# MAC-CBJ: maintaining arc consistency with conflict-directed backjumping

Department of Computer Science Research Report 95/177
May 1995

Patrick Prosser
Department of Computer Science
University of Strathclyde
Glasgow G1 1XH
Scotland
pat@cs.strath.ac.uk

**Abstract**

Sabin and Freuder have demonstrated that increased constraint propagation can lead to a reduction in overall search effort. They described an algorithm based on arc consistency, and they called it MAC, for maintaining arc consistency. Their algorithm is a chronological backtracker, and it is probable that it can be improved by adding conflict-directed backjumping (CBJ). The algorithm MAC-CBJ is described here, without proof, and without a comparative study.

# 1 Introduction

Sabin and Freuder [17] resurrected an algorithm that was reported by Gaschnig, and called it MAC, for maintaining arc consistency. They showed, that contrary to conventional wisdom, more constraint propagation can lead to more efficient search. However, their algorithm is a chronological backtracker. Clearly, there should be scope for improvement, if only by giving MAC a backjumping capability, and equally clearly, this is possible. I describe that algorithm below, and I call it MAC-CBJ[1]. The report starts by describing Mackworth's arc consistency algorithm AC3, and then moves on to MAC, showing that the forward checking routine can be considered as a degenerate form of MAC (we delete one line from the algorithm for MAC to get FC). Conflict-directed backjumping (CBJ) is then described, and it is shown how MAC can be combined with CBJ.

# 2 Arc Consistency

For a binary constraint network to be arc consistent we are guaranteed that for any value $x_i$ in the domain $D_i$ of a variable $V_i$ there will exist in the domain $D_j$ of an adjacent variable $V_j$ a value $x_j$ that is compatible with $x_i$. That is, there is *support* for values. There is a range of algorithms for achieving arc consistency[2], ranging from Mackworth's AC3, Mohr and Henderson's AC4, Van Hentenryck's AC5, Bessière's AC6 and AC6++ and Freuder's AC7 [13, 14, 2, 3, 8, 4][3]. We will define MAC in terms of AC3.

At the heart of AC3 is the function *revise*$(i, j)$, shown below. This function revises a constraint $C_{i,j}$, removing from the domain of $V_i$ (ie. domain[i]) values that have no support in the domain of $V_j$ (ie. domain[j]).

```
 1.   function revise(i,j)
 2.   begin
 3.     revised := false;
 4.     for each x in domain[i]
 5.     do begin
 6.        supported := false;
 7.        for each y in domain[j] while (not supported)
 8.        do supported := check(i,x,j,y);
 9.        if not supported
10.        then begin
11.             domain[i] := remove(x,domain[i]);
12.             revised := true;
13.             end;
14.        end;
15.      revised;
16.    end;
```

The function call *check*$(i, x, j, y)$ in line 8 delivers *true* if the pair of instantiations $V_i$ with $x$ and $V_j$ with $y$ is consistent with respect to the constraint $C_{i,j}$, and delivers *false* otherwise.

The arc consistency algorithm AC3 proceeds by putting all constraints in the graph onto a queue (call it $Q$). AC3 then iterates, by popping off a constraint $C_{i,j}$ from $Q$ and revising that constraint. If this results in values being removed from the domain of variable $V_i$ (ie. *revise*$(i, j)$ delivers a result of *true*) then all constraints

[1] The name appeals to me for two reasons. First, it is consistent with the naming convention adopted in [15] and it has a distinctive Scottish ring to it.

[2] Note that arc consistency is frequently referred to as 2-consistency.

[3] Bessière's AC6++ and Freuder's AC7 are in fact the same algorithm, although developed independently, and both were presented at the same workshop in ECAI-94. AC3 is cubic complexity, whereas AC-4/5/6/7 are quadratic.

incident on $V_i$ are added to $Q$, with the exception of the constraint $C_{j,i}$, so long as they are not already on $Q$. AC3 terminates when $Q$ is empty.

```
1.  function AC3(Q)
2.  begin
3.    consistent := true;
4.    while not(empty?(Q)) and consistent
5.    do begin
6.       (i,j) := pop(Q);
7.       if revise(i,j)
8.       then begin
9.            consistent := not(empty?(domain[i]));
10.           Q := Q U {(k,i) | (k,i) in arcs(G), k <> j}
11.           end;
12.      end;
13.    consistent;
14.   end;
```

Line 10 above is worthy of comment[4]. If $revise(i,j)$ results in the removal of values from $D_i$ (line 7) this cannot affect the support for values in $D_j$. That is, it is assumed that the constraints are symmetric, such that if $C_{i,j}$ exists then so too does $C_{j,i}$, and if $x_i$ supports $y_j$ then $y_j$ supports $x_i$ also. Consequently, if an unsupported value is removed from $D_j$ due to $revise(i,j)$ it cannot affect any values in $D_j$. Note also that we can use AC3 incrementally. Rather than call AC3 once with $Q = arcs(G)$, we may call AC3 repeatedly, each time with $Q$ being a pair of symmetric constraints. This may be less efficient, but it will not compromise completeness.

# 3 Maintaining arc consistency (MAC)

The algorithm MAC has appeared under various guises. In Gaschnig's thesis [10] the algorithm is referred to as DEEB, ie. *Domain Element Elimination with Backtracking*, and earlier in [9] as CS2. The basic idea is that when instantiating a variable $V_i$ with a value $x_i$ the domain of the current variable is set momentarily to a single value, ie. $D_i \leftarrow \{x_i\}$, and the uninstantiated variables (ie. the *future* variables) are then made arc-consistent. That is, when instantiating $V_i$ AC3 is applied, but only to the queue of constraints $Q \leftarrow \{(j,i) \mid (j,i) \in arcs(G), j > i\}$, ie. the set of constraints incident on $V_i$, and coming from the future, and subsequent propagation takes place only between future variables. If this results in a domain wipe out (dwo)[5] then the domains of the future variables are reset to what they were prior to the most recent call to AC3, and a new value is tried for $V_i$. If no more values remain for $V_i$ then $V_{i-1}$ is reinstantiated, ie. chronological backtracking takes place. On the other hand if values remain in the domains of all future variables after the application of AC3 then another variable may be instantiated.

```
function AC3-MAC(Q,cv)
begin
  consistent := true;
  while not(empty?(Q)) and consistent
  do begin
     (i,j) := pop(Q);
     if revise(i,j)
     then begin
          consistent := not(empty?(domain[i]));
          Q := Q U {(k,i) | (k,i) in arcs(G) & k <> j & k > cv}
          end;
     end;
```

[4] Also note that I have used <> in place of $\neq$

[5] That is, the domain of some variable becomes empty.

```
13.      consistent;
14.    end;
```

Function AC3-MAC performs the required actions. It takes an additional argument $cv$, the index of the current variable. Line 10 is modified such that propagation only takes place between future variables. It should be noted that if we delete line 10 above MAC becomes Haralick and Elliott's forward checking routine (FC) [12], and if $cv$ is set to zero AC3-MAC behaves as AC3.

It should be noted that there is no good reason why MAC should be based on AC3. Sabin and Freuder's implementation of MAC [17] is based on AC4, and is arguably more efficient than the one described here. It might be argued that the version of MAC presented here should strictly be called MAC3, and that there will be other versions, such as MAC4, MAC6, and MAC7.

MAC will be prone to thrashing [13]. That is, in the event of hitting a dead end MAC will attempt to resolve this by falling back on the previous instantiation, and this instantiation might play no role whatsoever in the conflict. The search process will then reinstantiate this variable and then proceed to carry out the same set of actions with the same set of outcomes. This is the most naive form of thrashing[6].

# 4 Maintaining arc consistency within conflict directed backjumping (MAC-CBJ)

The potential to thrash can be reduced, but probably not eliminated, by adding conflict-directed backjumping (CBJ) [15] to MAC. In it's simplest form[7], CBJ can be considered as a marriage between Gaschnig's backjumping (BJ) and Dechter's graph-based backjumping (GBJ) [6]. CBJ checks *backwards* from the current variable to the *past* variables[8]. If a trial instantiation of $V_i$ is inconsistent with respect to some past variable $V_g$, where $g < i$, then the index $g$ is added to the *conflict set* $CS_i$ of variable $V_i$. On reaching a dead end on $V_i$, CBJ jumps back to $V_g$ where $g$ is the largest value in $CS_i$. That is, $V_g$ is the deepest past variable in conflict with $V_i$. On jumping back to $V_g$ the conflict set $CS_g$ is updated such that it becomes $CS_g \leftarrow CS_g \cup CS_i - g$, ie. the union of the conflict sets with the index $g$ removed. Conflict sets *below* $V_g$ in the search tree are then annulled, ie. for all $h$, where $g < h \leq i$, $CS_h \leftarrow \emptyset$. If on jumping back to $V_g$ there are no values left to be tried CBJ jumps back again, to $V_f$, where $f$ is the largest value in $CS_g$.

In designing DEEB, Gaschnig considered the instantiation of a variable as the addition of a constraint[9]. We could stretch this further and consider $V_i$ as being in conflict with itself! Consequently, when $V_i$ is instantiated its conflict set becomes momentarily $CS_i \leftarrow \{i\}$. AC3-MAC is then applied to the queue of constraints $Q \leftarrow \{(j,i) \mid (j,i) \in arcs(G), j > i\}$ and $cv \leftarrow i$. If on revising a constraint $revise(j,i)$ removes any values from $D_j$ then the conflict set of $V_j$ is updated as follows: $CS_j \leftarrow CS_j \cup CS_i$. That is, some value in $D_j$ is in conflict with the current instantiation. We now need to modify AC3-MAC such that when constraints are propagated, so too are conflict sets. Furthermore, if a dwo does occur as a result of propagation we need to identify the variable involved. The modified function is given below as AC3-MAC-CBJ.

[6] Note that a more subtle form of thrashing can occur in informed backjumpers, such as BJ and CBJ [10, 15]. This phenomenon has been studied in some depth by Gent and Walsh and by Smith and Grant [11, 18]

[7] That is, CBJ on its own, not combined with any other algorithm such as BM or FC

[8] The past variables are the instantiated variables.

[9] An identical approach was taken by Burke when designing the constraint maintenance system for the Distributed Asynchronous Scheduler. A scheduling decision was viewed as the addition of a unary constraint [5]

```
1.    function AC3-MAC-CBJ(Q,cv)
2.    begin
3.      consistent := true;
4.      while not(empty?(Q)) and consistent
5.      do begin
6.         (i,j) := pop(Q);
7.         if revise(i,j)
8.         then begin
9.              consistent := not(empty?(domain[i]));
10.             Q := Q U {(k,i) | (k,i) in arcs(G) & k <> j & k > cv};
10.1            cs[i] := cs[i] union cs[j];
11.             end;
12.        end;
13.       if consistent then 0 else i;
14.    end;
```

A line has been added, 10.1, to propagate conflict sets. Line 13 has changed as well; the function now delivers a result of 0 if all future variables still have values left in their domains, otherwise the function delivers the index $i$ of the variable that has experienced a dwo, ie. $D_i = \emptyset$.

To realise MAC-CBJ we incorporate AC3-MAC-CBJ into the conflict-directed backjumper. On instantiating a variable $V_i$ a call is made to AC3-MAC-CBJ(Q,i), and if this delivers a result of zero then the search process can select another variable for instantiation; that is, the search moves forwards. If, on the other hand, AC3-MAC-CBJ(Q,i) delivers a result $x$, where $x > i$, then the conflict set $CS_i$ is reinstated to its value before the the call to AC3-MAC-CBJ(Q,i)[10] and is then updated as follows: $CS_i \leftarrow CS_i \cup CS_x - i$. Furthermore, the conflicts sets and domains of the future variables are then reset to what they were immediately prior to the call to AC3-MAC-CBJ(Q,i)[11]. If there are no more values remaining to be tried for $V_i$ then backjumping takes place to $V_g$, where $g$ is the largest index in $CS_i$. The conflict set $CS_g$ is updated as in CBJ, ie. $CS_g \leftarrow CS_g \cup CS_i - g$, and the conflict sets and domains of future variables are reset to the values they had immediately prior to the call to AC3-MAC-CBJ(Q,g).

# 5 Conclusion

The description of the algorithms have assumed a static instantiation order. This has only been done so that my task is easier, ie. I hope that the description of the algorithm is relatively easy to understand. A dynamic variable ordering heuristic [16] can readily be incorporated into MAC and MAC-CBJ. The trick is to replace the variable index with the position, or depth, of the variable within the search tree. Generally, this approach has been taken for granted, and has not been reported. However, a rather crisp description of this has been given by Bacchus and van Run [1].

This report might be considered a touch unusual in that it describes a new algorithm, admittedly one combined from two existing algorithms, but does not position it with respect to other algorithms. Part of the reason for doing this is that the empirical study required is a substantial task in its own right, and my objective was to report the algorithm as early as possible, and encourage others to position it. There is currently considerable interest in the study of exceptionally hard problems, ie. under-constrained problems that require exceptional amounts of effort to find a solution. It may be that MAC-CBJ could turn out to perform well

[10] That is, the value it had before it was set to $\{i\}$.

[11] Clearly, either a recursive implementation is required to do this, or an explicit mechanism must be put in place for the stacking and unstacking of domains and conflicts sets.

on this class of problem, even though it may be outperformed by less sophisticated algorithms on other problem classes.

There is another reason for reporting this algorithm. As Sabin and Freuder have noted, the constraint programming community have been using MAC, even though *conventional wisdom* suggests that they should not! However, the constraint programming community are still chronologically backtracking. It is time for them to jump back!

## Acknowledgements

This work wouldn't have happened, if I hadn't sat through an excellent presentation by Daniel Sabin at ECAI-94. I would also like to thank Jean-Francois Puget for encouraging me to do useful work, although he might not have succeeded ⌣̈

## References

[1] F. Bacchus and P. Van Run, Dynamic variable ordering in CSP's. Department of Computer Science, University Toronto, 1995

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

[3] C. Bessière, J-C Régin, An arc consistency algorithm optimal in the number of constraint checks. ECAI94 Workshop W6, Constraint Processing, 9-16

[4] C. Bessière, E.C. Freuder, and J-C Régin, Using inference to reduce arc consistency computation, to appear in *Proc IJCAI-95*

[5] P. Burke and P. Prosser, The Distributed Asynchronous Scheduler, in Intelligent Scheduling (eds. M. Zweben and M.S. Fox), Morgan Kaufmann Publishers (1994), 309-339

[6] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* **41**(3) (1990) 273-312.

[7] R. Dechter, Constraint Networks, in *Encyclopedia of Artificial Intelligence* (Wiley, New York, 2nd ed., 1992) 276-286.

[8] E.C. Freuder, Using metalevel constraint knowledge to reduce constraint checking. ECAI94 Workshop W6, Constraint Processing, 27-33

[9] J. Gaschnig, A constraint satisfaction method for inference making, *Proc 12th Annual Allerton Conf. Circuit and System Theory*, U. Ill. Urbana-Champaign, October. 2-4, 1974

[10] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Tech. Rept. CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA (1979).

[11] I.P. Gent and T. Walsh, The satisfiability constraint gap, to appear in *Artif. Intell.* special issue on phase transitions in problem spaces.

[12] R.M. Haralick and G.L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artif. Intell.* **14** (1980) 263-313.

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

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

[15] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, *Comput. Intell.* **9**(3) (1993) 268-299.

[16] P.W. Purdom, Search rearrangement backtracking and polynomial average time, *Artif. Intell.* **21** (1983) 117-133.

[17] D. Sabin and E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, *Proceedings ECAI-94,* Amsterdam, The Netherlands (1994) 125-129.

[18] B.M. Smith and S.A. Grant, Sparse constraint graphs and exceptionally hard problems, to appear in *Proc IJCAI-95.*