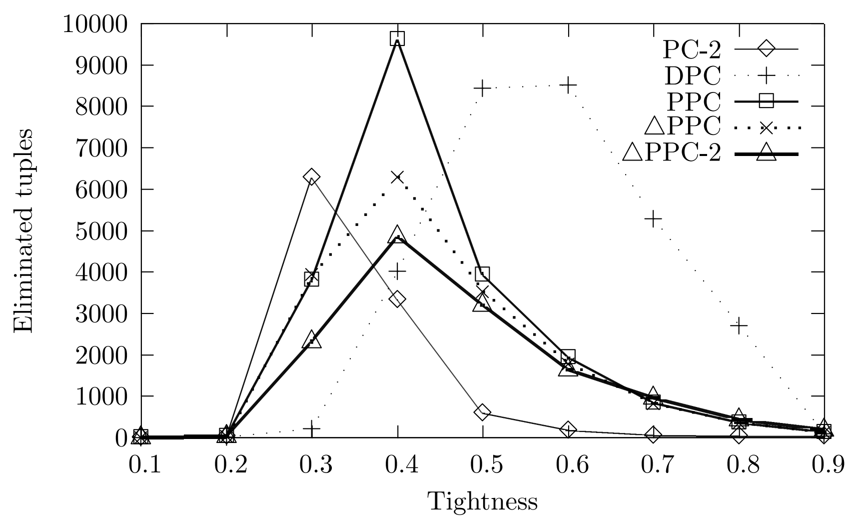
Table 2: Ratio of problems detected as inconsistent for $n = 32$, $a = 8$, and $p = 0.5$.

the possibly consistent problems. The results for $p = 0.5$ also expose a weakness of △PPC-2 — it does not detect inconsistent problems as well as PPC or △PPC. Moreover, we see that PPC can be more resource intensive than even PC-2 (at $t = 0.3$).

In an effort to make PPC and △PPC detect inconsistent problems as well as PC-2, we added an additional preprocessing step to the PPC-based algorithms. This step involves completing all paths of length two in the triangulated constraint graph by adding explicit universal constraints. The reasoning can be seen in the following example CSP:



(a) Number of calls to Revise-3.



(b) Number of eliminated tuples.

Figure 3: Average revision results for $n = 32$, $a = 8$, and $p = 0.5$.

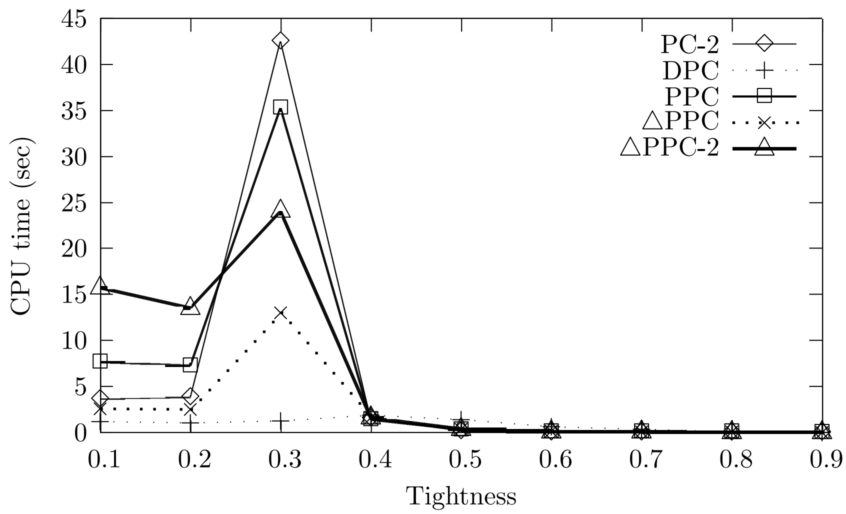
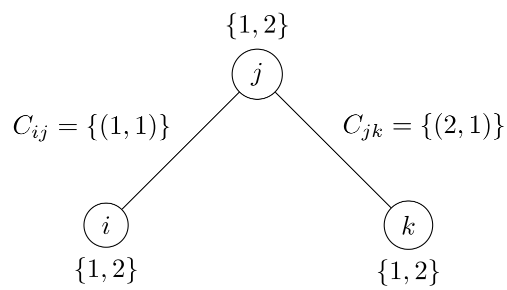


Figure 4: Average CPU time for $n = 32$, $a = 8$, and $p = 0.5$.



The supported tuples are given by the two constraints C_{ij} and C_{jk}. PC-2 will detect this problem as inconsistent. For example, it will notice that the only consistent assignments to i and j ($i = 1, j = 1$) cannot be extended to k. This will result in an empty constraint on the edge (i, j). The PPC-based algorithms operate on triangles, so this inconsistency will not be detected. Actually, these algorithms will do no filtering on this problem since there are no triangles. But completing the path (i, j, k) results in an explicit universal constraint between i and k. The PPC-based algorithms will now detect the inconsistency. Note that completing all paths of length two does not necessarily result in completing the entire constraint graph. In some cases, such as when the problems consists of a single path, much fewer additional edges are needed.

Results for the modified PPC-based algorithms are shown in figures 5 and 6. These results are for $p = 0.5$. PC-2 and DPC were not modified but are presented for comparison. The modification produced the desired results — all of the PPC-based algorithms now detected the same 21 inconsistent problems as PC-2 for $t = 0.3$ (compare with Table 2). The modification did increase the number of calls to Revise-3 (and CPU time) for these algorithms, most notably for PPC at $t = 0.3$.

5 Conclusions

Since PPC and △PPC do the same filtering on problems not determined inconsistent, they can be compared easily on such problems by looking at the average calls to Revise-3 (or CPU time). On the possibly consistent problems, △PPC outperformed PPC on average. Again, PPC and △PPC detect the same problems as inconsistent. On these problems, PPC is sometimes more efficient but not considerably.

As expected, △PPC-2 cannot detect as many inconsistent problems as △PPC. Furthermore, △PPC-2 is more resource intensive than △PPC in most cases, making it a poor replacement for △PPC (or PPC).

When we modify the PPC-based algorithms to work on triangulated graphs with paths of length two completed, △PPC-2 outperformed PPC, but it is still slower than △PPC. With further bookkeeping, △PPC-2 could be made to perform the same filtering as △PPC and PPC, but this would slow it down even more. It still may be possible to exploit the connectedness of the variables of the cliques and the relationships of the cliques in a join-tree to speedup △PPC-based algorithms.

Further work is needed to determine if completing paths of length two results in the PPC-based algorithms finding all of the inconsistent problems as PC-2. Given this, it is difficult to compare △PPC to PC-2 when this is desired. Looking solely at the filtering done by PC-2 and △PPC, PC-2has the advantage at the expense of increased resource usage.

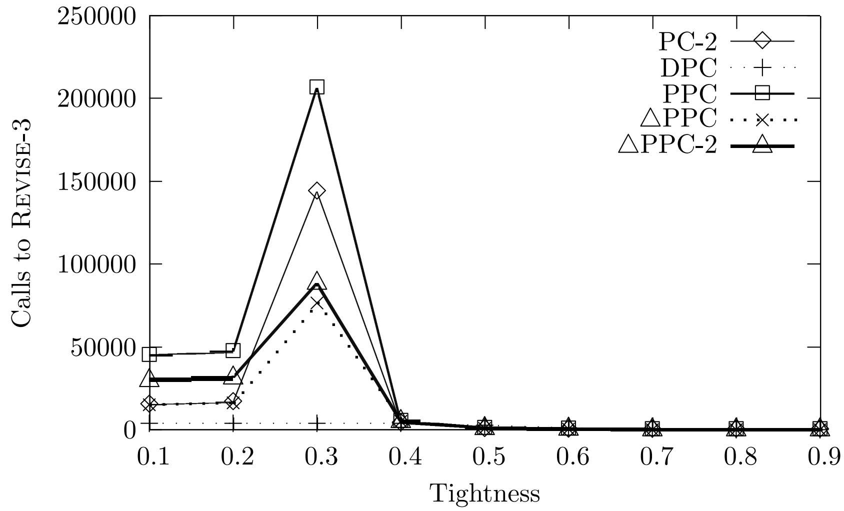
Even though △PPC runs faster than PC-2, it is still considerably slower than DPC, whose Revise-3 and CPU time curves remain effectively flat. This is not a fair comparison, however, since DPC does the least filtering by far.

References

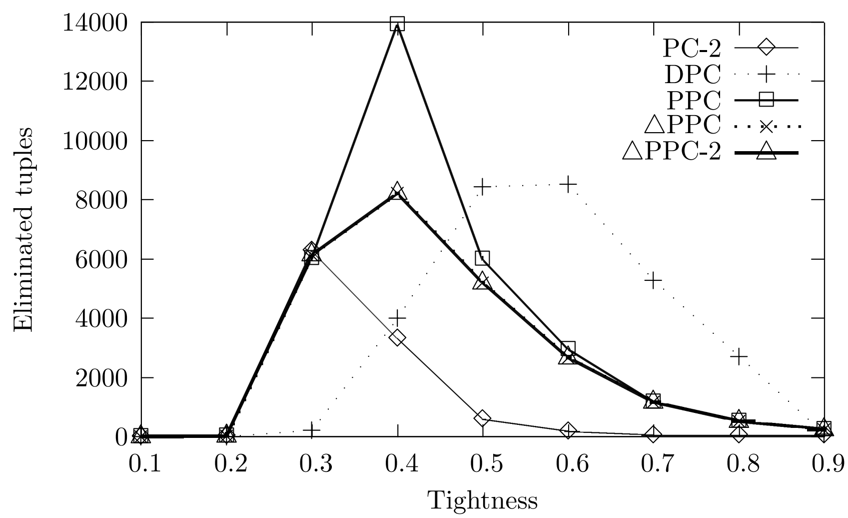
[Bliek and Sam-Haroud(1999)] C. Bliek and D. Sam-Haroud. Path consistency for triangulated constraint graphs. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 456–461, Stockholm, Sweden, 1999.

[Chmeiss and Jegou(1998)] A. Chmeiss and P. Jegou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[Dechter(2003)] R. Dechter. *Constraint Processing.* Morgan Kaufmann, San Francisco, CA, USA, 2003.



(a) Number of calls to REVISE-3.



(b) Number of eliminated tuples.

Figure 5: Average revision results for $n = 32$, $a = 8$, and $p = 0.5$ after completing paths of length two.

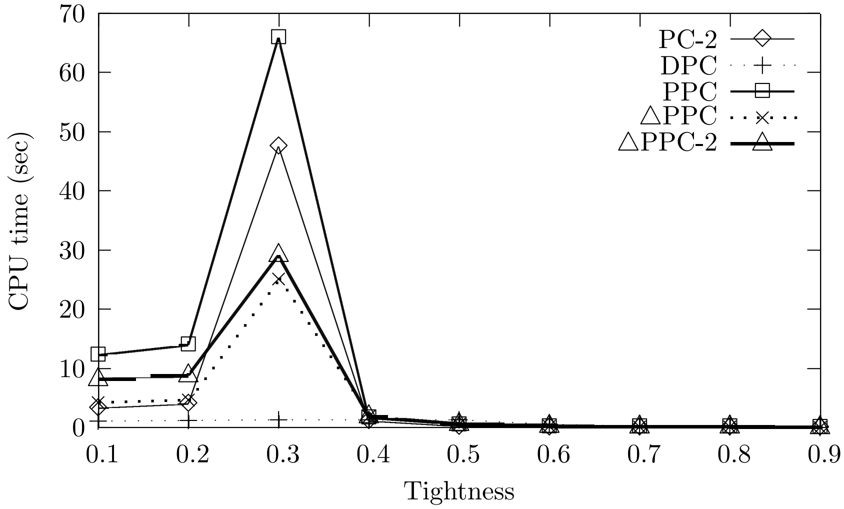


Figure 6: Average CPU time for $n = 32$, $a = 8$, and $p = 0.5$ after completing paths of length two.

[Gogate and Dechter(2004)] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in Artificial Intelligence*, pages 201–208, Banff, AB, Canada, July 2004.

[Golumbic(2004)] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*, chapter 4, pages 98–99. ELSEVIER, Amsterdam, The Netherlands, second edition, 2004.

[Mackworth(1977)] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[XCSP()] XCSP. *XML Representation of Constraint Networks Format XCSP 2.1.* Organising Committee of the Third International Competition of CSP Solvers, Jan. 2008.

[Xu(2003)] L. Xu. Consistency methods for temporal reasoning. Master's thesis, University of Nebraska-Lincoln, Lincoln, NE, USA, May 2003.