Understanding and Improving the MAC Algorithm

Daniel Sabin and Eugene C. Freuder

Department of Computer Science,
University of New Hampshire,
Durham, NH, 03824-2604,
USA

Abstract. Constraint satisfaction problems have wide application in artificial intelligence. They involve finding values for problem variables where the values must be consistent in that they satisfy restrictions on which combinations of values are allowed. Recent research on finite domain constraint satisfaction problems suggest that Maintaining Arc Consistency (MAC) is the most efficient general CSP algorithm for solving large and hard problems. In the first part of this paper we explain why maintaining full, as opposed to limited, arc consistency during search can greatly reduce the search effort. Based on this explanation, in the second part of the paper we show how to modify MAC in order to make it even more efficient. Experimental results prove that the gain in efficiency can be quite important.

1 Introduction

Constraint satisfaction problems (*CSPs*) involve finding values for problem variables subject to constraints that are restrictions on which combinations of values are allowed. They have many applications in artificial intelligence, ranging from design to diagnosis, natural language understanding to machine vision.

The complete description of a CSP is given by specifying the set of variables, the set of potential values for each variable, called its domain, and the set of constraints. The basic solution method is backtrack search. A solution is an assignment of one value to each variable such that all the constraints are simultaneously satisfied.

In order to improve efficiency, very often search is interleaved with consistency inference (constraint propagation) which is used to prune values during search. The basic pruning technique involves establishing or restoring some form of *arc consistency*, pruning values that become inconsistent after making search choices. Recent research on finite domain CSPs suggests that *Maintaining Arc Consistency (MAC)* (Sabin & Freuder 1994) is the most efficient general CSP algorithm (Grant & Smith 1995) (Bessiere & Regin 1996). Using implementations based on AC-7 or AC-Inference (Bessiere, Freuder, & Regin 1995) (Regin 1995), which have a very good space and worst case running time complexity, and a new dynamic variable ordering heuristic (Bessiere & Regin 1996), MAC can solve problems which are both large and hard.

The enhanced look ahead allows MAC to make a much more informed choice in selecting the next variable and/or value, thus avoiding costly backtracks later on during search. However, additional search savings will be offset by the additional costs if proper care is not taken during the implementation. There are two sources of overhead in implementing MAC:

- the cost of restoring arc consistency after a decision has been made during search (either to instantiate a variable or to delete a value)
- the cost of restoring the problem in the previous state in case the current instantiation leads to failure.

Specifically, most of the effort is spent in deleting inconsistent values, during the propagation phase, and adding them back to the domains, after backtracking. Accordingly, this paper proposes two ways in which we can lower these costs:

- *Instantiate less.* In the context of maintaining full arc consistency, the search algorithm can focus on instantiating only a subset of the original set of variables, yielding a partial solution which can be extended, in a backtrack free manner, to a complete solution. Depending on the problem's density, the size of this subset, and thus the effort to find a solution, can be quite small.
- *Propagate less.* Instead of maintaining the constraint network in an arc consistent state, we propose to maintain an equivalent state, less expensive to achieve because it requires less propagation, which is:
  - only partially arc consistent, but
  - guaranteed to extend to a fully arc consistency state.

The rest of the paper is organized as follows. Section 2 presents examples and observations that motivate the main ideas behind the two improvements mentioned above. Section 3 discusses related work. Section 4 describes more formally the methods we propose for improving MAC. The last section presents experimental evidence that proves that the gain in efficiency can be quite large.

2 Example

(Grant & Smith 1995) presents a major study of the performance of MAC[1] over a large range of problem sizes and topologies. The results demonstrate that the size of the search trees is much smaller for MAC than for FC and that MAC produces backtrack-free searches over a considerably larger number of problems across the entire range of density/tightness values commonly used to characterize random problem space.

If we expected MAC to do better than FC, due to its enhanced look-ahead capabilities, our own experiments showed an unexpected result: that on problems with low and medium constraint densities (up to $0.5 - 0.6$) a static variable

[1] The authors describe a "weak" form of MAC; we believe that the results would have been even stronger if the experiments had been done with the MAC we describe in Figure 4.

ordering heuristic, instantiating variables in decreasing order of their degree in the constraint graph, is in general more effective in the context of MAC than the popular dynamic variable ordering based on minimal domain size. In the majority of cases the gain in efficiency was due to a lower number of backtracks, very large regions of the search space being backtrack-free.

Trying to understand how can a static variable ordering be better than a dynamic one is what lead us to the ideas we will present next on a couple of examples. We restrict our attention here to *binary* CSPs, where the constraints involve two variables. One way of representing a binary CSP is in the form of a constraint graph. Nodes in the graph are the CSP variables and the constraints form the arcs.

Let us now consider the example represented by the constraint graph in Figure 1a, and see what happens during the search for a solution. For the sake of simplicity, assume that all constraints are *not-equal* and all domains are equal to the set $\{r, g, b\}$.

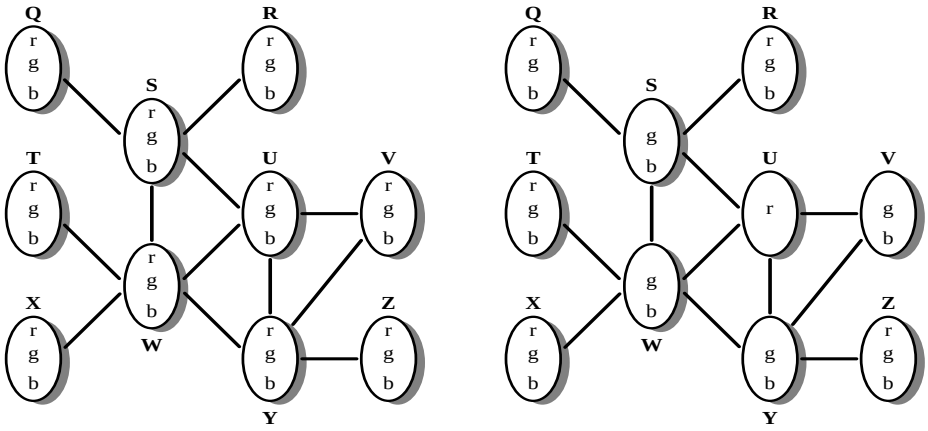

Figure 1. Sample constraint net

We are ready now to explain what we mean by *instantiate less*. Suppose that MAC will choose for some obscure (for now) reason, U as the first variable to be instantiated. After selecting value r, the algorithm will eliminate all the other values in the domain of U and will propagate the effects of these removals, restoring arc consistency, as shown in Figure 1b.

At this point the reader can verify that no matter which variable is next instantiated, and no matter which value is selected for the instantiation, MAC will find a complete solution without having to backtrack. Furthermore, we claim that if we had been interested only in finding out whether the problem is satisfiable or not, the algorithm could have stopped after having successfully instantiated variable U and have returned an affirmative answer. Why? Take a look at Figure 2a, which presents the state of the problem after instantiating variable U.

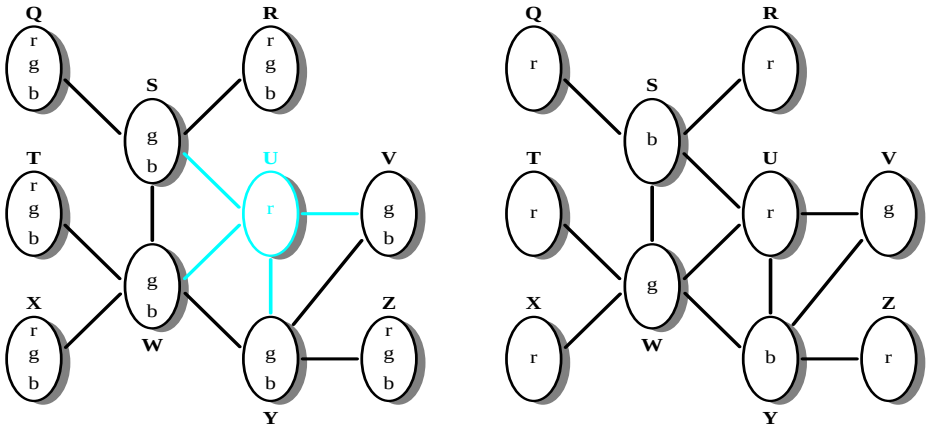

Figure 2. The constraint net after instantiating variable U (a) and a complete solution (b)

Intuitively, since r is the only value left in the domain of U and it supports (is consistent with) all the values remaining in the domains of neighbor variables, it will itself always have a support as long as these domains are not empty. U thus becomes irrelevant for the search process trying to extend this partial solution, and we can temporarily "eliminate" from further consideration both U and all constraints involving it.

If we ignore the grayed part of the constraint net, the constraint graph becomes a tree. This, plus the fact that every value in the domain of any variable is supported by at least one value at each neighbor (the network is arc consistent), implies that the problem is globally consistent and makes it possible to find a complete solution in a backtrack-free manner, for example the one in Figure 2b. In fact, for this reason, MAC is able to find all the solutions involving $U = r$ without having to backtrack. But what makes U so special ? If we look again to the graph in Figure 2a, we see that

- all the cycles in the graph have one node in common, the one corresponding to variable U, and
- by eliminating this node and all the edges connected to it we obtain an acyclic graph.

A set of nodes which "cut" all the cycles in a graph is called *cycle cutset*. In our case, the set $\mathcal{C} = \{ U \}$ represents a minimal cycle cutset for the graph in Figure 2a. It is obvious that the graph obtained from the original one by eliminating the nodes in any cycle cutset and the related edges is acyclic.

The observations made on the graph in our example are directly supported by research on discrete domain CSPs (Dechter & Pearl 1988) (Freuder 1982), and are similar to the results presented in (Hyvonen 1992) in the form of the following two theorems:

(1) An acyclic constraint net is globally consistent iff it is locally consistent.
(2) If the variables of any cutset of a constraint net S are singleton-valued, then S is globally consistent iff it is locally consistent.

If after all the variables in some cutset are instantiated the net is locally consistent, we can "eliminate" these variables and their related constraints from the problem, as shown above. This cuts the loops and makes the constraint net acyclic. In this case, according to Theorem (1), local consistency is equivalent to global consistency. In addition, regardless of the order in which variables are instantiated, a complete solution can be found without any backtracking. We can now present a first modified version of MAC, in the form of the following algorithm:

1. Enforce arc consistency on the constraint network. If the domain of any variable becomes empty, return failure
2. Identify a cycle-cutset $\mathcal{C}$ of the constraint graph
3. Instantiate all variables in $\mathcal{C}$ while maintaining full arc consistency in the entire constraint network. If this is not possible, return failure
4. Use any algorithm to extend the partial solution obtained in step 3 to a complete solution, in a backtrack-free manner.

So far we showed that, in order to guarantee the existence of a complete solution in the context of maintaining arc consistency, it is sufficient to obtain a partial solution, by successfully instantiating only a subset of the variables, namely the cycle-cutset of the constraint graph. A simple heuristics to find a cycle-cutset is to order the variables in decreasing order of their degree, which explains why this static ordering performed so well in our tests.

Let us see if we can do better by *propagating less*. As we indicated earlier, after each modification MAC tries to restore the network to an arc consistent state. We claim that it is sufficient to bring the network to a partially arc consistent state only. More exactly, we need to maintain arc consistency just in part of the constraint graph, involving only some of the variables and constraints of the original CSP.

Figure 3 presents the constraint graph of our example, in which variables not involved in any cycle have been grayed. Once arc consistency is established, these variables become irrelevant for the search process. If the problem is inconsistent, none of this variables can be the source of the inconsistency. If there is a partial solution instantiating any of the normal variables in Figure 3, we are guaranteed to be able to extend it to a complete solution in a backtrack-free manner. Therefore, they can be disconnected from the constraint network, until we decide whether it is possible to instantiate successfully the variable which are left. During search the algorithm will propagate any change, and restore consistency accordingly, only in a (potentially small) part of the network. This partially arc consistent state is equivalent with the fully arc consistent state in the sense that both lead to exactly the same set of complete solutions.

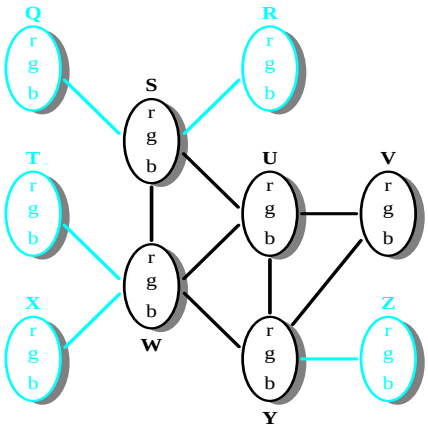

Figure 3. The constraint net after eliminating the variables in $\mathcal{T}$

Once all the variables which are still part of the network are instantiated, it is enough to reconnect the variables previously disconnected and to enforce arc consistency (or directed arc consistency) in order to obtain global consistency and to extend the partial solution to a complete solution in a backtrack-free manner.

The two ideas, *instantiate less* and *propagate less*, can now be combined under the name of *MACE (MAC Extended)*, which instantiates only a subset of the CSP variables while maintaining only a partially arc consistent state of the constraint network. The gain in efficiency is twofold. Instantiating a smaller number of variables aims at reducing the number of backtracks (and, accordingly, the number of constraint-checks, nodes visited, values deleted, etc.). Since the values disconnected are not part of any cycle-cutset, and hence, will not be instantiated in the first phase of the algorithm, the limited propagation implied by the second idea does not influence at all the number of backtracks or nodes visited, but reduces the number of constraint checks and values deleted.

3 Related Work

The idea of using the cycle-cutset of a constraint graph to improve the efficiency of CSP algorithms was used in (Dechter & Pearl 1988) as part of the *cycle-cutset method (CC)* for improving backtracking on discrete domain CSPs. (Hyvonen 1992) uses it for *interval CSPs*. A related idea is used in (Solotorevsky, Gudes, & Meisels 1996) for solving *distributed CSPs*.

Dechter and Pearl's cycle-cutset method can be described by the following scheme.

1. Partition the variables into two sets: a cycle-cutset of the constraint graph, $\mathcal{C}$, and $\mathcal{T}$, the complement of $\mathcal{C}$
2. Find a(nother) solution to the problem with variables in $\mathcal{C}$ only, by solving it independently. If no solution can be found, return failure

3. Remove from the domain of variables in $\mathcal{T}$ all values incompatible with the values assigned to variables in $\mathcal{C}$ and achieve directed arc consistency at variables in $\mathcal{T}$. If the domain of any variable becomes empty, restore all variables in $\mathcal{T}$ to their original state and repeat step 2
4. Use a backtrack-free search for extending the partial solution found in step 2 to a complete solution.

The major problem with the cycle-cutset method is its potential for thrashing. One type of thrashing is ilustrated by the following example. Suppose the variables in $\mathcal{C}$ are instantiated in the order $X, Y, \ldots$ Suppose further that there is no value for some variable Z in $\mathcal{T}$ which is consistent, according to constraint C_{XZ}, with value a for X. Whenever the solution to the cutset instantiates X to a, step 2 will fail. Since this can happen quite often, the cycle-cutset method can be very inefficient. We can eliminate this type of thrashing if we make the constraint network arc consistent before search starts, in a preprocessing phase.

A different type of thrashing, which cannot be eliminated by simply preprocessing the constraint network, is the following. Suppose that after making the network arc consistent initially, the domain of variable Z contains two values, c and d. Furthermore, value a for X supports value c on C_{XZ}, but does not support d. On the other hand, value b for Y supports d and not c on C_{YZ}. The cycle-cutset method will discover the inconsistency only while trying to instantiate Z, and this failure will be repeated for each solution of the cutset problem instantiating X to a and Y to b.

Our approach maintains arc consistency during the search (in fact, it maintains an equivalent state, as explained above). This eliminates both sources of thrashing and leads to substantial improvements over the cycle-cutset method.

4 Implementation

The goal of our paper is to compare the performance of three algorithms: MAC, the cycle-cutset method (CC), and the new algorithm we propose, MACE.

Figure 4 presents a high level description of the basic MAC algorithm.

It is worth stressing the differences between MAC and another algorithm that restores arc consistency arc consistency, called *Really Full Lookahead (RFL)* (Nadel 1988). Once the constraint network is made arc consistent initially (line 1), MAC restores arc consistency after each instantiation, or forward move, (lines 12–14), as RFL does, and, in addition:

- whenever an instantiation fails, MAC removes the refuted value from the domain and restores arc consistency (lines 6–7 and 19–20);
- after each modification of the network, both after instantiation and refutation, MAC chooses a (possible new) variable, as well as a new value (lines 3 and 16).

For our experiments we implemented a slightly improved version of MAC, called MAC-7ps (Regin 1995). According to the results presented in (Bessiere,

```
    MAC ( in: Var ; out: Sol ) return boolean
1       consistent ← INITIALIZE( )
2       while consistent do
3           (X, val_x) ← SELECT( Var, 0 )
4           if SOLVE( (X, val_x), Var \ {X}, Sol, 1 ) then
5               return true
6           D_x ← D_x \ {val_x}
7           consistent ← D_x ≠ ∅ and PROPAGATE( Var \ {X}, 1 )
        □
8       return false
    □
    SOLVE( in: (X, val_x), Var, Sol, level ; out: Sol ) return boolean
9       Sol ← Sol ∪ {(X, val_x)}
10      if level = N then
11          return true
12      for each a ∈ D_x, a ≠ val_x do
13          D_x ← D_x \ {a}
14      consistent ← PROPAGATE( Var, level )
15      while consistent do
16          (Y, val_y) ← SELECT( Var, level )
17          if SOLVE( (Y, val_y), Var \ {Y}, Sol, level+1 )
18              return true
19          D_y ← D_y \ {val_y}
20          consistent ← D_y ≠ ∅ and PROPAGATE( Var \ {Y}, level )
        □
21      Sol ← Sol \ {(X, val_x)}
22      RESTORE( level )
23      return false
    □
```

Fig. 4. Generic MAC Algorithm – Top Level Procedures

Freuder, & Regin 1995), (Bessiere & Regin 1996) and (Regin 1995), MAC-7ps is the best general-purpose CSP algorithm to date. It is an AC-7 based implementation of the basic MAC, with one notable improvement: special treatment of singleton variables. The idea is roughly the following. After restoring arc consistency, singleton variables can be disconnected temporarily from the network. The goal is to avoid studying the constraints connecting other variables to the singletons. A detailed description of the implementation can be found in (Regin 1995).

MACE and CC need an algorithm to find a cycle-cutset. There is no known polynomial algorithm for finding the minimum cycle-cutset. There are several heuristics which yield a good cycle-cutset at a reasonable cost. The simplest sorts first the variables in decreasing order of their degree. Then, starting with the variable with the highest degree, as long as the graph still has cycles, add the variable to the cycle-cutset and remove it, together with all the edges involving it, from the graph. Assuming that lexical ordering is used to break ties, this method yields for our example the cycle-cutset presented in Figure 5a. Variables are added to the cutset in the order W, S and U. The worst case run time

complexity for this heuristic is $O(ne)$.

A smaller cutset can be obtained if, before adding a variable to the cutset, we check whether it is part of any cycle or not. For example, after removing W from the graph, S is not involved in a cycle anymore, and, with the new algorithm, we find the cycle cutset in Figure 5b. The worst case time complexity for this heuristics is $O(ne)$. Additional work leads to an even smaller cutset. The cutset shown in Figure 2a is obtained by a third heuristic, which determines dynamically the number of cycles in which each variable is involved and adds to the cutset at each step the variable participating in the most cycles. The worst case time complexity of this heuristic is $O(n^2e)$.

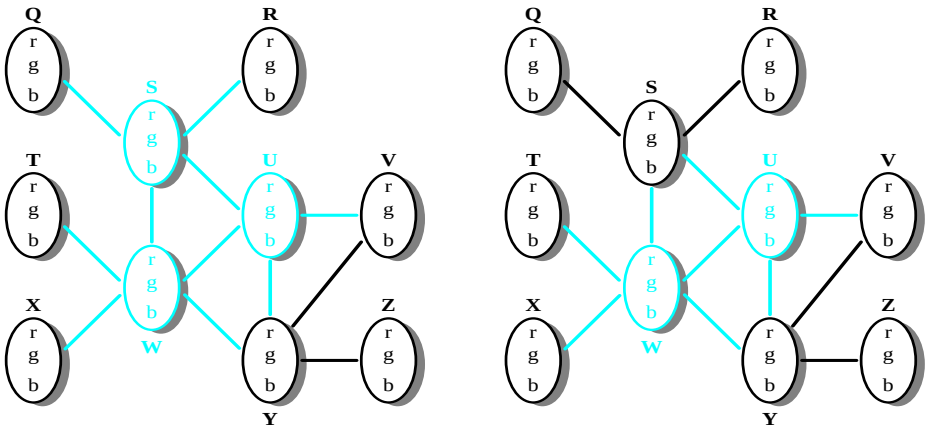

Figure 5. Different cycle-cutsets for the example in Figure 1

We used exactly the same algorithm in implementing both CC and MACE. We performed the tests using both the second and the third heuristic presented above. Based on our results, the third heuristic yields slightly smaller cutsets which translate sometimes into small gains in efficiency, but no major improvement. In the case of large problems probably the second heuristic is the best choice, because of its lower cost. Due to the space restriction, we report in this paper only the results obtained using the third heuristic.

The implementation of CC is straightforward. Since there is no requirement on the algorithm to solve the cutset subproblem, to keep the comparison with MACE as fair as possible, we used the basic MAC as our choice.

To implement MACE, we modified the algorithm in Figure 4 as follows.

- After enforcing arc consistency, procedure INITIALIZE (line 1) partitions the set of variables into two sets, one of which is the cycle-cutset $\mathcal{C}$. Disconnect from the constraint network all variables which are not involved in any cycle and add them to the set of disconnected variables, $\mathcal{U}$.
- Restrict procedure SELECT (lines 3 and 16) to choose only from among variables in $\mathcal{C}$.

- Whenever a variable becomes a singleton disconnect it from the network and add it to $\mathcal{U}$. If this makes other variables "cycle-free", disconnect them and add them to $\mathcal{U}$ as well. Continue this process until no more variables can be disconnected.
- Once all variables in $\mathcal{C}$ have been successfully instantiated, reconnect all variables in $\mathcal{U}$ and eliminate from their domains all values incompatible with the values assigned to variables in the cutset. Enforce directed arc consistency with respect to some width-1 order on the problem containing only variables in the complement of $\mathcal{C}$ and conduct a backtrack-free search for a complete solution.

5 Experiments

We tested our approach on random binary CSPs described by the usual four parameters: number of variables, domain size, constraint density and constraint tightness. We generate only connected constraint graphs (connected components of unconnected components can be solved independently). Therefore the number of edges for a graph with n vertices is at least $n-1$ (for a tree, density=0) and at most $n(n-1)/2$ (for a complete graph, density=1). Constraint density is the fraction of the possible constraints beyond the minimum $n-1$, that the problem has. Thus, for a problem with constraint density D and n variables, the exact number of constraints that the problem has is $\lfloor n-1+D(n-1)(n-2)/2 \rfloor$.

Constraint tightness is defined as the fraction of all possible pairs of values from the domains of two variables, that are not allowed by the constraint. So, for a domain size of d and a constraint tightness of t, the exact number of pairs allowed by the constraint is $\lfloor (1-t)d^2 \rfloor$.

The tests we conducted addressed the problem of finding a single solution to a CSP (or determining that no solution exists). We ran three sets of experiments on hard random problems problems, situated on the ridge of difficulty in the density/tightness space.

For the first two sets we generated problems with 20 variables and domain size of 20. The density of the constraint graph varies between 0.05 and 0.95, with a step of 0.05, while the tightness varies between $T_{crit} - 0.08$ and $T_{crit} + 0.08$, with a step of 0.01. For each pair of values (density, tightness) we generated 10 instances of random problems, which gives us roughly a total of 3,200 problems per set.

The problems in the third set have 40 variables and domain size of 20. We expected the problems in this set to be much harder than the ones in the previous sets. Therefore the constraint density varies only between 0.05 and 0.30. The tightness varies between $T_{crit} - 0.08$ and $T_{crit} + 0.08$, with a step of 0.01. We generated again 10 instances of random problems for each (density, tightness) pair, which gives us almost 1,000 problems for this set.

We present the results of the experiments using two types of plots. One type represents, on the same graph, the performance of two algorithms in terms of constraint checks, as a function of tightness. Due to space restrictions, the results

from different sets of problems, with different densities, are ploted on the same graph (e.g. Figure 6).

The second type of plots represents the ratio between the performance of two algorithms as a function of tightness, in the form of a set of points. Again, results from different sets of problems, with different densities, are ploted on the same graph. Each point on the graph represents the average over the 10 problems generated for the corresponding (density, tightness) pair (e.g. Figure 8).

It is very important what measure is used to judge the performance of algorithms. The usual measure in the literature is the number of constraint checks performed by an algorithm during the search for a solution. Whenever establishing that a value a for a variable X is consistent with a value b for a variable Y, a single consistency check is counted. Constraint checks are environment independent, but are highly dependent on the efficiency of the implementation. In our case, since we use more or less the same implementation for all the algorithms, we choose this measure as being representative for the search effort.

We ran experiments comparing the performance of three algorithms: the cycle-cutset method, MAC-7ps and MACE. All algorithms used the dynamic variable ordering heuristic proposed in (Bessiere & Regin 1996), choosing variables in increasing order of the ratio between domain size and degree.

The first set of experiments compares the performance of the cycle-cutset method and MACE on the first set of test problems. Figure 6 shows the relative average performance of the algorithms in terms of constraint checks.

As we can see, MACE outperforms substantially the cycle-cutset method on problems with densities up to 0.90–0.95, when they have approximately the same performance. The size of the cycle-cutset varies almost liniarly with the density, from 3 for density 0.05 to almost 18 for density 0.95. For problems in the high density area the cutset is almost the entire set of variables (this are 20-variable problems) and therefore the behavior of the two algorithms is almost identical.

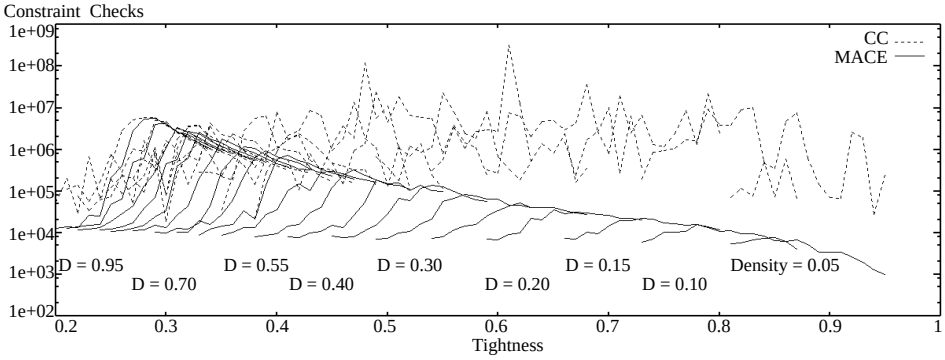

Figure 6. Comparison between the cycle-cutset method and MACE

As suggested in Section 3, we added an arc consistency preprocessing phase to CC and ran this combination on the same problem sets. The results are presented in Figure 7. As we can see, the preprocessing improves the performance of CC

only in the very sparse region, by discovering the arc inconsistent problems. On the rest of the problems the preprocessing had practically no effect.

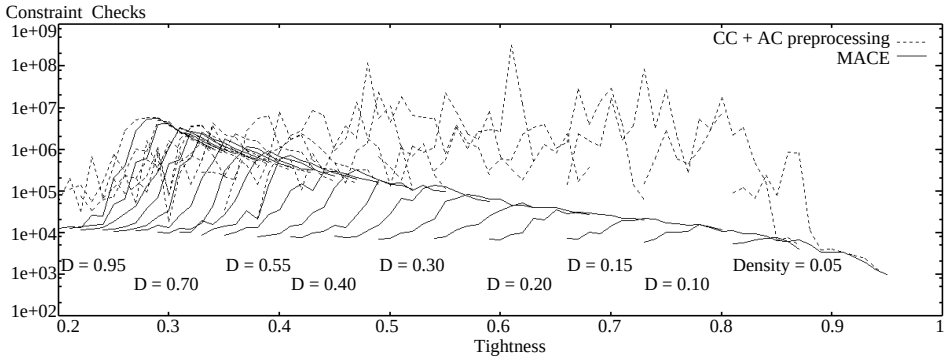

Figure 7. Comparison between the cycle-cutset method with arc-consistency preprocessing and MACE

The same results are presented from a different perspective in Figure 8, which shows the ratio between the constraint checks performed by the cycle-cutset method and MACE. The advantage of MACE is very clear.

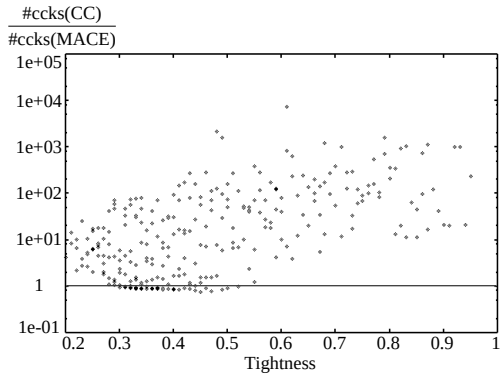

Figure 8. Performance ratio between the cycle-cutset and MACE

The second set of experiments compares the performance of MAC-7ps and MACE. Figure 9 shows the relative average performance of the two algorithms in terms of constraint checks on the second set of problems, with 20 variables. As we can also see from the plot in Figure 10, which shows the ratio between the number of constraint checks for MAC-7ps and MACE on the same set of problems, MACE performs better than MAC-7ps. For problems with high densities (0.9 – 0.95) although MACE still dominates, MAC-7ps wins a few times. Again, the explanation consists in the size of the cycle-cutset, which increases with the density. In this particular area the sets of variables instantiated by the two algorithms become almost the same. Therefore, both algorithms exhibit similar

behaviors, MACE being still slightly better than MAC-7ps on average.

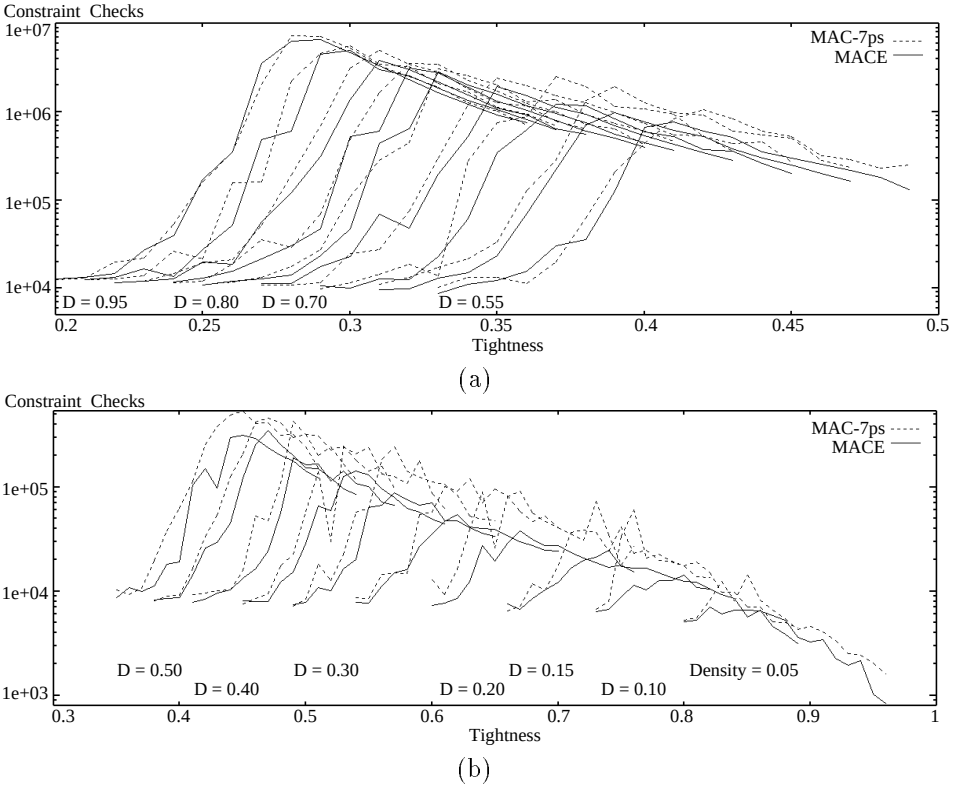

Figure 9. Comparison between MAC-7ps and MACE on problems with 20 variables

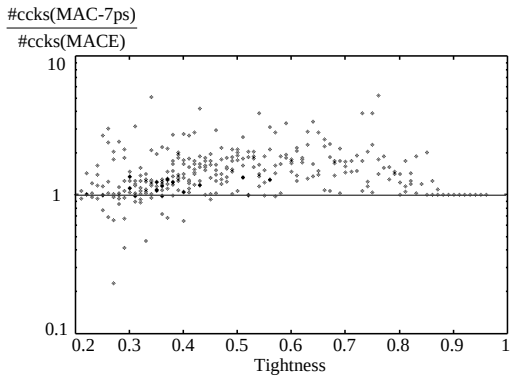

Figure 10. Performance ratio between MAC-7ps and MACE on problems with 20 variables

The last set of experiments studies the scalability of our approach as problem size increases. We therefore compared the performance of MAC-7ps and MACE

on problems with 40 variables, using the third set of random problems. Figure 11 shows again the relative average performance of the two algorithms in terms of constraint checks, while Figure 12 presents the same data, but in the form of the ratio between the number of constraint checks for MAC-7ps and MACE. Both plots show again that MACE outperforms MAC-7ps significantly. The data also suggests that MACE scales well, the relative gain in efficiency increasing as the problems become larger.

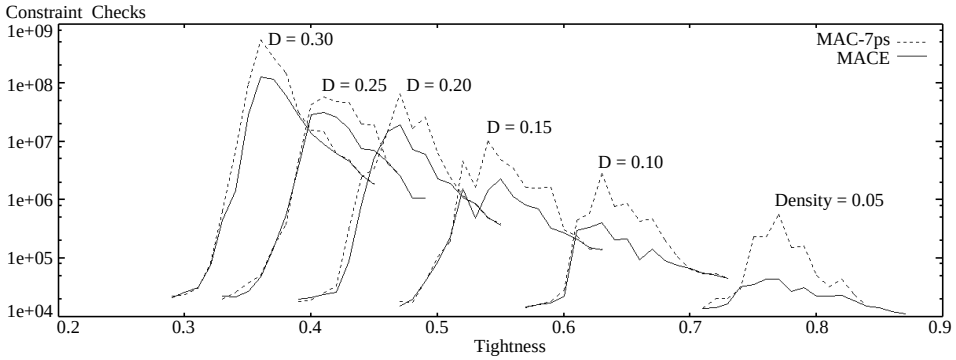

Figure 11. Comparison between MAC-7ps and MACE on problems with 40 variables

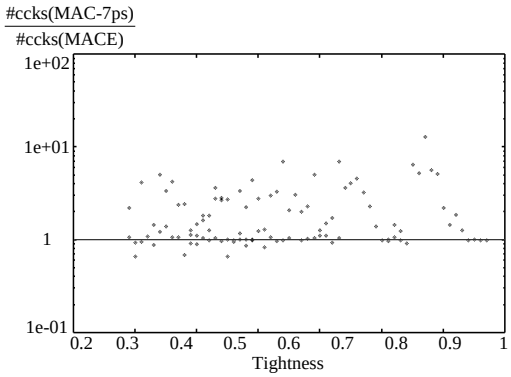

Figure 12. Performance ratio between MAC-7ps and MACE on problems with 40 variables

6 Conclusion

We have analyzed the advantages of maintaining full arc consistency during search. Our analysis led to an improved version of MAC, called MACE. In extensive experiments MACE consistently outperformed MAC.

Acknowledgment

References

1. Bessiere, C., and Regin, J.-C. 1996. Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *Second International Conference on Principles and Practice of Constraint Programming - CP96*, number 1118 in Lecture Notes in Computer Science, 61–75.
2. Bessiere, C.; Freuder, E. C.; and Regin, J.-C. 1995. Using inference to reduce arc consistency computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume I, 592–598.
3. Dechter, R., and Pearl, J. 1987. The cycle-cutset method for improving search performance in ai applications. In *Proceedings of the 3rd IEEE Conference on AI Applications*, 224–230.
4. Dechter, R., and Pearl, J. 1988. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence* 34(1):1–38.
5. Freuder, E. C. 1982. A sufficient condition for backtrack-free search. *Journal of the ACM* 29(1):24–32.
6. Grant, S. A., and Smith, B. M. 1995. The phase transition behaviour of maintaining arc consistency. Technical Report 95.25, University of Leeds, School of Computer Studies.
7. Hyvonen, E. 1992. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence* 58(1–3):71–112.
8. Nadel, B. 1988. Tree search and arc-consistency in constraint satisfaction algorithms. In Kanal, L., and Kumar, V., eds., *Search in Artificial Intelligence*. Springer-Verlag. 287–342.
9. Regin, J.-C. 1995. *Developpement d'outils algorithmiques pour l'Intelligence Artificielle. Application a la chimie organique*. Ph.D. Dissertation, Universite Montpellier II.
10. Sabin, D., and Freuder, E. C. 1994. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence*.
11. Solotorevsky, G.; Gudes, E.; and Meisels, A. 1996. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Second International Conference on Principles and Practice of Constraint Programming - CP96*, number 1118 in Lecture Notes in Computer Science, 561–562.
12. Tsang, E. 1993. *Foundations of Constraint Satisfaction*. Academic Press Limited.