CORE
Metadata, citation and similar papers at core.ac.uk
Provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) Department of Computer & Information Science

January 1995

Efficient Path Consistency Algorithms for Constraint Satisfaction Problems

Moninder Singh
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation
Moninder Singh, "Efficient Path Consistency Algorithms for Constraint Satisfaction Problems", . January 1995.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-30.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/199
For more information, please contact repository@pobox.upenn.edu.

Efficient Path Consistency Algorithms for Constraint Satisfaction Problems

Abstract

A large number of problems can be formulated as special cases of the Constraint Satisfaction Problem (CSP). In such a problem, the task specification can be formulated to consist of a set of variables, a domain for each variable and a set of constraints on these variables. A typical task is then to find an instantiation of these variables (to values in their respective domains) such that all the constraints are simultaneously satisfied. Most of the methods used to solve such problems are based on some *backtracking* scheme, which can be very inefficient with exponential run-time complexity for most nontrivial problems. Path consistency algorithms constitute an important class of algorithms used to simplify the search space, either before or during search, by eliminating inconsistent values from the domains of the corresponding variables.

However, the use of these algorithms in real life applications has been limited, mainly, due to their high space complexity. Han and Lee [5] presented a path consistency algorithm, PC-4, with $0(n^3a^3)$ space complexity, which makes it practicable only for small problems. I present a new path consistency algorithm, PC-5, which has an $0(n^3a^2)$ space complexity while retaining the worst-case time complexity of PC-4. Moreover, the new algorithm exhibits a much better average-case time complexity. The new algorithm is based on the idea (due to Bessiere [1]) that, at any time, only a minimal amount of support has to be found and recorded for a labeling to establish its viability; one has to look for a new support only if the current support is eliminated. I also show that PC-5 can be improved further to yield an algorithm, PC5++, with even better average-case performance and the same space complexity. I present experimental results evaluating the performance of these algorithms on various classes of problems. The results show that both PC-5 and PC5++ significantly outperform PC-4, both in terms of space and time, with PC5++ being the better of the two algorithms presented.

Comments

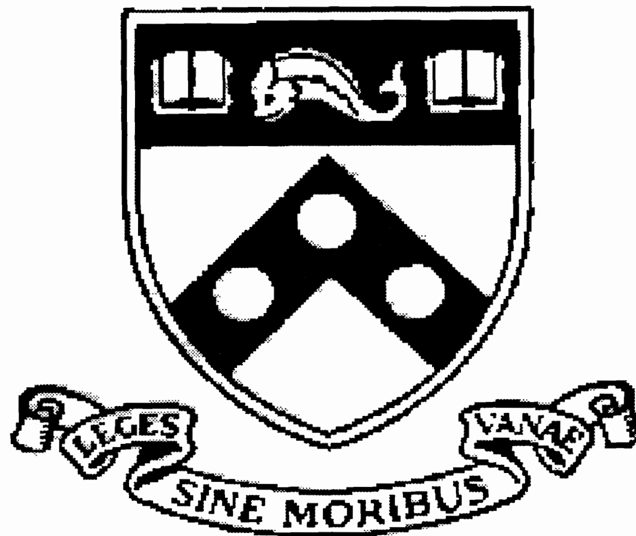
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-30.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/199

Efficient Path Consistency Algorithms for Constraint Satisfaction Problems

MS–CIS–95–30
LINC LAB ???

Moninder Singh



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

August, 1995

Efficient Path Consistency Algorithms for Constraint Satisfaction Problems

Moninder Singh*
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
USA
{*msingh@gradient.cis.upenn.edu*}

*This work was supported by the National Science Foundation under grant # IRI92-10030.

Abstract

A large number of problems can be formulated as special cases of the Constraint Satisfaction Problem (CSP). In such a problem, the task specification can be formulated to consist of a set of variables, a domain for each variable and a set of constraints on these variables. A typical task is then to find an instantiation of these variables (to values in their respective domains) such that all the constraints are simultaneously satisfied. Most of the methods used to solve such problems are based on some *backtracking* scheme, which can be very inefficient with exponential run-time complexity for most nontrivial problems. Path consistency algorithms constitute an important class of algorithms used to simplify the search space, either before or during search, by eliminating inconsistent values from the domains of the corresponding variables.

However, the use of these algorithms in real life applications has been limited, mainly, due to their high space complexity. Han and Lee [5] presented a path consistency algorithm, PC-4, with $O(n^3a^3)$ space complexity, which makes it practicable only for small problems. I present a new path consistency algorithm, PC-5, which has an $O(n^3a^2)$ space complexity while retaining the worst-case time complexity of PC-4. Moreover, the new algorithm exhibits a much better average–case time complexity. The new algorithm is based on the idea (due to Bessiere [1]) that, at any time, only a minimal amount of support has to be found and recorded for a labeling to establish its viability; one has to look for a new support only if the current support is eliminated. I also show that PC-5 can be improved further to yield an algorithm, PC5++, with even better average-case performance and the same space complexity. I present experimental results evaluating the performance of these algorithms on various classes of problems. The results show that both PC-5 and PC5++ significantly outperform PC-4, both in terms of space and time, with PC5++ being the better of the two algorithms presented.

Contents

List of Figures ii

1 Introduction 1

2 Motivation 3

3 The PC-5 algorithm 5

3.1 Description of the Algorithm 5

3.2 Space Complexity 9

3.3 Time Complexity 9

4 The PC5++ algorithm 10

4.1 Description of the Algorithm 10

4.2 Space Complexity 13

4.3 Time Complexity 13

5 Experimental Results 13

5.1 Comparison of PC-5 and PC-4 14

5.2 Comparison of PC-5 and PC5++ 20

6 Conclusion 23

References 24

List of Figures

1 A Counterexample to Chen's PC algorithm 4

2 The PC-5 algorithm: the initialization phase 6

3 The PC-5 algorithm: the nextsupport procedure 7

4 The PC-5 algorithm: the propagation phase 8

5 The PC5++ algorithm: the nextsupport procedure 11

6 The PC5++ algorithm: the initialization phase 12

7 Comparison of PC-4 and PC-5 on the n-queens problem. 15

8 PC-4 and PC-5 on randomly generated CNs with 10 variables where $pu = 0.7$ and $pc = 0.5$. 16

9 PC-4 and PC-5 on randomly generated CNs with 20 variables having 5 possible values where $pc = 0.3$. 17

10 PC-4 and PC-5 on randomly generated CNs with 10 variables having 10 possible values where $pc = 0.7$. 18

11 PC-4 and PC-5 on randomly generated CNs with 15 variables having 5 possible values. 19

12 Comparison of PC-5 and PC5++ on the n-queens problem. 20

13 PC-5 and PC5++ on randomly generated CNs with 10 variables where $pu = 0.7$ and $pc = 0.5$. 21

14 PC-5 and PC5++ on randomly generated CNs with 10 variables having 10 possible values where $pc = 0.7$. 22

15 PC-5 and PC5++ on randomly generated CNs with 20 variables having 5 possible values where $pc = 0.3$. 22

16 PC-5 and PC5++ on randomly generated CNs with 15 variables having 5 possible values. 23

1 Introduction

A large number of problems in AI can be posed as special cases of the Constraint Satisfaction Problem (CSP). In such a problem, the task specification can be formulated to consist of a set of variables, a domain for each variable and a set of constraints on these variables. A typical task is then to find an instantiation of these variables (to values in their respective domains) such that all the constraints are simultaneously satisfied.

My discussion here is restricted to CSPs that can be stated in terms of a finite number of variables, each variable having a finite and discrete domain, and problems in which each constraint[1] is either unary or binary. The latter does not limit the applications of my results since it is possible to convert any CSP with n-ary constraints into an equivalent CSP with unary and binary constraints [6, 11].

Formally, a CSP can be defined as follows ([7, 9]):

$N = \{i, j, \ldots\}$ is the set of nodes, with $|N| = n$,

$D = \{b, c, \ldots\}$ is the set of labels, with $|D| = a$,

$E = \{(i, j) |\ (i, j)$ is an edge in $N \times N\}$, with $|E| = e$,

$D_i = \{b |\ b \in D$ and (i, b) is admissible$\}$,

R_1 is a unary relation, and (i, b) is admissible if $R_1(i, b)$,

R_2 is a binary relation, and $(i, b) - (j, c)$ is admissible if $R_2(i, b, j, c)$.

Most of the methods used to solve such problems are based on some *backtracking* scheme, which can be very inefficient with exponential run-time complexity for most nontrivial problems. One of the reasons for this is that backtracking suffers from "thrashing" [7] i.e. search in different parts of the space keeps failing for the same reasons. Mackworth [7] identified three main causes for thrashing – node inconsistency, arc inconsistency and path inconsistency.

A number of methods have been developed to simplify constraint networks (before or during the search for solutions) by removing values that lead to such inconsistencies.

[1] A constraint is defined over a subset of variables and limits the combinations of values that the variables in this subset can take.

Node inconsistency concerns unary predicates and occurs when the domain of some variable contains one or more values that violate the unary predicate on that variable. Thrashing because of node inconsistency can be eliminated by simply removing those values from the domain of each variable that do not satisfy the unary predicate on that variable [7].

Arc inconsistency involves binary constraints between variables, and occurs when two variables are each instantiated to some value from their respective domains and this instantiation violates the binary constraint between the two variables. In other words, there is at least one value in the domain of one variable that disallows every value in the domain of the second variable. Such a value can obviously never exist in a solution to the CSP, and hence can be safely removed from the domain of the variable concerned. A number of algorithms have been developed for achieving arc consistency in constraint networks including Mackworth's AC-3 algorithm [7], Mohr and Henderson's AC-4 algorithm [9] and Bessiere's AC-6 [1] and AC6++ [2] algorithms.

The third cause for thrashing is path inconsistency. Path consistency implies that any node-value pair of labelings $(i, b) - (j, c)$ that is consistent with the direct constraint between i and j is also allowed by all paths between i and j. To achieve path consistency in a constraint network, it is sufficient to make all length-2 paths consistent because it has been shown by Montanari [10] that path consistency in a complete graph is equivalent to path consistency of all length-2 paths. Note that an incomplete graph can be trivially made complete by adding edges with the always "true" relation [9] between nodes that are not connected.

Once again, a number of algorithms have been designed for achieving path consistency in constraint networks. Mackworth's PC-2 algorithm [7], an improvement over Montanari's PC-1 algorithm [7, 10] has a worst case running time bounded above by $O(n^3a^5)$ [8]. Mohr and Henderson's path consistency algorithm [9], PC-3, uses the same ideas to improve PC-2 as they had used to design AC-4, an improvement over AC-3. However, Han and Lee [5] showed that PC-3 is incorrect, and presented a corrected version, PC-4, with a worst case time and space complexity of $O(n^3a^3)$.

Chen [3] attempted to modify PC-4 in order to improve its average case performance while retaining its worst case complexity. However, I shall show in Section 2 that this algorithm is incorrect.

I discuss the motivation for this research in Section 2, highlighting the problems with PC-4 and pointing out the errors in Chen's path consistency algorithm. In Section 3, I present the PC-5 algorithm and analyze its space and time complexity. In Section 4, I show how PC-5 can be further improved to yield the PC5++ algorithm[2] while I present some experimental results in Section 5.

2 Motivation

PC-4, Han & Lee's corrected version of PC-3, has an $O(n^3a^3)$ space complexity. As noted by Mohr and Henderson [9], the space complexity of the PC-3 algorithm (and hence of PC-4) makes it practicable only for small problems. Hence, it would be useful to reduce the space requirements of the PC-4 algorithm while keeping the same worst-case time complexity. Another problem with the PC-4 algorithm is that it has to consider entire relations in order to construct its data structures. Hence, in many problems where path consistency will not remove many values, the initialization step will be fairly time consuming. Therefore, it is desirable to reduce the complexity of the initialization phase.

Chen [3] attempted to modify the PC-4 algorithm in order to improve its average-case time and space complexity, while retaining its $O(n^3a^3)$ worst-case time and space complexity. However, I have found the following error in Chen's algorithm. This algorithm uses Counter[(i,b,j,c),k] to record *all* supports for a labeling $(i,b)-(j,c)$ in the domain of a node k. If a counter becomes zero, the corresponding labeling is invalid and must be removed from the appropriate relation. However, a labeling $(i,b)-(j,c)$ *cannot* be eliminated from the corresponding relation R_{ij} unless *all* values in the domain of some node k have been tested and found not to support the labeling. The error I have found in Chen's PC algorithm [3, procedure PC, page 347] is that,

[2] While PC-5 is based on AC-6 [1], PC5++ can be regarded as an extension to AC6++ [2].

in lines 26-31, a labeling $(i, b) - (k, d)$ can be eliminated from R_{ik} before all values in D_j have been tested (A similar error follows from lines 32-37). This can be seen through the following example.

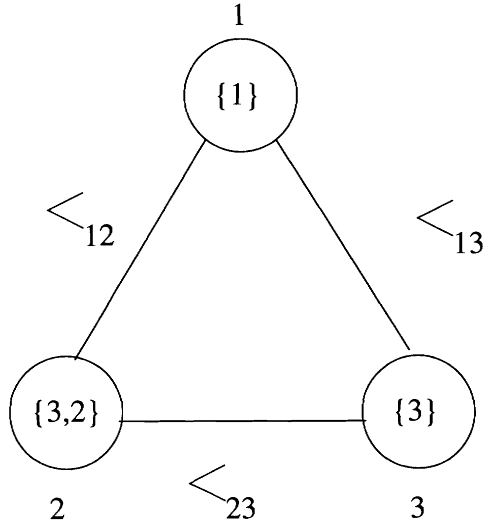


Figure 1: A Counterexample to Chen's PC algorithm

Consider the very simple constraint network of Figure 1. The domains of the three variables and the constraints between them are as shown. During the very first iteration, Chen's PC algorithm does the following:

$$i \leftarrow 1; \quad j \leftarrow 2; \quad k \leftarrow 3$$
$$b \leftarrow 1; \quad c \leftarrow 3; \quad d \leftarrow 3$$

It then checks to see if the assignment $(i, b) - (j, c)$ is supported by (k, d). Since it is not, it sets $R_{ij}(b, c)$ = False and $R_{ji}(c, b)$ = False. While this is correct, the algorithm goes further and also eliminates $R_{13}(1, 3)$ (i.e. $R_{ik}(b, d)$) as well as $R_{23}(3, 3)$ (i.e. $R_{jk}(c, d)$) because it concludes incorrectly that these assignments also have no support. However, the algorithm has not yet checked all the values in D_j (i.e. D_2). The value $2 \in D_2$ is a support for both these assignments - in fact, it is a solution to the problem. Chen's PC algorithm, however, incorrectly discarded the one and only solution to the problem.

The new algorithm, PC-5, that I present here, reduces the space complexity to $O(n^3a^2)$ (as compared to $O(n^3a^3)$ of PC-4) while keeping the worst-case time complexity of PC-4 ($O(n^3a^3)$). Moreover, PC-5 finds only as much evidence as is needed

to support a labeling $(i,b)-(j,c)$ as compared to PC-4 which finds *all* supports. Hence, the average-case time complexity of PC-5 should be substantially better than that of PC-4, especially in problems where path-consistency removes very few values. PC-5 can be further improved to yield another algorithm, PC5++, which has an even better average-case time complexity as compared to PC-5.

The main feature of Mohr and Henderson's AC-4 algorithm [9] was that it made the "support" of a labeling (i,b) evident by storing the relevant support information in an explicit data structure. They had used the same idea in designing PC-3, as did Han and Lee [5] in designing PC-4, the corrected version of PC-3. Bessiere's AC-6 algorithm [1] improves on AC-4 by reducing the space requirements while retaining its (optimal) worst-case time complexity. I use the same ideas as Bessiere to improve upon PC-4.

3 The PC-5 algorithm

3.1 Description of the Algorithm

As pointed out in section 2, PC-4 is based on the notion of "support". As long as a labeling $(i,b)-(j,c)$ (that is consistent with R_{ij}[3]) has supporting values[4] on each of the variables k (adjacent to both i and j in the constraint graph), this labeling is consistent. However, once there is a variable on which no remaining value is consistent with this labeling, it must be eliminated from the relation R_{ij}, i.e. $R_{ij}(b,c) = \text{false}$ and $R_{ji}(c,b) = \text{false}$.

In order to make this support evident, the PC-4 algorithm assigns to each labeling $(i,b)-(j,c)$ a counter$[(i,b,j,c),k]$. This counter records the number of admissible pairs $(i,b)-(k,d)$ that support the binary relation $R_{ij}(b,c)$ where d is any admissible label at node k. Any time $(i,b)-(k,d)$ or $(j,c)-(k,d)$ is removed from the corresponding relation, the support for $(i,b)-(j,c)$ at node k diminishes by 1.

[3] I use $R_{ij}(b,c)$ to represent the binary relation $R_2(i,b,j,c)$ used earlier in the definition of a CSP (page 2).

[4] A value d in D_k is said to support the labeling $(i,b)-(j,c)$ if $R_{ik}(b,d)$ and $R_{jk}(c,d)$ are both valid.

Hence counter$[(i,b,j,c),k]$ and counter$[(j,c,i,b),k]$ are each decremented by 1. If the counters become zero, the labeling $(i,b)-(j,c)$ is removed from R_{ij}. In addition to the counters, PC-4 also maintains sets S_{ibjc} which contain members of the form (k,d), where $R_{ik}(b,d)$ and $R_{ki}(d,b)$ are supported by $R_{ij}(b,c)$. Whenever a labeling $(i,b)-(j,c)$ is eliminated from R_{ij}, this information has to be propagated to the relations $R_{ik}(b,d)$ and $R_{jk}(c,d)$ where (k,d) is a member of S_{ibjc}.

```
M ← 0; S_ibjc = ∅; Waiting_list ← Empty_list;
for i = 1, n − 1 do
    for j = i + 1, n do
        for k = 1, n; k ≠ i, k ≠ j do
            for b ∈ A_i do
                for c ∈ A_j such that R_ij(b, c) = true do
                 begin
                     d ← 1; nextsupport(i, b, j, c, k, d, emptysupport);
                     if emptysupport then
                      begin
                          M[i, b, j, c] = 1; M[j, c, i, b] = 1;
                          R_ij(b, c) = false; R_ji(c, b) = false;
                          append(Waiting_list, (i, b, j, c))
                      end
                     else
                      begin
                          append(S_ibkd, (j, c));
                          append(S_jckd, (i, b))
                      end
                 end
```

Figure 2: The PC-5 algorithm: the initialization phase

As noted by Bessiere [1], computing the number of supports for each labeling $(i,b)-(j,c)$ and recording all of them implies an average-case time complexity and space complexity both increasing with the number of allowed pairs in the relations, since the number of supports is proportional to the number of pairs allowed in the concerned relations.

PC-5 rectifies this problem by determining and storing only one support for each labeling. In the initialization phase (Figure 2), the algorithm determines one support (the first one) for each labeling $(i,b)-(j,c)$ in the domain of a third node k (k is adjacent to both i and j in the constraint graph). If no such support is found, the

```
procedure nextsupport(i, b, j, c, k, var d, var emptysupport)
 begin
    if d ≤ last(D_k) then
     begin
        emptysupport ← false
        while ((M[i,b,k,d] or M[j,c,k,d]) and (d ≤ last(D_k))) do
            d ← d + 1;
        if d ≤ last(D_k) then
         begin
            while not ( R_ik(b,d) and R_jk(c,d)) and not emptysupport do
                if d < last(D_k) then
                    d ← next(d, D_k)
                else
                    emptysupport ← true
         end
        else
         emptysupport ← true
     end
    else
     emptysupport ← true
 end
```

Figure 3: The PC-5 algorithm: the nextsupport procedure

assignment $(i,b)-(j,c)$ is invalid. So this assignment is eliminated from the relations R_{ij} and R_{ji}. Moreover, this labeling is added to the waiting list to be propagated. If, however, (k,d) is found as the first support for this labeling on R_{ik} and R_{jk}, then (j,c) is appended to S_{ibkd} (signifying that $R_{ij}(bc)$ is supported by $R_{ik}(b,d)$). Similarly, (i,b) is appended to $S_{jc}(k,d)$. If then, at a later stage, a labeling $(i,b)-(k,d)$ is removed from R_{ik}, the algorithm tries to determine the next support for $(i,b)-(j,c)$ in k as well as for $(j,c)-(k,d)$ in i. The procedure *nextsupport* (Figure 3) is used to find the first as well as the next support of each labeling $(i,b)-(j,c)$ in the domain of k. This procedure is based on the nextsupport procedure used in AC-6 [1].

During the propagation phase (Figure 4), information about the invalid labelings (recorded in the waiting_list) has to be propagated to all the nodes. If (k,d,l,e) is removed from the waiting list, it means that the labeling $(k,d)-(l,e)$ is not valid;

```
while Waiting_list ≠ Empty_list do
 begin
    choose (k, d, l, e) from the Waiting_list and delete it;
    for (j, c) ∈ S_kdle do
     begin
        remove (j, c) from S_kdle and (k, d) from S_jcle;
        if M[k, d, j, c] = 0 then
         begin
            next ← e; nextsupport(k, d, j, c, l, next, emptysupport);
            if emptysupport then
             begin
                M[k, d, j, c] = 1; M[j, c, k, d] = 1;
                append(Waiting_list, (k, d, j, c));
                R_kj(d, c) = false; R_jk(c, d) = false
             end
            else
             begin
                append(S_kdlnext, (j, c));
                append(S_jclnext, (k, d))
             end
         end
     end
    for (j, c) ∈ S_lekd do
     begin
        remove (j, c) from S_lekd and (l, e) from S_jckd;
        if M[l, e, j, c] = 0 then
         begin
            next ← d; nextsupport(l, e, j, c, k, next, emptysupport);
            if emptysupport then
             begin
                M[l, e, j, c] = 1; M[j, c, l, e] = 1;
                append(Waiting_list, (l, e, j, c));
                R_lj(e, c) = false; R_jl(c, e) = false
             end
            else
             begin
                append(S_leknext, (j, c));
                append(S_jcknext, (l, e))
             end
         end
     end
 end
```

Figure 4: The PC-5 algorithm: the propagation phase

so all relations supported by it (members of S_{kdle}) are also invalid and the algorithm must find the next support for each one of these relations. So for each (j, c) in S_{kdle}, the algorithm tries to find the next support for the labeling $(k, d) - (j, c)$ in D_l as well as $(l, e) - (j, c)$ in D_k. If a support is found it is recorded in the relevant S set; otherwise the labeling is eliminated from the corresponding relations and is added to the waiting_list to be propagated to the other nodes.

Using PC-5 requires a total ordering on all domains; however, as pointed out by Bessiere [1], this is not a restriction since any implementation imposes a total ordering on the domains.

3.2 Space Complexity

The matrix M requires $O(n^2a^2)$ space where a is the size of the largest domain and n is the number of variables. Moreover, the sum of the size of the different sets S_{ibjc} is bounded by:

$$n \times \sum_{(i,j) \in N \times N} |A_i| \times |A_j| \leq n^3a^2$$

This is because each set S_{ibjc} can be, at most, of size n since it contains at most one support for the labeling $(i, b) - (j, c)$ in each node. Hence the space complexity of the entire algorithm is $O(n^3a^2)$ as compared to the $O(n^3a^3)$ space complexity of PC-4. Moreover, PC-5 does not use the *counters* used in PC-4 (which are very expensive requiring an additional $O(n^3a^2)$ space).

3.3 Time Complexity

The time complexity analysis of PC-5 is similar to that of PC-4. In the *initialization* phase, the innermost **for** loop will be executed on the order of n^3a^2 since $|D_i|$ and $|D_j|$ are both of size $O(a)$. Moreover, the inner loop requires a call to the procedure *nextsupport* which computes a support for a labeling, say $(i, b) - (j, c)$, in the domain of a variable, say k, starting at the current value. Hence, for each such assignment

(of the form $(i,b)-(j,c)$), each value in D_k will be checked at most once. So the worst-case time complexity of the *initialization* phase will be $O(n^3a^3)$.

In the *propagation* phase, the **while** loop is executed at most n^2a^2 times since there are at most n^2a^2 sets of type S_{ibjc}. Moreover, each of the **for** loops is bounded by the size of S_{kdle} which is of the order n. Moreover, each **for** loop requires a call to the procedure *nextsupport* which, as shown above, requires $O(a)$ time. Hence, the worst-case time complexity of the *propagation* phase is $O(n^3a^3)$.

Hence, PC-5 has the same worst-case time complexity as PC-4. Moreover, the average-case time complexity of PC-5 is substantially better than that of PC-4 since it stops processing of a value assignment to an edge just when it has proof that it is viable (i.e. the first support).

4 The PC5++ algorithm

4.1 Description of the Algorithm

It is possible to improve the average-case time complexity of PC-5 by increasing the space requirements slightly. The worst-case time and space complexities still remain $O(n^3a^3)$ and $O(n^3a^2)$ respectively. The improvement comes from the observation that each time PC-5 determines a support d in D_k for the labeling $(i,b)-(j,c)$, it in fact also finds a support (b in D_i) for $(j,c)-(k,d)$ as well as a support (c in D_j) for $(i,b)-(k,d)$. Hence, by recording the supports at this time, it is possible to avoid duplicating the effort in determining these supports at a later time. The problem with this approach is that now the algorithm has to keep track of the position from which it started checking for the first support. PC-5 starts looking for a support from the very first value in the domain; hence, it looks over the entire domain and if it reaches the last element in the domain without finding a support, it safely concludes that there is no support for the labeling under consideration in that domain. However, if we make the above mentioned modification, then when the support d in D_k is found for the labeling $(i,b)-(j,c)$, we also store the fact that b in D_i supports $(j,c)-(k,d)$ and c in D_j supports $(i,b)-(k,d)$. However, the labels preceding b in D_i have *not* been checked

```
procedure nextsupport(i, b, j, c, k, d, emptysupport)
 begin
    if d ≤ last(D_k) then
     begin
        emptysupport ← false
        while ((M[i, b, k, d] or M[j, c, k, d]) and (d ≤ last(D_k))) do
            d ← d + 1;
        if d ≤ last(D_k) then
         begin
            while not ( R_ik(b, d) and R_jk(c, d)) and not emptysupport do
                if d < last(D_k) then
                    d ← next(d, D_k)
                else
                    emptysupport ← true
         end
        else
         emptysupport ← true
     end
    else
     emptysupport ← true
 end
```

Figure 5: The PC5++ algorithm: the nextsupport procedure

to see if they support the labeling $(j, c) - (k, d)$. Similarly, the labels preceding c in D_j have not been checked to see whether any one supports the labeling $(i, b) - (k, d)$. This problem can be taken care of by using a data structure Tag[(i,b,j,c),k] which records the first position in D_k where the algorithm started looking for the support of a labeling $(i, b) - (j, c)$. The *nextsupport* procedure can be easily modified to take this fact into account. Instead of stopping after considering the last value in the domain, the procedure continues examining the values from the first value in the domain, and stops only when all values have been checked once (it reaches the value from where it started from i.e. Tag[(i,b,j,c),k]).

Figure 6 shows the initialization phase of the PC5++ algorithm. The propagation phase is the same as for PC-5.

```
M ← 0; S_ibjc = ∅; Waiting_list ← Empty_list; Tag ← 0;
for i = 1, n − 1 do
    for j = i + 1, n do
        for k = 1, n, k ≠ i, j do
            for b ∈ A_i do
                for c ∈ A_j such that R_ij(b, c) = true do
                 begin
                     if Tag[(i, b, j, c),k] = 0 then
                      begin
                         Tag[(i, b, j, c),k] = 1; Tag[(j, c, i, b),k] = 1;
                         d ← 1; nextsupport(i, b, j, c, k, d, emptysupport);
                         if emptysupport then
                          begin
                             M[i, b, j, c] = 1; M[j, c, i, b] = 1;
                             R_ij(b, c) = false; R_ji(c, b) = false;
                             append(Waiting_list, (i, b, j, c))
                          end
                         else
                          begin
                             append(S_ibkd, (j, c));
                             append(S_jckd, (i, b))
                             if Tag[(i, b, k, d),j] = 0 then
                              begin
                                 append(S_ibjc, (k, d));
                                 append(S_kdjc, (i, b));
                                 Tag[(i, b, k, d),j] = c; Tag[(k, d, i, b),j] = c;
                              end
                             if Tag[(j, c, k, d),i] = 0 then
                              begin
                                 append(S_kdib, (j, c));
                                 append(S_jcib, (k, d));
                                 Tag[(j, c, k, d),i] = b; Tag[(k, d, j, c),i] = b;
                              end
                          end
                      end
                 end
```

Figure 6: The PC5++ algorithm: the initialization phase

4.2 Space Complexity

PC5++ uses the same data structures as PC-5 with the exception of the Tag data structure. Since the algorithm has to maintain tags for each assignment $(i, b) - (j, c)$ and node k, the total storage required for the tags is $O(n^3a^2)$. Since the space requirement of the other structures is also $O(n^3a^2)$, the overall space complexity of the algorithm still remains $O(n^3a^2)$.

4.3 Time Complexity

The propagation phase is the same as for PC-5, whereas the extra steps in the initialization phase can all be performed in constant time. The procedure *nextsupport* still examines each value in a domain at most once. Hence, its complexity is still $O(a)$, and so the worst-case time complexity of PC5++ is still $O(n^3a^3)$.

5 Experimental Results

In order to compare the performance of PC-5 and PC5++ to that of PC-4, I carried out a series of experiments on a large spectrum of problems (described in the next section). For each problem, I counted the number of constraint checks (to compare the time complexity) and the number of supports recorded, i.e. size of the sets S_{ibjc} (to compare the space complexity). A constraint check[5] is performed each time the algorithm checks an assignment $(i, b) - -(j, c)$ for consistency with respect to the constraint R_{ij} (i.e. whether $R_{ij} =$ true). Although the performance of the three algorithms was measured on the same sets of problems, I present the results separately in order to emphasize the improvement of PC5++ over PC-5 (which would not always be apparent if all results were shown on the same figure).

[5] If the algorithms had been tested on problems where the constraints involved more than two variables, the time for performing a constraint check could not be regarded as constant [4].

5.1 Comparison of PC-5 and PC-4

The first experiment was done on the *zebra* problem [4, 1] which has similarities to some problems encountered in real life. I used the same encoding of the problem as used by Dechter [4]. Table 1 shows the results of this experiment. PC-5 outperformed PC-4 significantly both in terms of the number of constraint checks as well as the number of supports recorded. I also tested the various algorithms on the n-queens

	No. of Constraint checks	No. of Supports recorded
PC-4	1,682,560	1,326,250
PC-5	551,373	333,118
PC5++	412,537	340,300

Table 1: Comparison of PC-4 with PC-5 and PC5++ on the *zebra* problem

problem which was encoded as a constraint network by representing each column by a variable whose values are the rows. The constraint network is complete with a very weak constraint present between each pair of variables. A constraint R_{ij} between variables i and j specifies the positions (rows) in which two queens can be placed in columns i and j. As can be seen from Figure 7, both the space and time complexity of PC-4 deteriorates as the number of queens increases; PC-5 performs markedly better.

I also tested the algorithms on a variety of randomly generated problems, with different values of

n, the number of variables
a, the number of values per variable
pc, the probability that a constraint R_{ij} exists between variables i and j
pu, the probability that a pair (a, b) belongs to a relation R_{ij}

If two nodes did not have a constraint between them, the constraint with the always "true" relation was introduced between them. I generated twenty instances of problems for each set of parameter values, and averaged the results so as to get a more representative picture of each class. Figures 8 – 11 show the results of these experiments. In Figures 9 – 11, a broken vertical line shows the borderline between problems where wipe-out is generally produced (located on the left of the line) and problems where path-consistency is produced (on the right of the line).

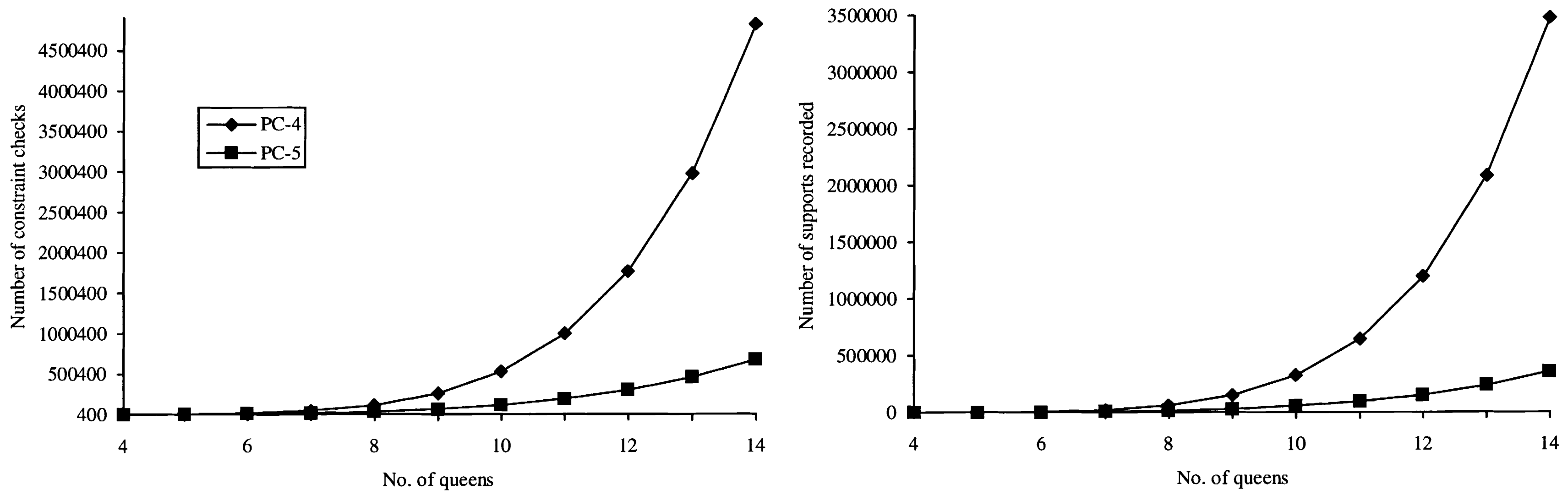


Figure 7: Comparison of PC-4 and PC-5 on the n-queens problem.

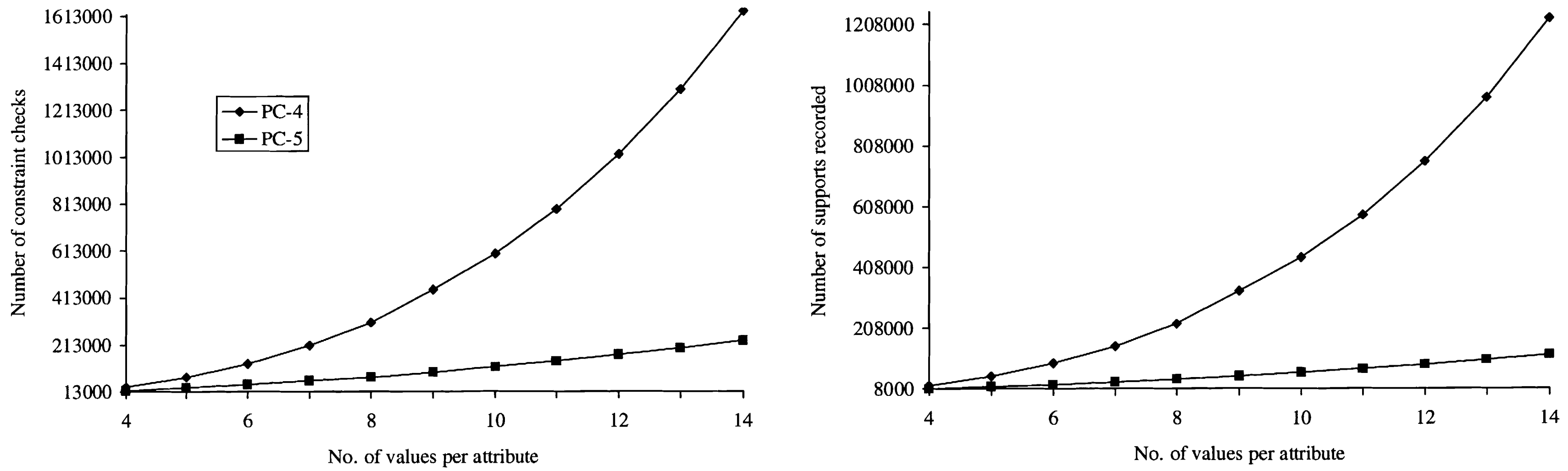
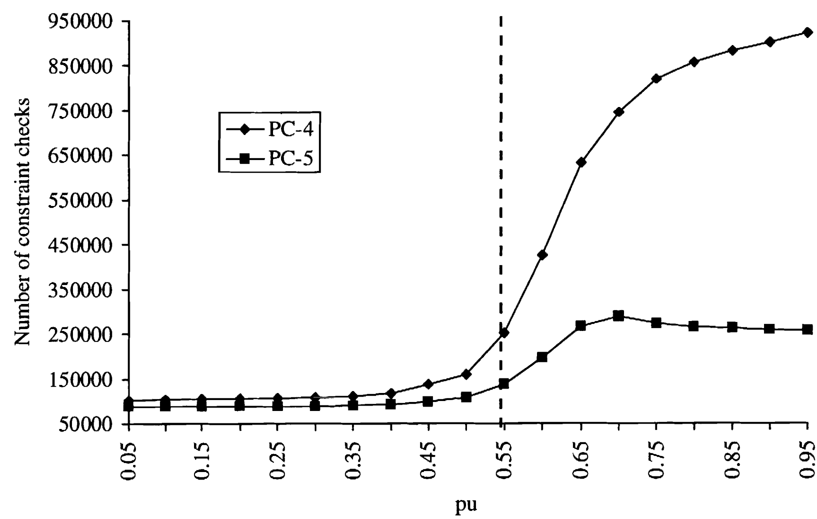


Figure 8: PC-4 and PC-5 on randomly generated CNs with 10 variables where $pu = 0.7$ and $pc = 0.5$.



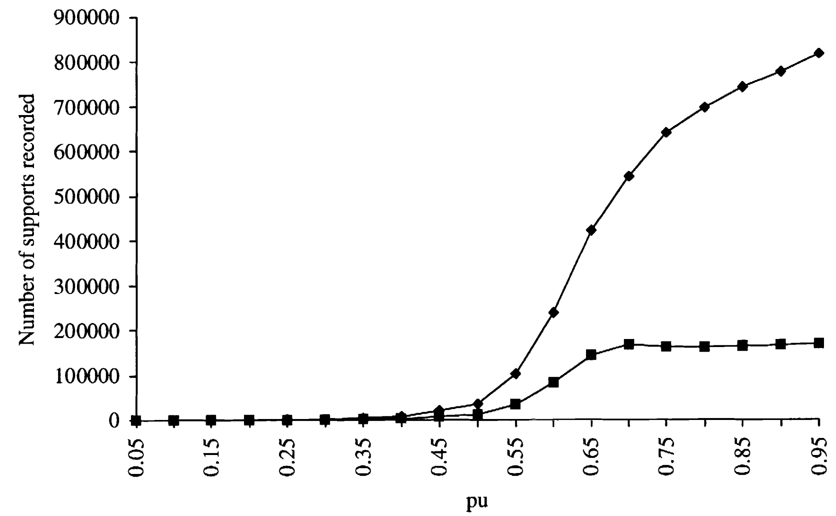
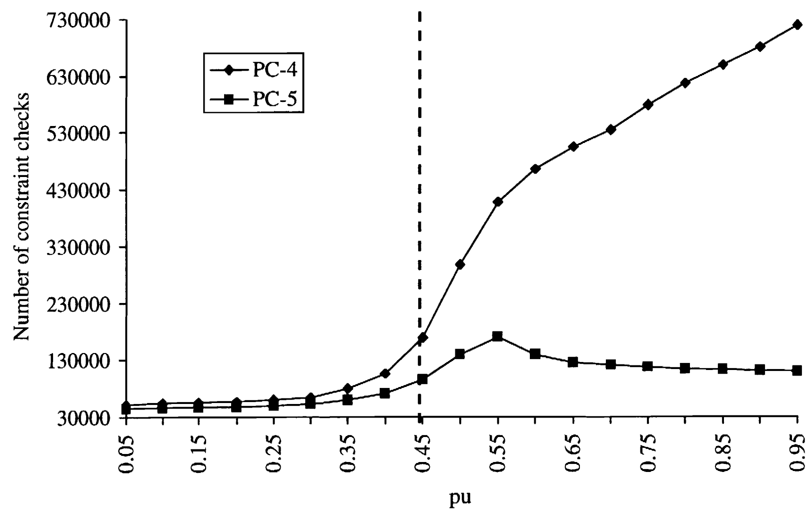


Figure 9: PC-4 and PC-5 on randomly generated CNs with 20 variables having 5 possible values where $pc = 0.3$.



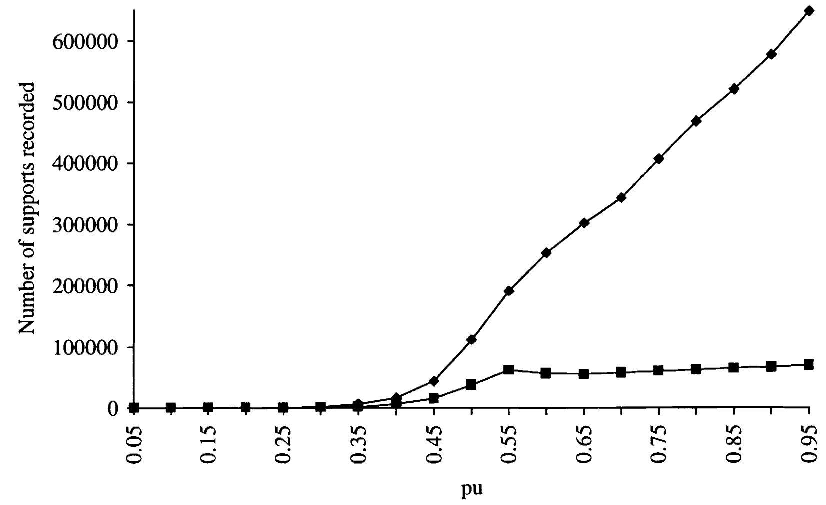
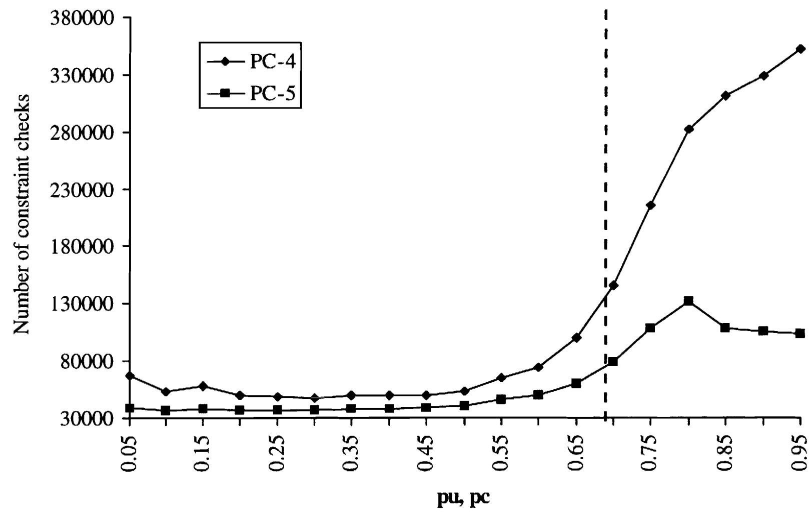


Figure 10: PC-4 and PC-5 on randomly generated CNs with 10 variables having 10 possible values where $pc = 0.7$.



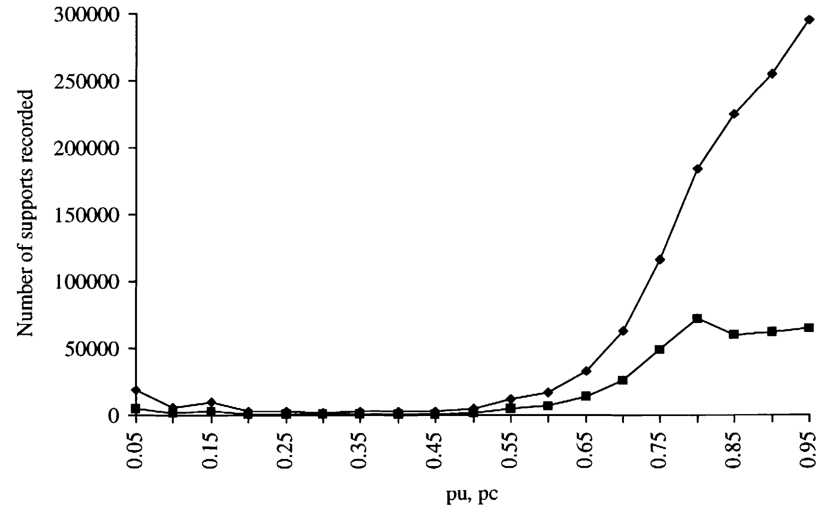


Figure 11: PC-4 and PC-5 on randomly generated CNs with 15 variables having 5 possible values.

The space requirement of PC-4 increases very rapidly with increasing *pu* (i.e. constraints become weaker) as seen in Figures 9-11 as well as with increasing number of values per attribute as seen in Figure 8. The space requirements of PC-5 (as expected from the algorithm's complexity) are significantly lower.

Both PC-4 and PC-5 perform roughly the same number of constraint checks when the constraints are strong (*pu* is small) and wipe-out is produced (on the left of the broken line). However, at higher values of *pu* when path consistency is produced (right of the broken line), the performance of PC-4 rapidly deteriorates whereas PC-5 performs substantially better.

Similarly, PC-5 performs significantly fewer constraint checks than PC-4 as the number of values per attribute increase (Figure 8).

5.2 Comparison of PC-5 and PC5++

As can be seen from Table 1, PC5++ easily outperformed both PC-4 and PC-5 on the zebra problem. A similar improvement was observed for the n-queens problems (Figure 12) where PC5++ performed upto 38% fewer constraint checks than PC-5.

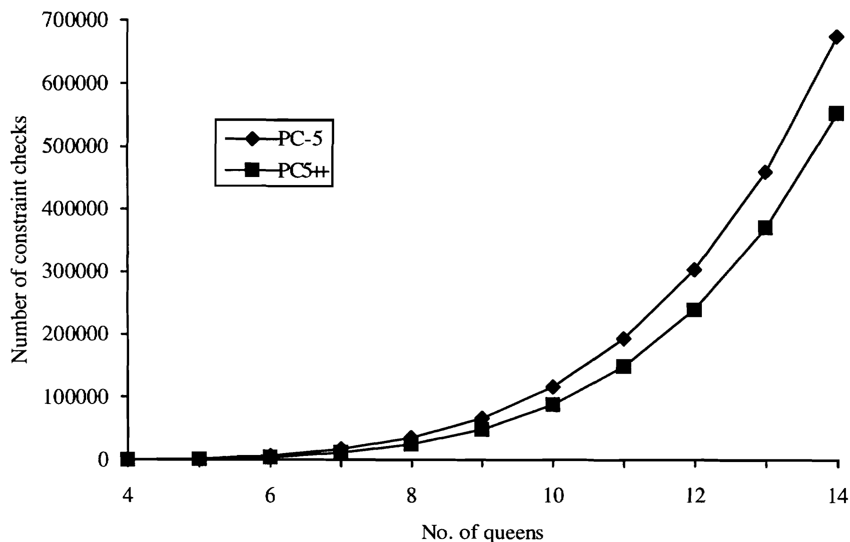


Figure 12: Comparison of PC-5 and PC5++ on the n-queens problem.

As can be seen from Figures 13–16, PC5++ also performed *substantially* better

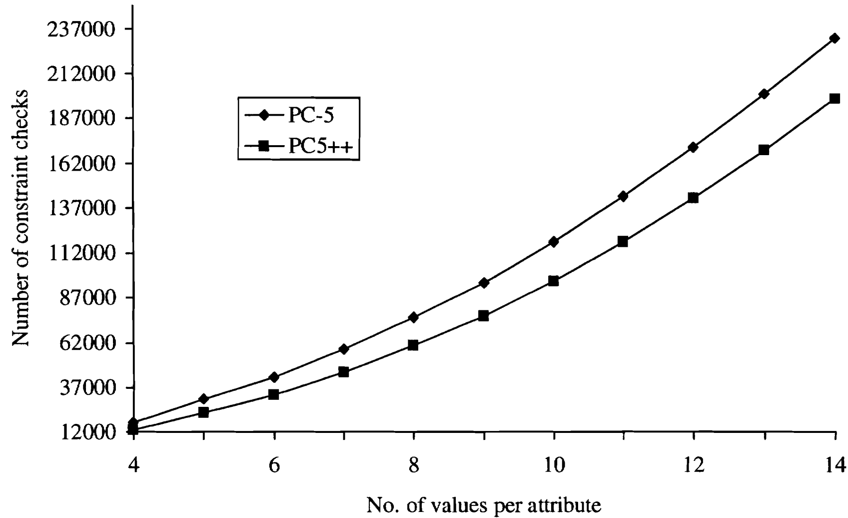


Figure 13: PC-5 and PC5++ on randomly generated CNs with 10 variables where $pu = 0.7$ and $pc = 0.5$.

than PC-5 on all the problems tested. PC5++ reduced the number of constraint checks performed by PC-5 by upto 26% in Figure 13, upto 23% in Figures 14 and 15 and upto 27% in Figure 16. The space requirements were almost the same for all problems.

Once again, the broken vertical line in Figures 14–16 shows the borderline between problems where wipe-out is generally produced and problems where path consistency is produced. I also checked the statistical significance of the difference between PC-5 and PC5++ by performing a paired t-test at a 99% confidence level. For Figure 13, PC5++ was found to be always significantly better than PC-5. For Figures 14–16, there was no statistical difference to the left of the broken vertical line (i.e. when wipe-out is produced). In each case, PC5++ performed significantly lesser number of constraint checks than PC-5 to the right of the broken vertical line. Thse results are as one would expect – when the problem has no solution, all algorithms will perform virtually the same amount of work, effectively checking all values in all constraints for consistency; however, for problems where a solution exists, then PC5++ makes the network consistent much faster than does PC-5.

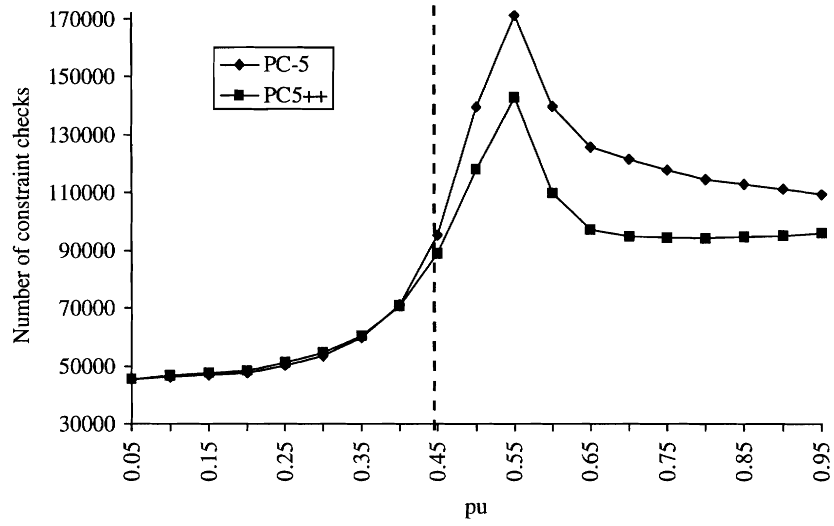


Figure 14: PC-5 and PC5++ on randomly generated CNs with 10 variables having 10 possible values where $pc = 0.7$.

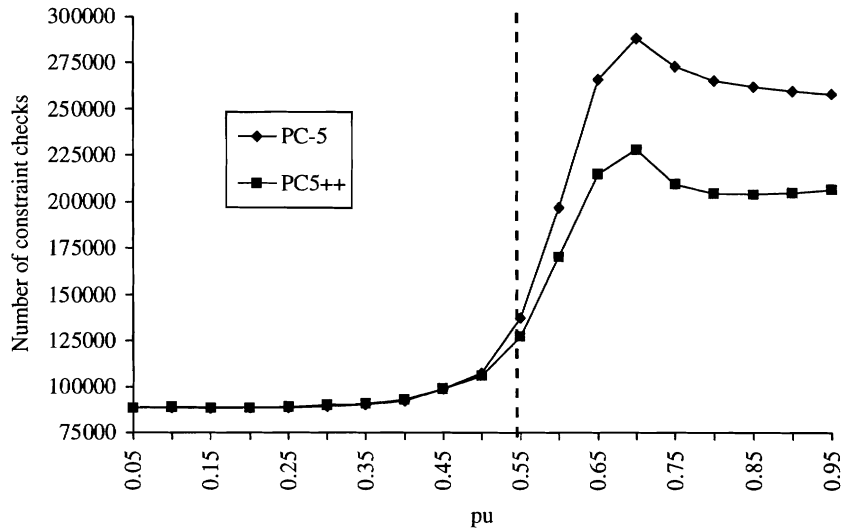


Figure 15: PC-5 and PC5++ on randomly generated CNs with 20 variables having 5 possible values where $pc = 0.3$.

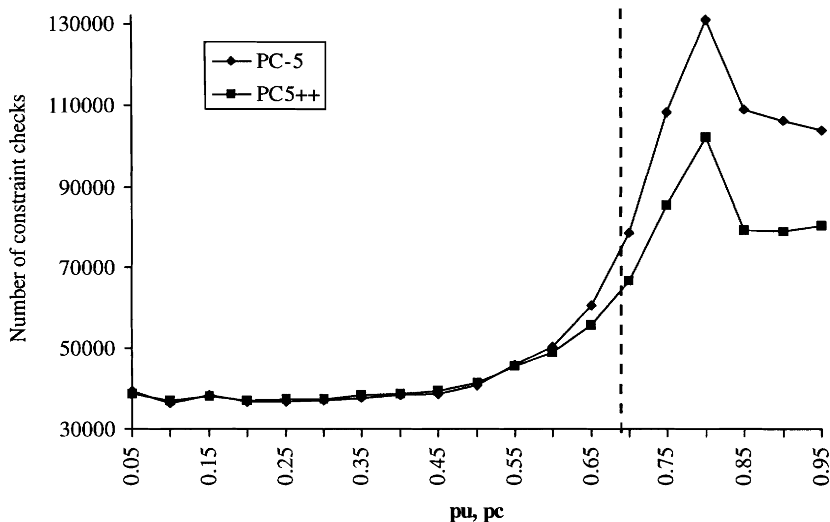


Figure 16: PC-5 and PC5++ on randomly generated CNs with 15 variables having 5 possible values.

6 Conclusion

I have presented a new algorithm, PC-5, for achieving path consistency in constraint networks. The main improvement of PC-5 over previous path consistency algorithms is its reduced space complexity ($O(n^3a^2)$). Moreover, it retains the $O(n^3a^3)$ worst-case time complexity of PC-4 while improving its average-case time complexity, especially on networks with weak constraints. I further show that PC-5 can be modified to yield another algorithm, PC5++, which retains the $O(n^3a^2)$ space complexity but exhibits even better average case performance. I also present experimental results which show that both PC-5 and PC5++ vastly outperform PC-4 on all the problems tested with PC5++ performing better, as expected, than PC-5. I must emphasize, though, that I do not claim that the algorithms presented have the best possible time complexity as our main aim has been to reduce the space complexity.

It may be possible to improve the performance of the algorithms even further. Note that a value b in D_i, which initially had support in D_j (where j is a neighboring node in the constraint graph) may lose all that support because every pair $(b, c), c \in D_j$

has been eliminated from R_{ij} due to the absence of support for each such labeling $(i,b) - (j,c)$ at one or more of the remaining nodes. As such, it is now possible to remove b from D_i, thereby preventing further consideration of this value. By keeping track of these changes, it may be possible to increase the efficiency of the algorithm. However, this will increase the space requirements which may not be worth the savings achieved.

Acknowledgements

The author would like to thank Prof. Bonnie Webber for her helpful comments and suggestions for improving the paper.

References

[1] C. Bessiere, Arc-consistency and arc-consistency again, *Artif. Intell.* **65** (1) (1994) 179-190.

[2] C. Bessiere and J. Regin, An arc-consistency algorithm optimal in the number of constraint checks, in: *Proceedings 6th IEEE Int. Conf. on Tools with AI* (1994) 397-403.

[3] Y. Chen, Improving Han and Lee's path consistency algorithm, in: *Proceedings 3rd IEEE Int. Conf. on Tools for AI* (1991) 346-350.

[4] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* **41** (1990) 273–312.

[5] C. Han and C. Lee, Comments on Mohr and Henderson's path consistency algorithm, *Artif. Intell.* **36** (1988) 125-130.

[6] V. Kumar, Algorithms for constraint-satisfaction problems: a survey, *AI Magazine* **13** (1992) 32-44.

[7] A.K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1) (1977) 99-118.

[8] A.K. Mackworth and E.C. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artif. Intell.* **25** (1985) 65-74.

[9] R. Mohr and T.C. Henderson, Arc and path consistency revisited, *Artif. Intell.* **28** (1986) 225-233.

[10] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, *Inf. Sci.* **7** (1974) 95-132.

[11] F. Rossi, C. Petrie and V. Dhar, On the equivalence of constraint satisfaction problems, Tech. rep. ACT-AI-222-89, MCC Corp., Austin, Texas, (as referenced by Kumar [6]).