An Improved Algorithm for Maintaining Arc Consistency in Dynamic Constraint Satisfaction Problems

Roman Barták

Charles University in Prague
Malostranské nám. 2/25, Praha 1, Czech Republic
roman.bartak@mff.cuni.cz

Pavel Surynek

Czech Technical University
Technická 2, Praha 6, Czech Republic
pavel.surynek@seznam.cz

Abstract

Real world is dynamic in its nature, so techniques attempting to model the real world should take this dynamicity in consideration. A well known Constraint Satisfaction Problem (CSP) can be extended this way to a so called Dynamic Constraint Satisfaction Problem (DynCSP) that supports adding and removing constraints in runtime. As Arc Consistency is one of the major techniques in solving CSPs, its dynamic version is of a particular interest for DynCSPs. This paper presents an improved version of AC|DC-2 algorithm for maintaining maximal arc consistency after constraint retraction. This improvement leads to runtimes better than the so far fastest dynamic arc consistency algorithm DnAC-6 while keeping low memory consumption. Moreover, the proposed algorithm is open in the sense of using either non-optimal AC-3 algorithm keeping a minimal memory consumption or optimal AC-3.1 algorithm improving runtime for constraint addition but increasing a memory consumption.

Introduction

Constraint programming is a successful A.I. technology for solving combinatorial problems modeled as constraint satisfaction problems (CSP). Dynamic Constraint Satisfaction Problem proposed by Decher and Dechter (1988) is an extension to a static CSP that models addition and retraction of constraints and hence it is more appropriate for handling dynamic real-world problems.

Several techniques have been proposed to solve Dynamic CSPs, including searching for robust solutions that are valid after small problem changes (Wallace and Freuder, 1998), searching for a new solution that minimizes the number of changes from the original solution (El Sakkout, Richards, Wallace, 1998), reusing the original solution to produce a new solution (Verfaillie and Schiex, 1994), or reusing the reasoning process. A typical representative of the last method – reusing the reasoning process – is maintaining dynamic arc consistency. The goal of maintaining dynamic arc consistency is keeping the problem arc consistent after constraint addition or constraint retraction. Adding a new constraint is a monotonic process which means that domains of variables can only be pruned. Existing arc consistency algorithms are usually ready for such incremental constraint addition so they can be applied when a new constraint is added to the problem. When a constraint is retracted from the problem then the problem remains arc consistent. However, some solutions of the new problem might be lost because the values from the original problem that were directly or indirectly inconsistent with the retracted constraint are missing in the domains. Consequently, such values should be returned to the domains after constraint retraction. Then we are speaking about *maximal arc consistency*.

Existing algorithms for restoring maximal arc consistency after constraint retraction are usually working in three stages: recovery of values deleted due to the retracted constraint, propagation of these recovered values to other variables, and removal of inconsistent values that were wrongly restored in the precedent stages. *DnAC-4* (Bessière, 1991) was one of the first dynamic arc-consistency algorithms. As the name indicates, the algorithm is based on AC-4 (Mohr and Henderson, 1986) and, actually, it uses all data structures proposed for AC-4. In addition to them, a new data structure *justification* was added to improve the efficiency of constraint retraction, in particular to minimize the number of wrongly restored values. This data structure keeps a link to the variable that caused deletion of the value from the variable domain during the AC domain pruning. DnAC-4 inherits the disadvantages of AC-4 and therefore *DnAC-6* (Debruyne, 1996) has been proposed to improve space complexity and average time complexity. DnAC-6 uses the same principles as DnAC-4 but it is integrated with AC-6 (Bessière, 1994) rather than using AC-4. DnAC-6 is currently the fastest dynamic arc consistency algorithm but it has the disadvantage of fine grained consistency algorithms which is a large space complexity. To keep low memory consumption, *AC|DC* algorithm has been designed by Berlandier and Neveu (1994). AC|DC is built around AC-3 (Mackworth, 1977) algorithm and it does not use the supporting data structures. As a consequence, more inconsistent values are restored in the second stage and these values must be removed in the third stage by non-

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

optimal AC-3 algorithm. Mouhoub (2003) proposed an improvement of the time complexity for AC|DC by using optimal AC-3.1 algorithm (Zhang and Yap, 2001). The resulting algorithm *AC-3.1|DC* has time and space complexity comparable to DnAC-6. Recently, Surynek and Barták (2004) proposed an improvement of AC|DC called *AC|DC-2* that uses AC-3 algorithm to preserve low memory consumption but improves the propagation stage by re-introducing justifications and new timestamps. Table 1 summarizes the worst-case time and space complexities of the existing dynamic AC algorithms.

	DnAC-4	**DnAC-6 AC-3.1|DC**	**AC|DC AC|DC-2**
Space	$O(ed^2+nd)$	$O(ed+nd)$	$O(e+nd)$
Time	$O(ed^2)$	$O(ed^2)$	$O(ed^3)$

Table 1. Complexity of existing dynamic AC algorithms.

In this paper we propose improvements to AC|DC-2 based on smarter usage of AC filtering and better exploitation of the timestamps. We will show experimentally that the resulting algorithm *AC|DC-2i* is practically faster than the so far fastest DnAC-6 and AC-3.1|DC algorithms. Moreover, the algorithm still keeps low memory consumption of AC|DC. To further improve time efficiency of the proposed algorithm it is possible to use optimal AC-3.1 algorithm resulting in *AC3.1|DC-2i*. This algorithm is slightly faster than AC|DC-2i when retracting a constraint and much faster when adding a constraint but it has the disadvantage of larger memory consumption.

Formal Background

A constraint satisfaction problem (CSP) P is a triple (X,D,C), where X is a finite set of variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for the variable x_i (the domain), and C is a finite set of constraints. In this paper we expect all the constraints to be binary, that is the constraint $c_{ij} \in C$ defined over variables x_i and x_j is a subset of the Cartesian product $D_i \times D_j$. Value a of variable x_i is *arc consistent* (AC) if and only if for each variable x_j connected to x_i by constraint c_{ij}, there exists a value $b \in D_j$ such that $(a,b) \in c_{ij}$. A CSP is *arc consistent* if and only if every value of every variable is arc consistent. A CSP is *maximally arc consistent* if it is the largest (according to inclusion) arc consistent problem. Arc consistency algorithms make the problems maximally arc consistent by removing only the values that are not arc consistent from respective domains.

Dynamic constraint satisfaction problem (DCSP) is a sequence $P_0, P_1, \ldots, P_n$, where each P_i is a CSP resulting from addition or retraction of a constraint in P_{i-1}. For simplicity reasons, we expect that P_0 contains no constraints; hence it is maximally arc consistent. The task of dynamic arc consistency is to make problem P_i maximally arc consistent using the information that problems $P_0, P_1, \ldots, P_{i-1}$ are maximally arc consistent.

Improved Algorithm AC|DC-2i

As we already noted, the existing dynamic arc consistency algorithms including AC|DC are working in three stages: recovery of values deleted due to the retracted constraint (initialization stage), propagation of these recovered values to other variables (propagation stage), and removal of wrongly restored inconsistent values (filtration stage). AC|DC loses efficiency due to restoration of many values that are not arc consistent and hence deleted afterwards (Debruyne, 1996). Therefore Surynek and Barták (2004) proposed to extend AC|DC with additional data structures that record a justification and a timestamp for every value eliminated from the variable domain. *Justification* is the first neighboring variable in which the eliminated value lost all supports. *Timestamp* is a time when the value has been eliminated. Time is modeled using a global counter that is incremented after every manipulation of variable domains. By using justifications and timestamps AC|DC-2 determines more accurately the set of values to be restored and hence it improves a lot over AC|DC.

We propose three extensions to AC|DC-2 that improve further the runtime. First, we suggest handling constraints in a directional way (like in standard AC-3; do not interchange it with directional AC) rather than the less efficient non-directional way from AC|DC-2. Second, we propose to apply the AC procedure only to newly restored values rather than to all values. Finally, we propose to use the timestamp in a finer way to further decrease the number of wrongly restored values.

Figure 1 shows an abstract code for constraint addition. It is basically the original AC-3 algorithm extended to compute justifications (`justif_var`) and timestamps (`justif_time`) and to update the global time counter (`gtime`). Moreover, only the values that are restored after the given time are checked for consistency (line 2 in **`filter-arc-ac3'`**). The CSP is represented by variables' domains `D` and by a set of arcs `C` describing the constraints.

```
function propagate-ac3'(revise, time)
1 queue := revise
2 while queue not empty do
3 |  select and remove an arc (u,v) from queue
4 L  queue := queue ∪ filter-arc-ac3'((u,v), time)
```

```
function filter-arc-ac3'((u,v), time)
1 modified := false
2 for each d∈D[u] s.t. restore_time[u,d]>time do
3 |   if d has no support in D[v] wrt (u,v) then
4 |     D[u] := D[u] − {d}
5 |     justif_var[u,d] := v
6 |     justif_time[u,d] := gtime
7 |     gtime := gtime + 1
8 |     modified := true
9 if not modified then return (∅)
10return ({(w,u)|(w,u)∈C, w≠u, w≠v)})

function add-constraint-ac|dc2i(c)
1 {u,v} := the variables constrained by c
2 C := C ∪ {(u,v),(v,u)}
3 propagate-ac3'({(u,v),(v,u)}, 0)
```

Figure 1. Constraint addition in AC|DC-2i.

As we mentioned, constraint retraction is done in three stages: initialization (**initialize-ac|dc2i**), propagation (**propagate-ac|dc2i**), and filtration (**propagate-ac3'**). We already explained filtration using AC-3. Just notice how the time is set in line 2 of **retract-constraint-ac|dc2i** so only the restored values will be checked for consistency during filtration.

During the initialization stage (**initialize-ac|dc2i**), the values that were removed due to the retracted constraint (line 3) are restored (the values for restoration are taken from the original domain D_0). The restored values are kept in a queue for the subsequent stage. Notice also, that this is the place where the restoration time is set (line 5).

During the propagation stage (**propagate-ac|dc2i**) we are restoring other values in the direct neighborhood of the variable whose domain has been extended. Assume that domain of variable *u* was extended and *v* is connected to *u* using a constraint. Then, we restore values *dv* in *v* that:

- were removed due to the constraint between *u* and *v* (found using the justification, line 7),
- have a supporter among the restored values of *u* (line 9),
- and this supporter has been deleted before *dv* (line 10).

```
function retract-constraint-ac|dc2i(c)
1 {u,v} := the variables constrained by c
2 retract_start := gtime
3 restored_u := initialize-ac|dc2i(u,v)
4 restored_v := initialize-ac|dc2i(v,u)
5 C := C − {(u,v),(v,u)}
6 revise:=propagate-ac|dc2i({restored_u,restored_v})
7 propagate-ac3'(revise, retract_start)

function initialize-ac|dc2i(u,v)
1 restored_u := ∅
2 for each d∈(D0[u]−D[u]) do
3 |  if justif_var[u,d] = v then
4 |    D[u] := D[u] ∪ {d}
5 |    restore_time[u,d] := gtime
6 |    gtime := gtime + 1
7 |    justif_var[u,d] := NIL
8 |    restored_u := restored_u ∪ {d}
9 return ((u,restored_u))

function propagate-ac|dc2i(restore)
1 revise := ∅
2 while restore not empty do
3 |  select and remove (u,restored_u) from restore
4 |  for each (u,v)∈C do
5 |  |  restored_v := ∅
6 |  |  for each dv∈(D0[v]−D[v]) do
7 |  |  |  if justif_var[v,dv] = u and
8 |  |  |   exists du∈restored_u s.t.
9 |  |  |    du is a support for dv wrt (u,v) and
10|  |  |    justif_time[v,dv]>justif_time[u,du] then
11|  |  |      D[v] := D[v] ∪ {dv}
12|  |  |      restore_time[v,dv] := gtime
13|  |  |      gtime := gtime + 1
14|  |  |      justif_var[v,dv] := NIL
15|  |  |      restored_v := restored_v ∪ {dv}
16|  |  if restored_v ≠ ∅ then
17|  |    restore := restore ∪ {(v,restored_v)}
18|  |    revise := revise ∪ {(v,w)|(v,w)∈C})
19return (revise)
```

Figure 2. Constraint retraction in AC|DC-2i.

The last condition above is fine tuning the condition from AC|DC-2 – now the deletion time of individual values is assumed rather than a minimum among the deletion times of all restored values in the variable. Because the new condition is stronger than the former one, fewer values are restored (but all values that are arc consistent with respect to the new problem are restored, see the next section). The propagation loop is repeated as long as any domain changes. Figure 3 presents a simplified example of three stages of constraint retraction (restoration time not shown – it is used within stage 3 only).

Note finally, that AC|DC-2i uses AC-3 as it is, only the justifications and timestamps should be updated upon value removal. Hence, AC-3 can be substituted by other arc consistency procedures, for example by using optimal algorithm AC-3.1 (Zhang and Yap, 2001) to get AC3.1|DC-2i. This change further improves time efficiency of the algorithm but it also increases space complexity due to additional structures required by AC-3.1 (and other optimal AC algorithms).

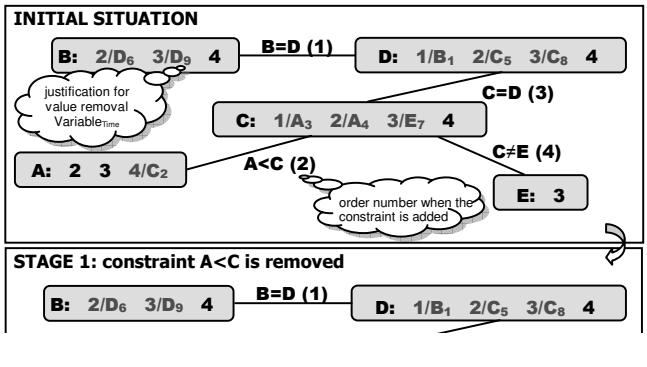

Figure 3. Example of constraint retraction.

Theoretical Analysis

In this section we will prove formally the correctness of AC|DC-2i algorithm and we will also describe time and space complexity of AC|DC-2i and AC3.1|DC-2i.

Soundness and Completeness

The correctness of the operation of constraint addition follows directly from the correctness of AC-3, thus extra argumentation is not necessary.

Assume now that constraint c_i is retracted from the problem P = (X, D, $\{c_1,c_2,\ldots,c_n\}$). We say that the operation of constraint retraction is correct if we obtain the maximally arc consistent problem for the problem P' = (X, D, $\{c_1,c_2,\ldots,c_{(i-1)},c_{(i+1)},\ldots,c_n\}$).

Proposition 1. Algorithm AC|DC-2i performs a correct retraction of a constraint with respect to maximal arc consistency.

Proof. To prove the proposition it is sufficient to verify that the algorithm restores all values that are necessary to be restored, that is the values that have to be present in the maximum arc consistent state of the new problem. If the algorithm restores some extra values, it does not matter because the final filtration stage will remove them. This fact directly follows from the correctness of AC-3.

Consider the following situation. We have a problem P, which is the result of addition of the constraints from the set $\{c_1,c_2,\ldots,c_n\}$, and we are retracting a constraint c_i, that restricts the variables *u* and *v*. As a reference we will use an auxiliary maximally arc consistent problem Q, which consists of the constraints $\{c_1,c_2,\ldots,c_{(i-1)},c_{(i+1)},\ldots,c_n\}$. Our task is to show that all the values that are present in the current domains of the variables of problem Q and are not present in the current domains of the variables of problem P will be restored by AC|DC-2i.

We will proceed by mathematical induction according to the removal time of the values. Let constraint c_i be added to problem P at time t_0. Next, let t_0+t_1 be the time when a value from a variable different from *u* or *v* has been removed for the first time (after time t_0). Thus the values removed from the problem in the time interval $\langle t_0, t_0+t_1\rangle$ came only from the current domains of the variables *u* and *v*. The reason for elimination of these values has been directly the constraint c_i together with the current state of given domains. All these values are restored within the initialization stage, because they have the opposite variable as their justification and so they satisfy the condition of restoration (line 3 in **`initialize-ac|dc2i`**).

Now, let us suppose that we need to restore a value *d* in the current domain of a variable *x* such that value *d* has been removed at time t_2, where $t_2>t_0+t_1$. It means that value *d* is present in the current domain of variable *x* in problem Q while this is not true in problem P. By induction hypothesis we can assume that all the values removed before time t_2 have already been tested for restoration. If these values were present in the current domains of problem Q they had been correctly restored in problem P. Before value *d* was removed from the current domain of variable *x* it must have lost all supports in some of the neighboring variables first. Suppose that *y* is such a variable with no support for *d*. It is clear that all supports for *d* were eliminated before time t_2 from the current domain of *y*. Value *d* is present in problem Q, thus there must be present also some support for *d* in the current domain of *y* in problem Q. By induction hypothesis such a support has been already restored and the restoration of their neighbors has been scheduled. Of course, variable *x* and its value *d* belong among these neighbors and therefore it is also scheduled for restoration. When the propagation process reaches the restoration of variable *x*, value *d* will be put back into the current domain of *x* since it satisfies all the conditions for restoration (lines 7-10 in **`propagate-ac|dc2i`**).

AC|DC-2i algorithm correctly restores the maximum arc consistency in the problem after constraint retraction.

q

Proposition 2. Algorithm AC|DC-2i performs at most as many steps as algorithms AC|DC and AC|DC-2.

Proof. The proposition directly follows from the correctness of AC|DC-2i and from the fact that a value has to fulfill more conditions in the AC|DC-2i algorithm than in AC|DC and AC|DC-2 before it is put back into the current domain of a variable. This theoretically shows that a propagation chain of the restoration stage is not longer in AC|DC-2i than in AC|DC and AC|DC-2.

q

Time and Space Complexity

The space complexity of AC|DC is $O(nd+e)$ where n is a number of variables, d is a size of domains of the variables, and e is a number of constraints (Berlandier and Neveu, 1994). The additional data structures (justifications, removal times, and restoration times) require additional space $O(nd)$ ($O(1)$ for every value) so the overall worst-case space complexity of AC|DC-2i is $O(nd+e)$, same as AC|DC and AC|DC-2. If we use AC-3.1 instead of AC-3' then we need to keep additional data structures for AC-3.1 so we obtain the worst-case space complexity $O(ed+nd)$ for AC3.1|DC-2i.

The worst-case time complexity of AC-3 (Mackworth, 1977) and hence AC-3' is $O(ed^3)$ which is the complexity of the filtration stage of AC|DC-2i. The worst-case time complexity of the initialization and the propagation stages together is $O(ed^2)$ because every pair of values in the domains of different variables constrained by a constraint is tested at most once there. Thus the overall worst-case time complexity of AC|DC-2i is $O(ed^3)$, same as AC|DC and AC|DC-2. If AC-3.1 is used instead of AC-3 then we get better worst-case time complexity of the filtration stage $O(ed^2)$ and hence the overall worst-case time complexity of AC3.1|DC-2i is $O(ed^2)$.

Experimental Results

We have implemented the proposed algorithms AC|DC-2i and AC3.1|DC-2i in C++ and compared them with existing dynamic arc consistency algorithms AC|DC, AC|DC-2, AC3.1|DC, and DnAC-6 on Random CSPs.

A Random Constraint Satisfaction Problem (RCSP) (MacIntyre et al, 1998) represents probably the most frequently used benchmark set in the area of constraint satisfaction. Each problem instance is characterized by a 4-tuple $\langle n, d, p_1, p_2\rangle$, where n is a number of variables, d is a uniform domain size, p_1 is a measure of the density of the constraint graph, and p_2 is a measure of the tightness of the constraints. We use a so called model B of a Random CSP where $p_1.n.(n\text{-}1)/2$ random pairs of variables are selected to form binary constraints and $p_2.d^2$ pairs of values are picked randomly as incompatible in each constraint.

We did the experiments with RCSP $\langle 100, 50, 0.5, p_2\rangle$, where p_2 was selected within the phase transition area (0.87–0.89). In each experiment, the best algorithm among AC|DC, AC|DC-2, AC3.1|DC, and DnAC-6 was chosen first and AC|DC-2i and AC3.1|DC-2i were compared to this best algorithm then. Note that AC|DC-2i always performed better than AC|DC-2. The experiments run on 2.4 GHz Pentium 4 with 512 MB RAM under Mandrake Linux 10 in the following way.

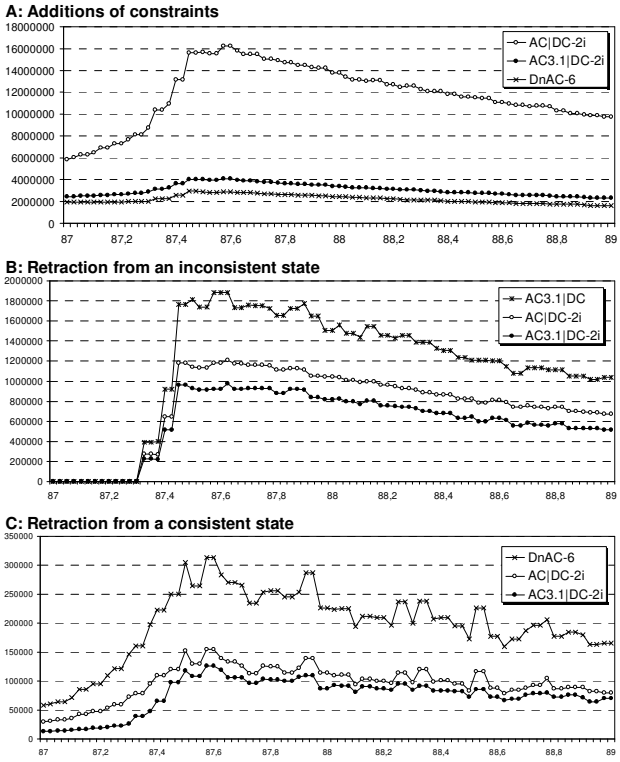

Figure 4. Number of constraint checks as a function of tightness in RCSP $\langle 100, 50, 0.5, p_2\rangle$.

First, the constraints were added to the problem one by one until a given density (0.5) has been reached (part A in graphs). If an inconsistent state has been reached during constraint addition then the constraint responsible for inconsistency has been removed (part B in graphs). When a consistent state has been reached (or after removing the constraint that caused inconsistency), we removed 10% of randomly selected constraints (part C in graphs). For each value of tightness we generated ten random problems. Mean values of the runtime (Figure 5) and the number of constraints checks (Figure 4) are presented here.

The experiments showed that for constraint retraction the proposed algorithm AC|DC-2i beats the so far fastest dynamic AC algorithms both in the number of constraint checks and in the runtime. The reason is that AC|DC-2i restores less wrong (inconsistent) values and it applies the AC procedure only to newly restored values. In case of constraint addition AC|DC-2i suffers from non-optimality of AC-3. As the experiments showed, this can be easily

improved by using AC-3.1 instead of AC-3 resulting in AC3.1|DC-2i.

To compare further AC|DC-2i and AC3.1|DC-2i algorithms we made another experiment where the memory consumption of the algorithms was measured. Again, we used Random CSP $\langle 100, d, 0.5, p_2 \rangle$, but we varied the size of the variables' domains as well as tightness.

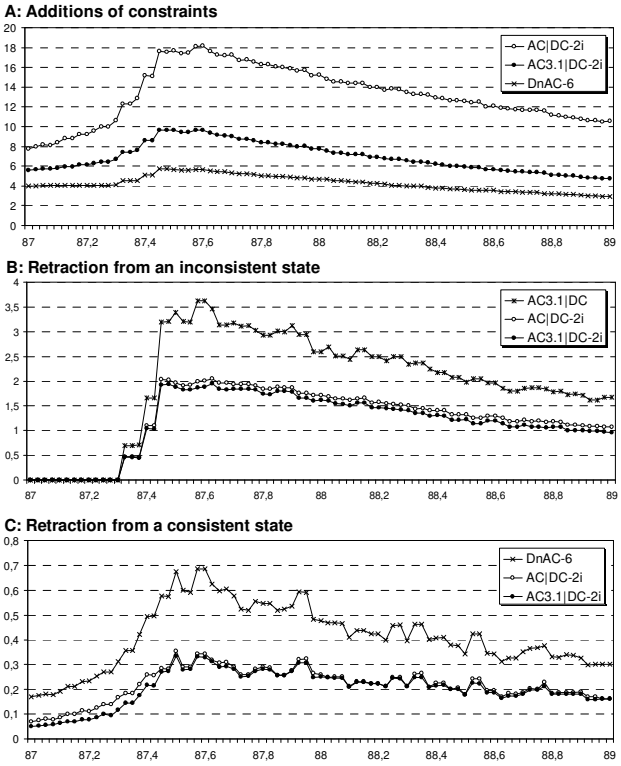

Figure 5. Runtime (in seconds) as a function of tightness in RCSP $\langle 100, 50, 0.5, p_2 \rangle$.

We were adding randomly generated constraints until we reached the inconsistent state when some of the domains became empty. We measured the memory consumption just before the constraint causing inconsistency was added. At that point the data structures stored the maximum number of records and so the memory consumption was the largest. Table 2 presents the memory consumption of data structures used by the compared algorithms (excluding the extensional representation of the constraints). As we can see, the memory consumption of AC|DC-2i is identical to AC|DC while AC3.1|DC-2i requires more memory comparable to AC-3.1|DC but still less than DnAC-6.

Domain size (d)	20	30	40	50	60	70	80	90
$100*p_2$	71	79	84	87	89	90	91	92
AC|DC	<1	<1	<1	<1	<1	<1	<1	<1
AC|DC-2	<1	<1	<1	<1	<1	<1	<1	<1
AC-3.1|DC	2	3	5	5	7	7	9	10
DnAC-6	2	4	6	7	9	10	12	13
AC|DC-2i	*<1*	*<1*	*<1*	*<1*	*<1*	*<1*	*<1*	*<1*
AC3.1|DC-2i	*2*	*3*	*5*	*5*	*7*	*7*	*9*	*10*

Table 2. Memory consumption (in MB) depending on the size of variable domains for RCSP $\langle 100, d, 0.5, p_2 \rangle$.

Conclusions

The paper presents an improved algorithm for maintaining arc consistency after constraint retracting and constraint addition. The algorithm builds on AC|DC-2 which the proposed improvements make the so far fastest dynamic arc consistency algorithm. Two versions of the algorithm are presented: the first one with AC-3 procedure that keeps low memory consumption, the second one with optimal AC-3.1 procedure that improves further the time complexity (especially for constraint addition) but also increases the memory consumption. As a consequence, the user may choose having either decent memory consumption (AC|DC-2i) or a better runtime (AC3.1|DC-2i). Note finally, that the proposed algorithm is not tightly integrated with the AC procedure so it can be easily extended to intentionally defined constraints as well as to n-ary constraints, provided that the underlying propagators give functionally similar to `filter-arc-ac3'` procedure namely updating the justifications and timestamps.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract No. 201/04/1102. We would like to thank anonymous reviewers for their valuable comments and corrections.

References

Berlandier P. and Neveu B., 1994. Arc-Consistency for Dynamic Constraint Satisfaction Problems: a RMS free approach. In *Proceedings of the ECAI-94 Workshop on "Constraint Satisfaction Issues Raised by Practical Applications"*, Amsterdam, The Netherlands.

Bessière Ch., 1991. Arc-Consistency in Dynamic Constraint Satisfaction Problems. In *Proc. of the 9th National Conference on Artificial Intelligence (AAAI-91)*, Anaheim, CA, USA. AAAI Press, 221–226.

Bessière Ch., 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65:179–190.

Debruyne R., 1996. Arc-Consistency in Dynamic CSPs is no more prohibitive. In *Proc. of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-96)*, Toulouse, France, 239–267.

Dechter R. and Dechter A., 1988. Belief Maintenance in Dynamic Constraint Networks. In *Proc. of the 7th National Conference on Artificial Intelligence (AAAI-88)*, St. Paul, MN, USA. AAAI Press, 37–42.

MacIntyre E., Prosser P., Smith B., and Walsh T., 1998. Random Constraint Satisfaction: theory meets practice. In Michael Maher and Jean-Francois Puget (eds.): *Principles and Practice of Constraint Programming (CP98)*. Springer-Verlag LNCS 1520, 325–339.

Mackworth A.K., 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8: 99–118.

Mohr R. and Henderson T.C., 1986. Arc and Path Consistency Revised. *Artificial Intelligence* 28: 225–233.

Mouhoub M., 2003. Arc Consistency for Dynamic CSPs. In Vasile Palade, Robert J. Howlett, Lakhmi C. Jain (Eds.): *Proceedings of the 7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems – Part I (KES 2003)*, Oxford, UK. Springer Verlag LNCS 2773, 393–400.

Surynek P. and Barták R., 2004. A New Algorithm for Maintaining Arc Consistency After Constraint Retraction. In Mark Wallace (ed.): *Principles and Practice of Constraint Programming (CP 2004)*. Springer-Verlag LNCS 3258, 767–771.

Verfaillie G. and Schiex T., 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, USA. AAAI Press, 307–312.

Wallace R. and Freuder E., 1998. Stable Solutions for Dynamic Constraint Satisfaction Problems. In Michael Maher and Jean-Francois Puget (eds.): *Principles and Practice of Constraint Programming (CP98)*. Springer-Verlag LNCS 1520, 447–461.

Zhang Y. and Yap R., 2001. Making AC-3 an Optimal Algorithm. In *Proceedings of International Joint Conference in Artificial Intelligence (IJCAI-01)*, 316–321.