A New Algorithm for Maintaining Arc Consistency after Constraint Retraction

Pavel Surynek, Roman Barták[*]

Charles University in Prague, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
pavel.surynek@seznam.cz, roman.bartak@mff.cuni.cz

Abstract. Dynamic Constraint Satisfaction Problems play a very important role in modeling and solving real-life problems where the set of constraints is changing. The paper addresses a problem of maintaining arc consistency after removing a constraint from the constraint model. A new dynamic arc consistency algorithm is proposed that improves the practical time efficiency of the existing AC|DC algorithm by using additional data-structures. The algorithm achieves real time efficiency close to the so far fastest DynAC-6 algorithm while keeping the memory consumption low.

1 Introduction

For solving many real-life problems, the traditional static formulation of the constraint satisfaction problem (CSP) is not sufficient because the problem formulation is changing dynamically. Assume for example the problem of train scheduling [6] where the system must respond to train delays or the university timetabling problem [1,16] where the system must accommodate the changes proposed by the teachers. There exist many other dynamic problems requesting changes in the problem formulation to which the system must react by modifying the existing solution or proposing a new solution [9,10,11]. To model such problems Dechter and Dechter [8] proposed a notion of *Dynamic Constraint Satisfaction Problem* (DCSP) that is a sequence of static CSPs, where each CSP is a result of addition or retraction of a constraint in the preceding problem.

Several techniques have been proposed to solve Dynamic CSPs, including searching for robust solutions that are valid after small problem changes [18], searching for a new solution that minimizes the number of changes from the original solution [1,10], reusing the original solution to produce a new solution [17], 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 can only be pruned. Existing arc consistency algorithms are usually

[*] Corresponding author.

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.

In this paper we address the problem of maintaining maximal arc consistency after constraint retraction. The straightforward method how to do it is to restore the original domains for all the variables and then to call some arc consistency algorithm that removes the inconsistent values. However, this method is very inefficient because it repeats many constraint checks that were already tested in the previous runs. Therefore more efficient algorithms were proposed to exploit the information that the problem was arc consistent before the constraint retraction. For example, DnAC-4 [3] and DnAC-6 [7] are based on extended data structures of the underlying AC algorithm, namely AC-4 [14] and AC-6 [4]. These algorithms are very fast because they minimize the number of constraint checks but they are also memory consuming and complicated for implementation. To keep low space complexity, the AC|DC algorithm [2] was proposed as a reverse version of the popular AC-3 algorithm. Like AC-3, AC|DC can also be easily extended to non-binary constraints. However, AC|DC is not as time efficient as DnAC-6, the so far fastest dynamic arc consistency algorithm.

When exploring the behavior of AC|DC we have found out that the main reason for its inefficiency is restoring too many values in the domains that are immediately pruned in the completion stage of the algorithm when calling AC-3. Via using some additional data structures, similar to DnAC, that keep information about the reason of value removal, we can improve the time efficiency of AC|DC by restoring only the most promising values. The hope is that if fewer values are restored then fewer values will be pruned during the subsequent run of the AC-3 algorithm. This should improve the practical time efficiency of the algorithm. We call the resulting algorithm AC|DC-2 and we will show by an experimental evaluation that AC|DC-2 improves significantly the practical time efficiency of AC|DC without increasing much the space complexity. Actually, the practical time efficiency of AC|DC-2 is comparable to DnAC-6 while the memory consumption of AC|DC-2 is much smaller than this of DnAC-6.

The paper is organized as follows. First, we will describe the basic terminology followed by a survey of existing approaches to maintaining dynamic arc consistency. Then, we will introduce the AC|DC-2 algorithm, we will prove its correctness, and we will present its theoretical worst-case time and space complexity. The paper is concluded by an experimental comparison of AC|DC-2 to AC|DC and DnAC-6.

2 Preliminaries

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 the variables x_i and x_j is a subset of the Cartesian product $D_i \times D_j$. The value *a* of the variable x_i is *arc consistent* (AC) if and only if for each variable x_j connected to x_i by the constraint c_{ij}, there exists a value $b \in D_j$ such that $(a,b) \in c_{ij}$. The CSP is *arc consistent* if and only if every value of every variable is arc consistent. The CSP is *maximally arc consistent* if every arc consistent value stays in the respective domain. 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 $P_0, P_1, \dots, P_n$, where each P_i is a CSP resulting from the 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 the problem P_i maximally arc consistent using the information that the problems $P_0, P_1, \dots, P_{i-1}$ are maximally arc consistent.

3 Related Works

Arc consistency (AC) is the most frequently used consistency technique thanks to its good ratio between the number of removed inconsistencies and the time and space complexity. Many AC algorithms have been proposed to make the constraint satisfaction problem arc consistent. Among them, AC-3 [13] is the most popular algorithm that is implemented in many existing constraint solvers. There are three reasons for popularity of AC-3: the algorithm is easy to implement, it can be naturally extended to non-binary constraints, and its practical time and space efficiency is very good despite the fact that AC-3 is not a (worst-case) time optimal AC algorithm. AC-4 [14] was the first AC algorithm with the optimal worst-case time complexity. However, the average time complexity of AC-4 is not as good and moreover, its space complexity is quite large. Therefore, another optimal algorithm called AC-6 [4] has been proposed to decrease the memory consumption and to improve the average time efficiency. Recently, two new versions of AC-3 algorithm, AC-3.1 [19] and AC-2001 [5], have been independently proposed to achieve the optimal worst-case time complexity without the complex data structures typical for AC-4 and AC-6.

All above described AC algorithms can be used when the constraints are incrementally added to the problem. However, the algorithms should be modified to work effectively with constraint retraction. DnAC-4 [3] was one of the first dynamic arc-consistency algorithms. As the name indicates, the algorithm is based on AC-4 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. This data structure keeps a link to the variable that caused deletion of a value from the variable domain during the AC domain pruning. Constraint addition in DnAC-4 is realized via the AC-4 algorithm; the only difference is that justifications are being computed there. The constraint retraction is realized in three steps. First, the values deleted due to the retracted constraint are added back to the domains. Second, the extension of domains is propagated to other domains and selected values are added back to them. Justifications are used there to minimize the number of restored values, for details see [3]. Because, the values that are possibly inconsistent might also be recovered, the third step ensures that only the arc consistent values stay in the domains by calling AC-4. DnAC-4 inherits the disadvantages of AC-4 and therefore DnAC-6 [7] has been proposed to improve memory consumption and average time complexity. DnAC-6 uses the same principles like DnAC-4 but it is integrated with the AC-6 algorithm rather than using AC-4. DnAC-6 is currently the fastest dynamic arc consistency algorithm but it has the disadvantage of the fine grained consistency algorithms which is a large space complexity. Moreover, implementation of DnAC-4 and DnAC-6 is quite complicated.

AC|DC [2] was built on different foundations than DnAC algorithms; in particular AC|DC does not use the supporting data structures and hence its space complexity is very low. AC|DC is built around the AC-3 algorithm and, actually, constraint addition is realized there via the original AC-3 algorithm. Like in DnAC, constraint retraction in AC|DC is realized in three steps: recovery of the values deleted due to the retracted constraint, propagation of these domain extensions to other variables, and removal of the inconsistent values. Propagation of domain extensions is realized via an inverted AC-3 procedure that does not use any additional data structures which keeps memory consumption low. On the other hand, many values that are not arc consistent are recovered in the second step and immediately deleted in the third step. This makes the algorithm less time efficient especially in comparison to DnAC-6. However, unlike DnAC-6 and DnAC-4, AC|DC is easily extendible to non-binary constraints without large memory consumption. Paper [15] proposed an improvement of the time complexity for AC|DC by using the optimal AC-3.1 algorithm instead of AC-3. The resulting algorithm is called AC-3.1|DC. The theoretical study and the experimental results showed that the time and space efficiency of AC-3.1|DC is comparable to DnAC-6.

Table 1. Time and space complexity of existing dynamic arc consistency algorithms.

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

4 Algorithm AC|DC-2

In this section we propose a new algorithm for maintaining dynamic arc consistency that improves the way of domain restoration in AC|DC. As in the case of AC|DC algorithm [2] the constraint retraction using AC|DC-2 is carried out in three phases. In the first (initialization) phase the algorithm puts back the values into the variable domains that have been removed directly because of the retracted constraint. The second (propagation) phase consists of a propagation of the initial restorations from the first phase. Finally, in the last (filtering) phase the algorithm removes the inconsistent values that are the values that have been incorrectly restored in the previous phases.

The key idea of our algorithm is to exploit a certain type of information recorded during the filtering phase in order to perform the constraint retraction more effectively. The recorded information allows us to determine more accurately what values worth consideration whether to be put back into variable domains during the constraint relaxation process. In this way we reduce the number of incorrect value additions and consequently its ill-fated propagations and final filtrations.

More precisely, we extend the original AC|DC algorithm with additional data structures that record a justification and a current “time” for every value eliminated from the variable domain. By the *justification* we mean the first neighboring variable in which the eliminated value lost all supports. A similar data structure is also used within DnAC-4 [3] and DnAC-6 [7] algorithms. To model the time we use a global counter which is incremented after every manipulation of the variable domains. When a constraint is retracted from the problem the algorithm uses the justifications and removal times to determine the set of values which have been removed possibly because of the retracted constraint and that should be restored in the relaxed problem.

4.1 Algorithm

In this section we will describe the AC|DC-2 algorithm formally. The first step towards the new algorithm is the slightly modified AC-3 algorithm which is used by AC|DC-2 as a procedure for the reestablishment of arc-consistency. The pseudo code of the modified AC-3 that we call AC-3’ is shown in Figure 1.

```
function propagate-ac3'(P, data, revise)
  queue := revise
  while queue not empty do
    select and remove a constraint c from queue
    {u,v} := the variables constrained by c
    (P,data,revise_u) := filter-arc-ac3'(P, data, c, u, v)
    (P,data,revise_v) := filter-arc-ac3'(P, data, c, v, u)
    queue := queue ∪ revise_u ∪ revise_v
  return (P,data)

function filter-arc-ac3'(P, data, c, u, v)
  modified := false
  for each d in P.D[u] do
    if d has no support in P.D[v] w.r.t. c then
      P.D[v] := P.D[v] - {d}
      data.justif[u,d].var := v
      data.justif[u,d].time := data.time
      data.time := data.time + 1
      modified := true
  if not modified then
    return (P,data,Ø)
  return (P,data,{e in P.C|u is constrained by e and e≠c})
```

Fig. 1. A pseudo code of the modified arc-consistency algorithm AC-3’

The AC-3' algorithm consists of two functions. The first function `propagate-ac3'` repeatedly revises the constraints till an arc-consistent state is reached. The revision of the constraint is performed by the function `filter-arc-ac3'`. This function removes the values which are not consistent with a given constraint. The function is called separately for both directions of the constraint. That is, if the constraint binds two variables, say *u* and *v*, the function has to be called separately for both arcs (*u*,*v*) and (*v*,*u*). The main difference between the original AC-3 and our modified version is that we record the justification and the current time for every value eliminated from the variable domain.

Both functions, which the algorithm consists of, get parameters named `P` and `data`. The parameter `P` represents the input constraint satisfaction problem. `P` is a compound structure – it consists of a set of variables denoted by `P.V`, a set of constraints denoted by `P.C`, and an array denoted by `P.D`, which represents the current domains of variables. In the following algorithm we will also need to access the original (non-pruned) variable domains that will be denoted by $\texttt{P.D}_0$. The second parameter `data` holds the additional data structures used during domain restoration. It is again a compound data structure. The component denoted by `data.justif` represents an array, which stores the justifications and removal times of every value eliminated from the current domains. The second component denoted by `data.time` represents the global counter, which we are using to model the time.

If we want to make a given constraint satisfaction problem `P` arc-consistent then we call the program by `propagate-ac3'(P, data, P.C)`. Thus all the constraints occurring within the problem are subjected to revision by the AC-3' algorithm.

Now all the ingredients for AC|DC-2 are ready, so we can describe the algorithm itself. The pseudo code of both operations provided by the AC|DC-2 algorithm – an addition and a retraction of a constraint – is shown in Figure 2. For the addition of a constraint the function `add-constraint-ac|dc2` is used. The operation of constraint retraction is realized by the function `retract-constraint-ac|dc2`.

The addition of a constraint using AC|DC-2 is straightforward. Simply, the new constraint is added to the problem and its revision by AC-3' is performed subsequently. The result of these two steps is an arc-consistent problem enlarged by the new constraint.

The retraction of a constraint is a more difficult operation. One must be aware of the fact that the constraint retraction causes a relaxation of the restrictions and therefore it would be necessary to put back some values into the current domains to restore the maximum arc-consistency after removing the constraint. As we said at the beginning of this section, within the AC|DC-2 algorithm this is done in three phases.

In the first phase the algorithm performs the initial restoration of the missing values in the current domains of variables constrained by a retracted constraint. This phase is carried out by the function `initialize-ac|dc2`. We restore the values, which were removed directly because of the retracted constraint. In other words, if the constraint binds the variables *u* and *v*, we restore the values in the current domain of *u* that were eliminated because of the fact that they lost all supports in the current domain of *v*. This step is, of course, done for both directions of the constraint, that is separately for arcs (*u*,*v*) and (*v*,*u*). As it is described in the pseudo code (Figure 2), the absent value has to fulfill two criteria before it is put back into the domain. The justification for the value has to be the opposite variable with respect to the retracted constraint and the value has to have no support in the current domain of the opposite variable (line 4 in `initialize-ac|dc2`).

After the initialization phase the algorithm proceeds with the propagation phase, in which restorations from the initialization phase are propagated into other variables. This step is realized by the function `propagate-ac|dc2`. When a set of values is restored in the current domain of some variable, the algorithm schedules the restoration of the current domains of the neighboring variables. The value will be put back into the current domain of the neighboring variable if a new support of the value is among the values restored in the previous step (line 11 in `propagate-ac|dc2`). Before a search for a support is started, the value is tested whether it was removed due to the previously restored variable (line 9) and whether there exists a chance that a new support may exist among the previously restored values (line 11). Moreover the value has to pass an additional test on removal time comparison (line 10). The removal time of the value has to be greater than the smallest removal time of the values restored in previous step. If both tests (lines 9,10) are passed successfully, it is possible that the value was removed due to the absence of the previously restored values and it should be also restored. In this case, the algorithm proceeds with the search for a support for the absent value among the restored values with respect to a given connecting constraint (line 11). The full argumentation of why this approach works and why it is correct is given in the next section.

Note that these two additional tests (lines 9,10) make AC|DC-2 different from the original AC|DC. These tests reduce the number of constraint checks as well as the total number of incorrectly restored values and subsequently their ill-fated propagations. The consequence of this approach is a shorter running time of both the propagation and the filtration phases.

When the propagation phase is finished the algorithm continues with the filtration phase. At this phase the algorithm eliminates the values that were incorrectly restored, that is the values that did not gain another necessary supports in the current domains of their neighboring variables. This step is carried out at line 8 of the function `retract-constraint-ac|dc2`.

During the whole process of propagation we collect all the constraints that restrict the variables for which the domain has been changed. At the end of the restoration phase we subject all these constraints to revision by the AC-3’ procedure.

The result of the above described process is a maximum arc-consistent problem from which a constraint was retracted. We will prove this claim formally in the next section.

```
function add-constraint-ac|dc2(P, data, c)
  P.C := P.C ∪ {c}
  (P,data) := propagate-ac3'(P, data, {c})
  return (P,data)

function retract-constraint-ac|dc2(P, data, c)
  (P,data,restored_u) :=
      initialize-ac|dc2(P, data, c, u, v)
  (P,data,restored_v) :=
      initialize-ac|dc2(P, data, c, v, u)
  P.C := P.C - {c}
  (P,data,revise) :=
      propagate-ac|dc2(P, data,{restored_u,restored_v})
  (P,data) := propagate-ac3'(P, data, revise)
  return (P,data)

function initialize-ac|dc2(P, data, c, u, v)
  restored_u := ∅
  time_u := ∞
  for each d in (P.D$_0$[u] - P.D[u]) do
    if data.justif[u,d].var = v then
      P.D[u] := P.D[u] ∪ {d}
      data.justif[u,d].var := NIL
      restored_u := restored_u ∪ {d}
      time_u := min(time_u, data.justif[u,d].time)
  return (P,data,(u,time_u,restored_u))

function propagate-ac|dc2(P, data, restore)
  revise := ∅
  while restore not empty do
    select and remove (u,time_u,restored_u) from restore
    for each c in P.C|u is constrained by c do
      {u,v} := the variables constrained by c
      restored_v := ∅
      time_v := ∞
      for each d in (P.D$_0$[v] - P.D[v]) do
        if data.justif[v, d].var = u then
          if data.justif[v, d].time > time_u then
            if d has a support in restored_u w.r.t. c then
              P.D[v] := P.D[v] ∪ {d}
              data.justif[v,d].var := NIL
              restored_v := restored_v ∪ {d}
              time_v := min(time_v, data.justif[v,d].time)
      restore := restore ∪ {(v,time_v,restored_v)}
    revise := revise ∪ {e in P.C|u is constrained by e})
  return (P,data,revise)
```

Fig. 2. A pseudo code of the dynamic arc-consistency algorithm AC|DC-2

4.2 Correctness

In this section we will prove formally the correctness of the idea which the AC|DC-2 algorithm is based on. The correctness of the operation of constraint addition follows directly from the correctness of the AC-3 algorithm, thus extra argumentation is not necessary.

Before we start with the proof the correctness of the constraint retraction, we first have to describe what it means to correctly retract a constraint from a problem with respect to maximal arc-consistency. Let us consider the empty problem (that is, the problem which consists of several variables but which does not contain any constraint) into which we will add one by one the constraints from the set $\{c_1,c_2,\ldots,c_n\}$ using the operation of constraint addition. At the end we will obtain a maximally arc-consistent problem containing the constraints from the set $\{c_1,c_2,\ldots,c_n\}$. Now, the result of a correctly finished retraction of a constraint, say c_i, is the problem with the same current domains of the variables as if it was constructed from the empty problem by the addition of the constraints from the set $\{c_1,c_2,\ldots,c_{(i-1)},c_{(i+1)},\ldots,c_n\}$. Notice that the constraint c_i is missing in this set.

Proposition 1. The algorithm AC|DC-2 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 the values that are necessary to be restored, that is the values that has 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 phase will remove them. This fact directly follows from the correctness of the AC-3 consistency algorithm.

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 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 the problem Q and are not present in the current domains of the variables of the problem P will be restored by the algorithm AC|DC-2.

We will proceed by mathematical induction according to the removal time of the values. Let the constraint c_i has been added to the 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 the 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 phase of the constraint retraction operation, particularly within the function `initialize-ac|dc2`. Every value removed in the time interval $\langle t_0, t_0+t_1\rangle$ has the opposite variable as its justification with respect to c_i and has no support in the current domain of the opposite variable at the time between t_0 and t_0+t_1 and thus also at the time when the constraint is retracted. This is just what is tested in the function `initialize-ac|dc2` before the value is put back into the domain. This concludes the initial induction step.

Now, let us suppose that we need to restore a value *d* in the current domain of a variable *x*. In fact it means that the value *d* is present in the current domain of the variable *x* in the problem Q while this is not true in the problem P. To continue by induction let us suppose furthermore that the value *d* has been removed at time t_2, where $t_2 > t_0 + t_1$. By induction hypothesis we know that all the values removed before the time t_2 have been already tested for restoration. If these values were present in the current domains of the problem Q they had been correctly restored in the problem P. Before the value *d* was removed from the current domain of the variable *x* it must have lost all supports in some of the neighboring variables first. Suppose that *y* is the variable with no support for the value *d*. It is clear that all supports for the value *d* were eliminated before the time t_2 from the current domain of *y*. The value *d* is present in the problem Q, thus there must be present also some supports for *d* in the current domain of *y* in the problem Q. By induction hypothesis these supports have been already restored and the restoration of their neighbors has been scheduled. Of course, the 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 the variable *x*, the value *d* will be put back into the current domain of *x* since it satisfies all necessary conditions for restoration.

Now the proof is finished and we can conclude that the AC|DC-2 algorithm correctly restores the maximum arc-consistency in the problem after the retraction of a constraint.

❑

Proposition 2. The algorithm AC|DC-2 performs at most as many steps as the existing algorithm AC|DC.

Proof. The proposition directly follows from the correctness of AC|DC-2 and from the fact that a value has to fulfill more conditions in the AC|DC-2 algorithm than in AC|DC before it is put back into the current domain of a variable. This theoretically shows that a propagation chain of the restoration phase is shorter in the AC|DC-2 algorithm than in AC|DC.

❑

4.3 Time and Space Complexity

The space complexity of the operations for constraint addition and constraint retraction is $O(nd+e)$ where *n* is the number of variables, *d* is the size of the domains of variables, and *e* is the number of constraints. This is the space complexity of the AC|DC algorithm [2]. The additional data structures (justifications and removal times) requires a space of $O(1)$ for every value in the variable domains. Exactly, the additional space of $O(1)$ is required for every value that is not currently present in the domain of a variable. Thus, the additional data structures require $O(nd)$ space which is included in the above total space complexity.

The worst case time complexity of the operation of constraint addition directly corresponds with the time complexity of the AC-3' algorithm. It is well known that the worst case time complexity of this algorithm is $O(ed^3)$ [13]. The worst case time complexity of the initialization and the propagation phases of the operation of constraint retraction together is $O(ed^2)$. This result is easily to see if we realize that every pair of values in the domains of different variables constrained by a constraint is tested at most once. The overall time complexity is $O(ed^3)$ because of the final filtration phase which calls the AC-3' procedure.

5 Experimental Results

In order to show the practical efficiency of our new algorithm AC|DC-2, we have implemented the algorithm in C++ as a part of our experimental library for working with Dynamic CSPs. For comparison we have also implemented the algorithms AC|DC and DnAC-6. We performed the experiments on a set of randomly generated binary constraint satisfaction problems and we measured the number of constraint checks, the overall time, and the memory consumption. The experiments run under Red Hat Linux 9.0 on 2 GHz Pentium 4 with 512 MB of memory.

Random Binary Constraint Satisfaction Problems represent probably the most frequently used benchmark set in the area of constraint satisfaction. Each problem instance is characterized by a tuple $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d is the 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 A [12] of Random CSP where a pair of variables is selected randomly with the probability p_1 to form a binary constraint and a pair of values is picked randomly with the probability p_2 as incompatible. Actually, we used a model with a complementary tightness, so instead of selecting incompatible pairs of values with the probability p_2, we selected the compatible pairs with the probability $(1–p_2)$.

In the first set of experiments we measured the practical speed of the algorithms on random CSP $\langle 100, 50, 0.3, 1–p_2 \rangle$ for p_2 in the range 0.05 – 0.32, where the phase transition is located. Constraints were added incrementally to the problem by the operation of constraint addition until a given density was reached or an inconsistent state was encountered (a variable with an empty domain appeared). After this step, 10% of randomly selected constraints were retracted from the problem by the operation of constraint retraction. We measured the number of constraint checks (Figure 3) and the overall time to perform all these operations (Figure 4). For each problem instance, ten random problems were generated and the mean values are presented here.

The experiments confirmed our expectation that AC|DC-2 performs much less constraint checks than AC|DC even if the number of constraint checks is still higher than for DnAC-6. In terms of runtime, the results are even more encouraging because the runtime of AC|DC-2 is very close to the runtime of DnAC-6 despite the worse theoretical time complexity. Of course, the runtime of AC|DC-2 is much smaller than the runtime of AC|DC thanks to fewer constraint checks.

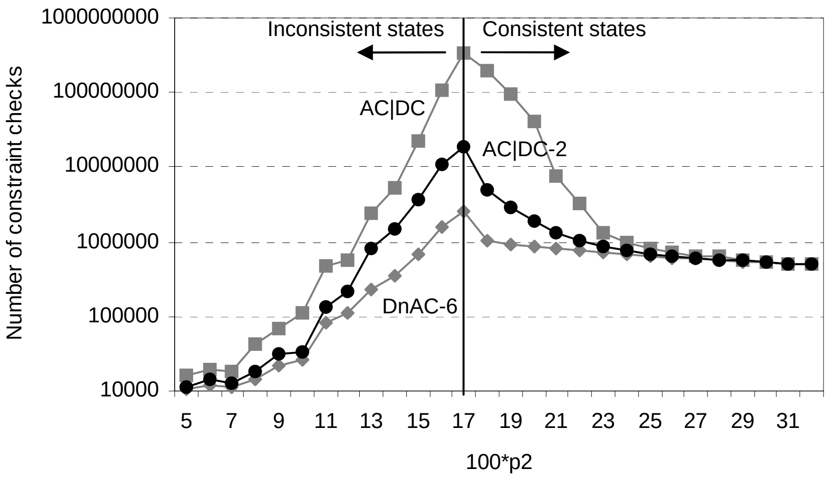

Fig. 3. Number of constraint checks (a logarithmic scale) as a function of tightness for random constraint satisfaction problems ⟨*100*, *50*, *0.3*, 1-p_2⟩.

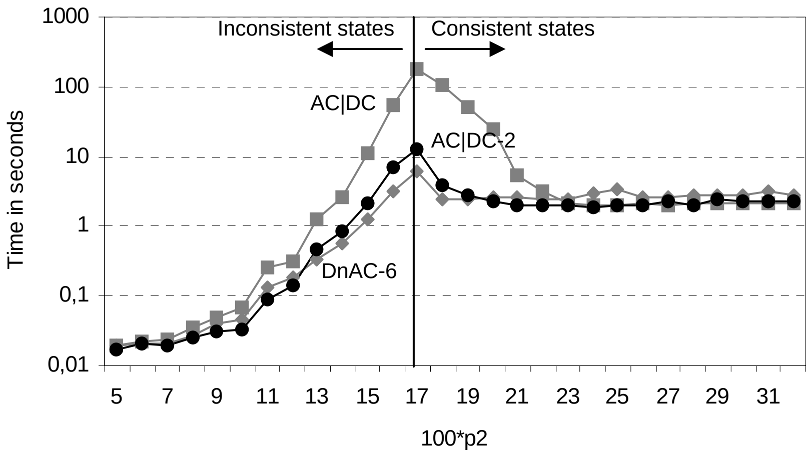

Fig. 4. Runtime (logarithmic scale in seconds) time as a function of tightness for random constraint satisfaction problems ⟨*100*, *50*, *0.3*, 1-p_2⟩.

In the second set of experiments we measured the memory consumption of the algorithms. We use the same random CSP ⟨*100*, *d*, *0.3*, 1–p_2⟩ but now we vary the size of variable domains. Recall that the problems are not static and the set of constraints is changing. So we were adding randomly selected constraints until we reached the inconsistent state when some of the domains became empty. We measure 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. In Table 2, we present the total memory consumption for AC|DC, DnAC-6, and AC|DC-2 that includes the memory consumption of the extensional representation of the constraints

Table 2. Memory consumption depending on the size of variable domains for random constraint satisfaction problems ⟨*100*, *d*, *0.3*, 1–p_2⟩.

Domain size (*d*)	**20**	**25**	**30**	**35**	**40**	**45**	**50**	**55**	**60**	**65**	**70**	**75**	**80**
100*p_2	**32%**	**26%**	**23%**	**22%**	**20%**	**19%**	**18%**	**17%**	**16%**	**16%**	**15%**	**15%**	**15%**
AC|DC	20MB	22MB	27MB	38MB	44MB	51MB	60MB	66MB	72MB	87MB	90MB	110MB	127MB
DnAC-6	41MB	48MB	59MB	80MB	93MB	106MB	124MB	137MB	149MB	174MB	184MB	217MB	247MB
AC|DC-2	20MB	22MB	27MB	38MB	44MB	51MB	60MB	66MB	72MB	87MB	90MB	110MB	127MB

The results showed that the memory consumption of AC|DC-2 was comparable to AC|DC; just in the couple of problems it was slightly larger. Moreover, the memory consumption of AC|DC-2 is much smaller than for DnAC-6. Note finally, that a significant portion of the memory consumption is due to the extensional representation of the constraints, in which every binary constraint is represented by a list of all compatible pairs of values, that is, by the pairs of values satisfying the given constraint.

5 Conclusions

In the paper, we proposed a new algorithm AC|DC-2 for maintaining arc consistency in dynamic environments where the constraints are added and retracted incrementally. Our main goal was to improve the time efficiency of the algorithm AC|DC while keeping its low space complexity. Rather than just switching the AC-3 algorithm in the AC|DC scheme for a more efficient AC-3.1 algorithm like AC-3.1|DC did, we accompanied the AC|DC algorithm by additional data structures that helped us to decrease the number of initially restored values. Consequently, we improved significantly the practical time efficiency of the AC|DC algorithm and the time is now comparable to the so far fastest DnAC-6 algorithm. Moreover, the additional data structures did not increase much the space complexity of AC|DC which remains much smaller than for DnAC-6. Moreover, AC|DC is easier to implement than DnAC-6.

We did not perform yet the detail comparison to the new AC-3.1|DC algorithm. According to the results presented in [15], we expect similar time efficiency because AC-3.1|DC is comparable to DnAC-6. Practical space complexity of AC-3.1|DC was not compared to AC|DC in [15] but we expect that AC|DC-2 will consume less memory than AC-3.1|DC because AC|DC-2 does not use the additional data structures necessary for AC-3.1. Combining the ideas of AC|DC-2 with the optimal AC-3 based arc consistency algorithms like AC-3.1 or AC-2001 might be a promising direction of the future research. In any case, using AC-3.1 instead of AC-3 in AC|DC-2 will improve the theoretical time complexity of AC|DC-2 to $O(ed^2)$.

Acknowledgements

This work is supported by the Czech Science Foundation under the contract No. 201/04/1102.

References

1. Barták, R.; Müller, T. and Rudová, H.: A New Approach to Modelling and Solving Minimal Perturbation Problems. In Recent Advances in Constraints, 2003. Springer-Verlag LNAI 3010, 2004. To appear.
2. Berlandier, P. and Neveu, B.: 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, 1994.
3. Bessière Ch.: Arc-Consistency in Dynamic Constraint Satisfaction Problems. In Proc. of the 9th National Conference on Artificial Intelligence (AAAI-91), Anaheim, CA, USA. AAAI Press, 1991, 221–226.
4. Bessière Ch.: Arc-consistency and arc-consistency again. Artificial Intelligence 65, 1994, 179–190.
5. Bessière, Ch. and Régin, J.-Ch.: Refining the Basic Constraint Propagation Algorithm. In Proceedings of International Joint Conference in Artificial Intelligence (IJCAI-01), 2001, 309–315.
6. Chiu, C.; Chou, C.; Lee, J.; Leung and Leung Y.: A Constraint-Based Interactive Train Rescheduling Tool. In Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP-96), Cambridge, MA, USA. Springer Verlag LNCS 1118, 1996, 104–118.
7. Debruyne R.: 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, 1996, 239–267.
8. Dechter R. and Dechter A.: Belief Maintenance in Dynamic Constraint Networks. In Proc. of the 7th National Conference on Artificial Intelligence (AAAI-88), St. Paul, MN, USA. AAAI Press, 1988, 37–42.
9. Drabble, B. and Najam-ul Haq: *Dynamic Schedule Management: Lessons from the Air Campaign Planning Domain*. In Pre-proceedings of the Sixth European Conference on Planning (ECP-01), 2001, 193–204.

10. El Sakkout, H.; Richards, T. and Wallace, M.: *Minimal Perturbation in Dynamic Scheduling*. In Henry Prade (editor): 13th European Conference on Artificial Intelligence (ECAI-98). John Wiley & Sons, 1998.
11. Kocjan W.: *Dynamic scheduling: State of the art report*, SICS Technical Report T2002:28, ISSN 100-3154, 2002.
12. MacIntyre, E.; Prosser, P.; Smith, B. and Walsh,T.: 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, 1998, 325–339.
13. Mackworth, A.K.: Consistency in Networks of Relations. In Artificial Intelligence 8, 1977, 99–118.
14. Mohr R., Henderson T.C.: Arc and Path Consistency Revised. Artificial Intelligence 28, 1986, 225–233.
15. Mouhoub, M.: 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, 2003, 393–400.
16. Rudová, H. and Murray, K.: *University Course Timetabling with Soft Constraints*. In Edmund Burke and Patrick De Causmaecker (eds.): Practice And Theory of Automated Timetabling IV. Springer-Verlag LNCS 2740, 2003, 310–328.
17. Verfaillie G. and Schiex T.: Solution Reuse in Dynamic Constraint Satisfaction Problems. In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, WA, USA. AAAI Press, 1994, 307–312.
18. Wallace, R. and Freuder E.: Stable Solutions for Dynamic Constraint Satisfaction Problems. In Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP-98). Pisa, Italy, Springer Verlag, 1998, 447–461.
19. Zhang, Y. and Yap, R.: Making AC-3 an Optimal Algorithm. In Proceedings of International Joint Conference in Artificial Intelligence (IJCAI-01), 2001, 316–321.