

Maintaining Arc Consistency in Non-Binary Dynamic CSPs using Simple Tabular Reduction

Matthieu Quéva^{1,2} and Christian W. Probst¹ and Laurent Ricci²

Abstract. Constraint Satisfaction Problems (CSPs) are well known models used in Artificial Intelligence. In order to represent real world systems, CSPs have been extended to Dynamic CSPs (DCSPs), which support adding and removing constraints at runtime. Some approaches to the NP-complete problem of solving CSPs use filtering techniques such as arc consistency, which also have been adapted to handle DCSPs with binary constraints. However, there exists only one algorithm targeting non-binary DCSPs (*DnGAC4*). In this paper we present a new algorithm *DnSTR* for maintaining arc consistency in DCSPs with non-binary constraints. Our algorithm is based on Simple Tabular Reduction for Table Constraints, a technique that dynamically maintains the tables of supports within the constraints. Initial results show that our algorithm outperforms *DnGAC4* both for addition and removal of constraints.

1 INTRODUCTION

One of the many interesting topics in Artificial Intelligence is the use of Constraint Programming to solve combinatorial problems. Those problems are usually modeled as Constraint Satisfaction Problems (CSPs), where the number of constraints in the system is known beforehand. However, many problems involve dynamic behaviours: user requirements may for example evolve during interactive design, or the set of tasks to be performed in a scheduling problem may be modified due to external factors. Dynamic Constraint Satisfaction Problems (DCSPs) have thus been introduced to permit the addition and removal (or retraction) of constraints at runtime.

The task of finding solutions in CSPs is NP-complete, and thus solvers usually apply filtering techniques to reduce the search space and simplify the problem. Filtering techniques can also be used in interactive problems, such as product configuration, in order to reduce the set of values available for each variable. The most used techniques are the ones achieving arc consistency.

There has been a considerable amount of work in designing arc consistency algorithms for CSPs with binary constraints. Among them are the well-known AC3 [10], AC4 [12] and AC6 [6]. Most authors have generally not extended their work to achieve arc consistency on non-binary problems, as any non-binary CSP can potentially always be transformed into an equivalent binary CSP, although it is sometimes impracticable [1]. However, non-binary constraints have been proved to be practically relevant (e.g. in configurators or database systems), and dedicated algorithms are essential to handle them.

Among existing algorithms for non-binary CSPs are GAC3 [11] and GAC4 [13]. These algorithms are extensions of binary algorithms, and are much more expensive than their counterparts. Indeed, the best worst-case time complexity with such generic algorithms is exponential in the greatest constraint arity. Other algorithms have then been designed to achieve arc consistency for specific problems, containing only table constraints for instance. This is the case for Simple Tabular Reduction (STR) [15, 9], which can achieve arc consistency in linear time in the size of the table.

Arc consistency algorithms are well suited for growing constraint problems, and thus can easily handle the addition of constraints in a problem. However, this is not true for constraint retraction. The problem is still arc consistent, but some solutions may be lost as result of the retraction, and thus the system has to find out which values should be restored, in order to achieve *maximal* arc consistency. Several techniques have been proposed to maintain arc consistency in binary Dynamic CSPs. The most recent ones are DnAC4 [4], DnAC6 [8], and AC|DC and its extensions [3, 14, 2]. Those algorithms are based on static arc consistency algorithms, respectively AC4, AC6 and AC3. For non-binary DCSPs, however, DnGAC4 [5] is currently the only algorithm that has been extended to handle them. This algorithm is an extension of DnAC4, and has thus the same time complexity as GAC4.

In this paper we propose a new algorithm *DnSTR* for solving Dynamic CSPs with non-binary constraints, described as (extensional) table constraints. Table constraints are commonly used in applications where databases are involved, for example product configuration systems. Our algorithm is based on STR, one of the most efficient approaches for maintaining arc consistency on table constraints, instead of the generic GAC4 algorithm.

The paper is organized as follows. Section 2 recalls the definitions of Dynamic CSPs and arc consistency, then presents GAC4 and the most used method for maintaining arc consistency in DCSPs. The algorithm DnSTR is described in detail in Section 3, and is analyzed in Section 4. Finally, an experimental study is presented in Section 5, while Section 6 gives a summary along with some final remarks.

2 DEFINITIONS AND PRELIMINARIES

We will now define what a CSP and a Dynamic CSP are, as well as the concept of arc consistency. We will then introduce the current algorithms for solving non-binary CSPs, and the method to maintain arc consistency in DCSPs.

2.1 Dynamic Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a triple $\mathcal{P} = \langle X, D, C \rangle$ where: $X = \langle x_1, x_2, \dots, x_n \rangle$ is an n -tuple of variables, $D =$

¹ DTU Informatics, Technical University of Denmark, email: {mq, probst}@imm.dtu.dk

² Microsoft Development Center Copenhagen, email: {t-mqueva, lricci}@microsoft.com

$\langle D(x_1), D(x_2), \dots, D(x_n) \rangle$ is an n -tuple of *domains*, representing, for each variable x_i , the set of possible values it can take, and $C = \langle c_1, c_2, \dots, c_t \rangle$ is a t -tuple of constraints restricting the values that the variables can simultaneously take. A constraint c involves a subset $scp(c)$ of variables from X and can be defined by the set of all tuples of values allowed (extensional table constraint) or by a relation between variables (in intension). Solving a CSP consists in assigning a value to each variable such that all constraints are satisfied. The example of a CSP can be seen in Figure 1.

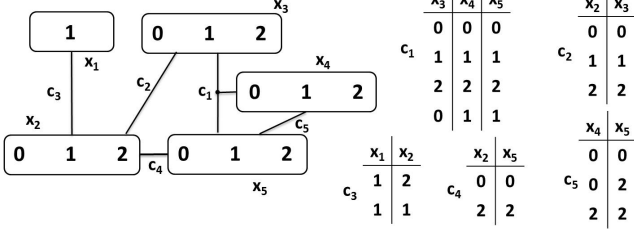


Figure 1. Example of CSP. Each node represents a variable whose domain is explicitly indicated. The constraints are defined in extension to specify the set of allowed tuples.

A Dynamic Constraint Satisfaction Problem (DCSP) is a sequence of CSPs $\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_m$ where each problem results from an addition or removal of a constraint c in its predecessor. More precisely, if $\mathcal{P}_i = \langle X, D, C_i \rangle$, then $\mathcal{P}_{i+1} = \langle X, D, C_{i+1} \rangle$ where $C_{i+1} = C_i \pm c$.

2.2 Arc consistency

Constraint satisfaction problems on finite domains are typically solved using a form of search. However, since finding solutions in a CSP is an NP-complete task, filtering techniques can be used to simplify the problem during search. The most used algorithms are the ones achieving arc consistency.

Definition 1 For a constraint c on $(x_1, \dots, x_i, \dots, x_p)$, a tuple $\tau = (d_1, \dots, d_i, \dots, d_p)$ is a **support** of c if it is an allowed tuple ($\tau \in c$) and $\forall x_i \in scp(c), \tau(x_i) = d_i \in D(x_i)$. A pair (x_i, d_{x_i}) is said to be **arc consistent** iff, for each $c \in C$ constraining x_i , there exists a support $\tau \in c$ such as $\tau(x_i) = d_{x_i}$. A problem $\mathcal{P}_k$ is **arc consistent** iff $\forall x_i \in X, \forall d_{x_i} \in D(x_i), (x_i, d_{x_i})$ are arc consistent. $\mathcal{P}_k$ is **maximally arc consistent** if enlarging the domain of any of its variables makes it not arc consistent.

2.3 Arc consistency in DCSPs

Maintaining arc consistency in Dynamic CSPs is equivalent to making the problem $\mathcal{P}_i$ maximally arc consistent, provided that $\mathcal{P}_0, \dots, \mathcal{P}_{i-1}$ are maximally arc consistent. Handling constraint *addition* is fairly straightforward: the new problem can indeed be made maximally arc consistent just by applying GAC algorithms again. However, the task of *removing* a constraint is a bit different. In fact, when removing a constraint, the problem stays arc consistent, but not necessary maximally arc consistent. This is due to the fact that some solutions can be lost, and formerly inconsistent values may need to be restored to re-establish maximal arc consistency.

Existing algorithms for arc consistency in DCSPs are working in three phases during constraint removal: first the values deleted by the removed constraints are restored (initialization phase); then those

restorations are propagated to other variables connected through existing constraints (propagation phase); finally, the values are filtered again (filtering phase). The last phase ensures that all wrongly restored values are removed. The latest and most efficient algorithms for binary constraints are AC[DC-2i and AC3.1][DC-2i [2].

2.4 CSPs with non-binary constraints

Enforcing arc consistency on CSPs with non-binary constraints is NP-complete [7], and the best worst-time complexity obtained by generic algorithms such as GAC4 is $O(erd^r)$, with e the number of constraints, r the greatest constraint arity and d the greatest domain size. Many other algorithms have been designed to enforce arc consistency in a more efficient way for specific problems, for example non-binary CSPs with table constraints. Tables can actually be considered as the most explicit way to represent constraints, as in theory any constraints in intension can be transformed into a table constraint. However, this may in practice lead to time and space explosion when solving the problem.

Among the work on table constraints, one particular method of interest is Simple Tabular Reduction (STR) [15, 9]. This algorithm differs from previous methods as it dynamically maintains the tables in order only to go through the supporting tuples when enforcing arc consistency. More precisely, when the domain of a variable is reduced, each table constraining that variable is updated, by removing all invalid tuples. The variables' values that are not arc consistent are then removed from their domain. STR uses arrays of indexes to avoid moving tuples in memory, and is enforcing arc consistency in $O(er(d + t'))$, where t' is the maximum size of the current tables. Worst-case space complexity of STR is $O(e(n + rt))$, with t the greatest original table size. STR has been shown to be one of the most efficient approaches when dealing with table constraints. A more detailed description of STR can be found in [9].

3 THE ALGORITHM DnSTR

In this section, we will describe our new DnSTR algorithm. This algorithm can be used to solve DCSPs with table constraints. It relies on a timeline of events, which is composed by timestamps. Each timestamp t is defined by two integers (dp, lt) , which represent the *depth* of the timestamp (the depth is increased every time a constraint is added) and the *local time* at a given depth. The timestamps can also be compared: $\forall t_1, t_2 : t_1 < t_2 \Leftrightarrow (t_1.dp < t_2.dp \vee (t_1.dp = t_2.dp \wedge t_1.lt < t_2.lt))$.

Like the other algorithms targeting DCSPs, DnSTR is working in three phases (as described in Section 2.3). However, the filtering phase in DnSTR is performed incrementally during constraint retraction, in order to keep the timeline consistent and use it to restore only specific tuples. Indeed, during constraint retraction, wrongly restored tuples need to be put back at the depth they would have been deleted at if the removed constraint had never been added to the problem. Thanks to this mechanism, when a value is restored at a certain depth, only tuples at this same depth have to be restored during propagation (except if the constraint associated to those tuples was added later, in which case the depth considered is this constraint addition's depth).

The algorithm works with the following data structures:

- A global depth counter $gdepth$, as well as an array $lastLTime$ that for each depth dp associates the last local time lt in the current timeline.
- Two justifications arrays $justif_t$ and $justif_c$ that, for each pair (x, d_x) , keep track of when and by which constraint the value d_x

has been removed from x 's domain. Each constraint is also associated with the time of addition, recorded in the array $time_c$.

- As we already pointed out, the algorithm DnSTR is based on Simple Tabular Reduction. It thus shares similar data structures for each table constraint, including an initial set of tuples $c.table$, the index of the first tuple in the current table $c.first$ (i.e. the table of valid tuples), an array $c.next$ that links lists of tuples (by their indexes), and two arrays $c.removedHead$ and $c.removedTail$ that give the index of the first and last invalid tuples at each depth. Note that $c.next[i] = -1$ if i is the position of the last tuple, either in the current table if i is a valid tuple or in the removed list if i has been removed. For each constraint c , we also store the index of the last restored tuple in $c.restoredTail$, and an array $c.removed_t$ that records the removal time (or timestamp) of each tuple.

3.1 Constraint addition

Adding a constraint to the DCSP is done by filtering the system through the function ADDCONSTRAINT described in Algorithm 1.

Algorithm 1: ADDCONSTRAINT(c : Constraint): Bool

```

1 add  $c$  to  $C$ ;
2  $gdepth = gdepth + 1$ ;  $lastLTime[gdepth] = 0$ ;
3  $time_c[c] = (gdepth, 0)$ ;
4  $revise[gdepth] = \{c\}$ ;
5 return FILTER( $revise, D, (gdepth, 0)$ );

```

Algorithm 2: FILTER($revise, values$: Arrays, (dp_i, lt_i) : Time): Bool

```

1 while  $revise[dp_i] \neq \emptyset$  do
2   select and remove  $c$  from  $revise[dp_i]$ ;
3    $changed = \emptyset$ ;
4   if not GACSTRDYN( $c, changed, (dp_i, lt_i), values$ ) then
5     return false;
6   foreach  $(x, jt_x) \in changed$  do
7      $c_{revise} = \{c_r \in C \text{ s.t.}$ 
8        $c_r \neq c, x \in scp(c_r), time_c[c_r] \leq (dp_i, lt_i)\}$ ;
9     if  $jt_x.dp > dp_i$  then
10      add  $c_{revise}$  to  $revise[jt_x.dp]$ ;
11    else
12      add  $c_{revise}$  to  $revise[dp_i]$ ;
13 return true;

```

The $revise$ structure contains for each depth the constraints to evaluate at filtering. FILTER iterates over the constraints to revise at the given depth (the latest for constraint addition) and applies a modified version of STR (Algorithm 3), where only the elements in the list $values$ are taken into account ($values$ represents the full domain D for constraint addition, but only the restored values for constraint retraction). GACSTRDYN also differs from [9] as it handles the restored tuples (for constraint retraction) and the update of the justifications. Those are computed at lines 30-31, except if the pair (x, d_x) had been removed by the same constraint at a later stage (line 29) — in which case another support for (x, d_x) is still valid at this time but will be removed later in time. ISVALID also differs as it checks the value's justification time, while REMOVETUPLE is identical to the one described in [9], and is thus not shown. FILTER then updates $revise$ with potentially affected constraints. The timestamp parameter (dp_i, lt_i) is used to specify when the filtering occurs (line 7), so that the constraints added *after* that time are not filtered yet (but will be later, at the incremental filtering used during constraint retraction). If x was changed at a later depth (lines 8-9), the constraint to revise is inserted at that specific depth $jt_x.dp$ instead of the current one.

Algorithm 3: GACSTRDYN(c : Constraint, $changed$: Set, (dp_i, lt_i) : Time, $values$: Array): Bool

```

1  $S_{sup} = \emptyset$ ;
2 foreach  $x \in scp(c)$  do
3    $gacValues[x] = \emptyset$ ;
4   if  $values[x] \neq NIL$  then
5     add  $x$  to  $S_{sup}$ ;
6  $prev = -1$ ;
7  $curr = c.first$ ;
8 while  $curr \neq -1$  do
9    $\tau = c.table[curr]$ ;
10   $nextT = c.next[curr]$ ;  $rTail = c.restoredTail$ ;
11  if ISVALID( $c, \tau, (dp_i, lt_i)$ ) then
12    foreach  $x \in S_{sup}$  do
13      if  $(\tau[x] \notin gacValues[x] \text{ and } \tau[x] \in values[x])$  then
14        add  $\tau[x]$  to  $gacValues[x]$ ;
15        if  $|gacValues[x]| = |values[x]|$  then
16          remove  $x$  from  $S_{sup}$ ;
17     $prev = curr$ ;
18  else
19     $c.removed_t[curr] = (dp_i, lt_i)$ ;
20    if  $c.restoredTail = curr$  then  $c.restoredTail = prev$ ;
21    REMOVETUPLE( $c, prev, curr, dp_i$ );
22  if  $curr = rTail$  then break;
23   $curr = nextT$ ;
24 foreach  $x \in S_{sup}$  do
25   foreach  $d_x \in values[x]$  do
26     if  $d_x \notin gacValues[x]$  then
27       remove  $d_x$  from  $values[x]$ ;
28       if  $|D[x]| \neq |values[x]|$  then remove  $d_x$  from  $D[x]$ ;
29       if not ( $justif_c[x, d_x] = c$  and  $justif_t[x, d_x] > (dp_i, lt_i)$ ) then
30          $justif_c[x, d_x] = c$ ;
31          $justif_t[x, d_x] = (dp_i, lt_i)$ ;
32         add  $(x, justif_t[x, d_x])$  to  $changed$ ;
33   if  $|D[x]| = 0$  then return false;
34  $lastLTime[dp_i] = lt_i + 1$ ;
35 return true;

```

Algorithm 4: ISVALID(c : Constraint, τ : Tuple, (dp_i, lt_i) : Time): Bool

```

1 foreach  $x \in scp(c)$  do
2   if  $(\tau[x] \notin D[x])$  and  $(justif_t[x, \tau[x]] \leq (dp_i, lt_i))$  then
3     return false;
4 return true;

```

Algorithm 5: REMOVECONSTRAINT(c : Constraint): Bool

```

// Initialization phase
1  $restoredValues = \emptyset$ ;
2 foreach  $x \in scp(c)$  do
3   foreach  $d_x \in D_0[x] \setminus D[x]$  do
4     if  $justif_c[x, d_x] = c$  then
5       add  $d_x$  to  $D[x]$ ;
6       add  $d_x$  to  $restoredValues[x]$ ;
7 Remove  $c$  from  $C$ ;
// Propagation phase
8  $revise = \emptyset$ ;
9  $tcr = PROPAGATE(restoredValues, revise, time_c[c].depth)$ ;
// Filtering phase
10 for  $dp_i \leftarrow time_c[c].dp$  to  $gdepth$  do
11   FILTER( $revise, restoredValues, (dp_i, lastLTime[dp_i])$ );
// Cleaning phase
12 foreach  $c_j \in tcr$  do
13    $c_j.restoredTail = -1$ ;
14 foreach  $x \in restoredValues$  do
15   foreach  $d_x \in restoredValues[x]$  do
16      $justif_t[x, d_x] = NIL$ ;
17      $justif_c[x, d_x] = NIL$ ;
18    $restoredValues[x] = NIL$ ;
19 return true;

```

Algorithm 6: PROPAGATE(*restoredValues*: Set, *revise*: Array, *dp_{rem}*: Int): Set

```

1 restore =  $\emptyset$ ;
2 foreach  $x \in \text{restoredValues}$  do
3   UPDATELISTS( $x$ , restoredValues[ $x$ ], revise, restore, dprem)
4 while restore  $\neq \emptyset$  do
5   select and remove ( $c$ , minRestoredTimes) from restore;
6   newRestored =  $\emptyset$ ; add  $c$  to tcr;
7   foreach ( $dp, lt$ )  $\in$  minRestoredTimes do
8     curr = c.removedHead[ $dp$ ]; prev = -1;
9     rTail = c.removedTail[ $dp$ ];
10    if c.restoredTail = -1 then c.restoredTail = rTail;
11    RESTORETUPLES( $c$ ,  $dp$ );
12    while curr  $\neq$  -1 do
13      nextT = c.next[curr];
14      if c.removedt[curr] < ( $dp, lt$ ) then
15        if c.restoredTail = curr then
16          c.restoredTail = prev;
17        REMOVETUPLE( $c$ , prev, curr,  $dp$ );
18      else
19         $\tau$  = c.table[curr]; jTimes =  $\emptyset$ ;
20        foreach  $x \in \text{scp}(c)$  do
21          if  $\tau[x] \notin D[x]$  then
22            if justifc[ $x$ ,  $\tau[x]$ ]  $\neq c$  then
23              add justifc[ $x$ ,  $\tau[x]$ ] to jTimes;
24            else
25              add  $\tau[x]$  to D[ $x$ ];
26              add  $\tau[x]$  to newRestored[ $x$ ];
27          if jTimes  $\neq \emptyset$  then
28            add  $c$  to revise[min(jTimes)];
29          c.removedt[curr] = NIL; prev = curr;
30          if curr = rTail then break;
31          curr = nextT;
32 foreach ( $x \in \text{newRestored}$ ) do
33   UPDATELISTS( $x$ , newRestored[ $x$ ], dprem, revise, restore);
34   add newRestored[ $x$ ] to restoredValues[ $x$ ];
35 return tcr;

```

Algorithm 7: UPDATELISTS(x : Var, *restored_x*: Set, *dp_{rem}*: Int, *revise*, *restore*: Arrays)

```

1 foreach  $c_x \in C$  constraining  $x$  do
2   if ( $\exists d_x \in \text{restored}_x$  s.t. justift[ $x$ ,  $d_x$ ].dp  $\leq$  timec[ $c_x$ ].dp) then
3     add  $c_x$  to revise[max(timec[ $c_x$ ].dp, dprem)];
4   jtimes = {max(justift[ $x$ ,  $d_x$ ], timec[ $c_x$ ]) s.t.  $d_x \in \text{restored}_x$  and justifc[ $x$ ,  $d_x$ ]  $\neq c_x$ };
5   add { $t \in jtimes$  s.t.  $t = \min(t_1 \in jtimes, t_1.dp = t.dp)$ } to restore[ $c_x$ ];

```

3.2 Constraint removal

Constraint retraction is performed by the function REMOVECONSTRAINT (Algorithm 5).

During the initialization stage, all pairs (x, d_x) that were removed by the retracted constraints are restored. Notice that all restored values are placed in a list *restoredValues* so that only those values are considered in the filtering stage.

During the propagation stage in function PROPAGATE, tuples from constraints connected to restored variables are restored. The list *minRestoredTimes* contains the minimum times (dp, lt) for each depth where tuples need to be restored. After tuples have been restored for a specific depth (line 11), the ones that have been deleted before the time (dp, lt) considered are immediately removed (lines 14-17). For each tuple, only values that had been deleted by the current constraint are restored. The function RESTORETUPLES(c, t) is identical to the one described in [9]. In case another constraint has made a tuple invalid, the table is added to *revise* at the time when a value first invalidated the tuple (line 27-28). Thanks to the strictly maintained timeline, only the tuples that have been removed after a pair (x, d_x)

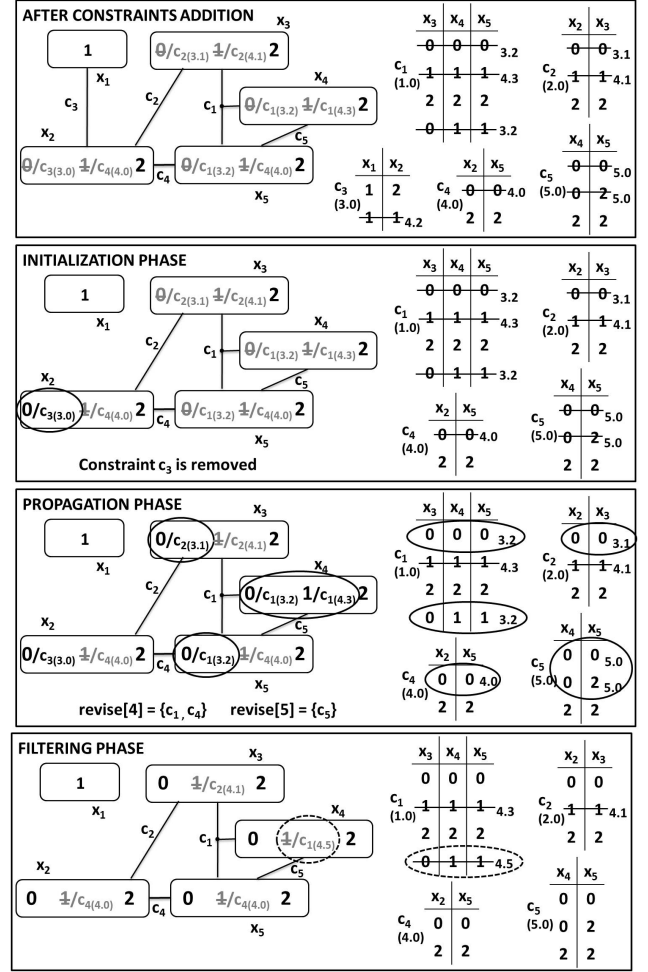


Figure 2. Removing constraint c_3 in the CSP example from Figure 1. Associated timestamps are written as “3.1” for ($dp = 3, lt = 1$) next to each added constraint and removed tuple. Justifications (constraint and time) are also shown next to each deleted value. Plain circles show the values and tuples restored since the previous phase, while dashed circles represent values and tuples re-filtered.

and in the same depth need to be restored. This is done through the update of *restore* in function UPDATELISTS: depending whether the value d_x was deleted before the constraint c_x was added, the justification time or the constraint addition time is retrieved (line 4) and the minimum for each depth is added to *restore* (line 5). The array *revise* is also updated at the beginning of the propagation and after each restoration to include constraints that were added after the restored values were deleted (line 2-3 in UPDATELISTS). Note that the PROPAGATE algorithm returns the set *tcr* of table constraints that have been modified during restoration.

Finally, filtration is performed on the constraints in *revise* (line 12-13 in REMOVECONSTRAINT). This is done incrementally so that constraints are evaluated in the same order as they were added, keeping the new problem’s timeline consistent with what it would be if the retracted constraint had never been added. Although it may introduce redundancy in the constraint evaluation process, this type of filtering is necessary to guarantee that all invalid tuples restored in the propagation stage are put back at the correct depth.

3.3 Example

An example of constraint removal can be seen in Figure 2, after adding all constraints in the CSP from Figure 1. It is interesting to note that although the last tuple of table c_1 is restored, it is potentially invalid, as $(x_5, 1)$ has been removed by c_4 : $(x_5, 1)$ is thus not restored, and the table c_1 is put in *revise* for the filtering phase. The two tables c_4 and c_5 are also added to *revise*, as the restored $(x_2, 0)$ and $(x_4, 1)$ had been deleted before the addition of those tables: they might be deleted again in the filtering stage.

4 ANALYSIS

In this section, we will discuss the correctness of DnSTR, and describe the time and space complexity of the algorithm.

4.1 Correctness

During constraint addition, the filtering depth is the global depth $gdepth$, and the whole domain values are checked when using the STR algorithm. As the second check in ISVALID is always true, the STR algorithm used is equivalent to the original algorithm. All constraints to revise are put back in *revise*[$gdepth$] in FILTER as the check of line 8 is always false with $dp_i = gdepth$, so the correctness directly comes from the correctness of STR.

For constraint removal, let us consider a problem $\mathcal{P}_n$, resulting of the addition of $(c_0, \dots, c_n)$, and where we are removing constraint c_i . Before the removal of c_i , problem $\mathcal{P}_n$ is arc consistent, so for all $\mathcal{P}_j$ with $j < n$, all justifications arrays are coherent with the actual timeline, all tuples have been removed at the correct depth and always at a local time greater than any justification of deleted value that might have caused its removal.

First, consider that c_i was added at $t_i = (d_i, 0)$, and $t_i + t_0$ is the first time that a value from a variable $x \notin scp(c_i)$ has been removed.

All the removal of values before $t_i + t_0$ are caused by c_i , and so those values are directly restored during the initialization phase, thanks to having c_i as justification constraint. For each constraint, the tuples impacted by a value removed at t_r are tuples deleted at a later time $t > t_r$. Moreover, the only impacted tuples are the ones deleted in the same depth, as $\mathcal{P}_i$ was made consistent before any new constraint addition (and change of depth), *except* if they are from a constraint c_j added *after* c_i , as they may have been removed because of values deleted in $[t_i, t_i + t_0[$ at $d_j > d_i$. All those tuples are considered for restoration in UPDATELISTS (line 4-5) and restored in PROPAGATE (line 11). All values supported by those restored tuples of any table c and which had been removed by c are restored in PROPAGATE (line 25-26). The propagation of these restorations (via the *restore* list) ensures that a superset of all impacted values is restored.

We need now to make sure that all restored tuples that are removed during the filtering phase are put back at the correct depth and time. The incremental filtering is performed chronologically, so constraints just need to be added in revise at the correct depth. The local time is always increased during filtering using STR, so any removed tuple will always be placed after the first value responsible for its deletion. There are two cases when table constraints must be revised. The first one occurs when invalid tuples are restored. The tuples must then be placed at the earliest time when one of its value were deleted from its domain. This is taken care of in PROPAGATE (line 28). In the second case, a value restored during propagation may actually have to be deleted by another constraint that had been added later (line 2-3 in UPDATELISTS).

Finally, justifications need to be maintained throughout the filtering process. When a restored value (x, d_x) is filtered by a constraint c_m at t_1 , its justification constraint is obviously c_m . However, if d_x had been removed (prior to restoration) because of c_m but at a later time $t_2 > t_1$, it means that there is still another supporting tuple for (x, d_x) valid at t_1 (that will be removed only at t_2), and thus the justification time should stay equal to t_2 (GACSTRDYN line 29).

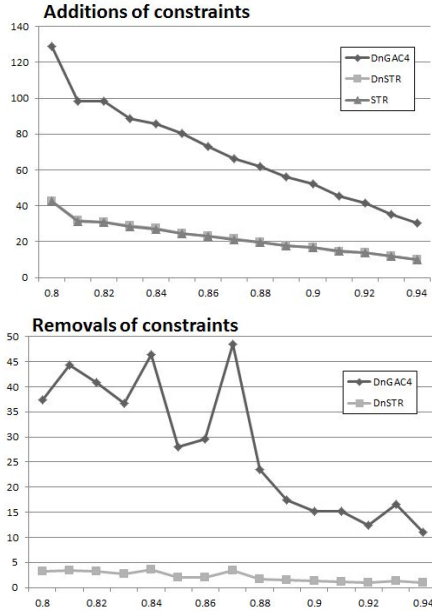


Figure 3. Runtime (in ms) as a function of p_2 for $r = 8$

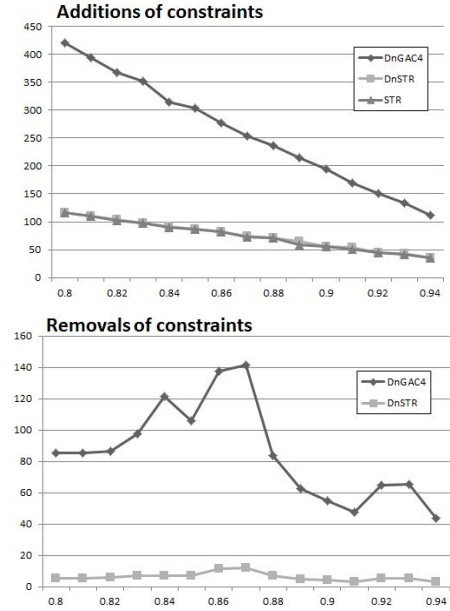


Figure 4. Runtime (in ms) as a function of p_2 for $r = 9$

Table 1. Memory consumption for RCSP $\langle 8, 20, 3, 100, p_2 \rangle$ (in MB)

p_2	0.82	0.84	0.86	0.88	0.9	0.92	0.94
DnSTR4	8.3	8.2	7.6	7.4	7.3	6.7	5.1
STR	3.2	3	2.7	2.5	2.5	1.7	1.1
DnGAC4	12.5	12.3	10.4	8	7.1	6.9	5.7

4.2 Time and space complexity

The worst-case time complexity of the STR is $O(er(d + t'))$ (the number of constraints e is added because the whole constraint system is considered). In the case of DnSTR, the worst-case time complexity of the initialization phase and the propagation phase is $O(er(d + t'))$, where t' represents the size of the removed tuples list. However, the incremental filtering in the filtering phase may introduce a factor e , as the filtering could occur in the worst case as many times as there are constraints (or depths). The worst-case time complexity for DnSTR is thus $O(e^2r(d + t'))$.

The space complexity of STR is $O(e(n + rt))$. For DnSTR, justifications arrays are in $O(nd)$. For each constraint, as for STR, *table* is in $O(rt)$ and *next* in $O(t)$. The arrays *removedHead* and *removedTail* are each in $O(e)$, as they can be filled for each depth, and *removed_t* is in $O(t)$, which makes a worst-case space complexity of $O(e + rt)$ for each constraint, and $O(nd + e(e + rt))$ for DnSTR.

5 EXPERIMENTAL RESULTS

In order to evaluate the practical behaviour of the proposed algorithm, we have implemented DnSTR in C# and compared it to DnGAC4 and the classic STR (where backtracking is used to remove constraints). The experiments were conducted using Random CSPs with model B [16]. Each instance is characterized by a tuple $\langle r, n, d, m, p_2 \rangle$ where r is the arity of each constraint, n the number of variables, d the uniform domain size, m the number of constraints and p_2 the tightness of the each constraint. The tightness defines the number $t = (1 - p_2)d^k$ of allowed tuples in each constraint. For each arity $r \in \{8, 9\}$, we did the experiments with 10 instances of RCSP $\langle r, 20, 3, 100, p_2 \rangle$, where $p_2 \in [0.8 - 0.94]$. The constraints were first added one by one (inconsistent systems were discarded). In order to see some real change in the variables' domain with such a low density (a more suitable density for a problem with arity 8 requires much more constraints), 10% random unary constraints were then added. Then 10% of all the constraints were removed. This protocol was applied 10 times per instance. The mean values of the runtime for adding and removing constraints are shown in Figures 3 and 4.

The experiments show that when adding a constraint, STR and DynSTR are almost identical and faster than DnGAC4. This is not surprising, as the STR algorithm, also used in DnSTR, has been shown more efficient than GAC4 in [9]. For constraint removal, DnSTR outperforms both STR and DnGAC4. The results for the STR algorithms are not shown, as it performs more than 20 times worse than DnGAC4. The poor performance of STR can be explained by the sometimes important amount of work it has to perform again after backtracking all the way to the removed constraint. DnGAC4 performs better than the classic STR, as it only checks restored values and updates justifications arrays. However, DnSTR beats DnGAC4 again, thanks to the interaction with the dynamic tables of valid tuples and the maintained timeline. It is worth pointing that the overhead of the incremental filtering necessary for preserving the timeline does not seem to have such an important impact on performance.

We also measured the memory consumption of DnSTR, STR and DnGAC4 with RCSP $\langle 8, 20, 3, 100, p_2 \rangle$. To use as much memory as possible, we added all the constraints and assigned values to random variables until the problem stopped being consistent. Table 1 shows the mean memory consumption of the algorithms for different values of the tightness p_2 , just before the system became inconsistent. As we can see, the memory consumption between DnGAC4 and DnSTR is close, also DnSTR performs a bit better, while STR uses much less memory, as it has much less data structures to maintain.

6 CONCLUSION

In this paper, we propose a new algorithm (DnSTR) to solve Dynamic CSPs with non-binary constraints, described as (extensional) table constraints. Table constraints are commonly used in applications where databases are involved, for example product configuration systems. Our algorithm uses a timeline to record when and why values from domains have been removed. This timeline allows to decide the consequences of retracting constraints. While similar algorithms are based on the generic GAC4 algorithm, our algorithm is based on Simple Tabular Reduction for table constraints. We have presented benchmarks for our algorithm that show that DnSTR as a result performs better than the only other existing algorithm for such problems, DnGAC4.

Before concluding, it is worth noting that DnSTR uses the same justifications data structures as the best algorithms for DCSPs with binary constraints (AC|DC-2i and AC3.1|DC-2i [2]). This allows to adapt our approach easily to deal with binary constraints defined in intension.

REFERENCES

- [1] F. Bacchus, X. Chen, P. Beek, and T. Walsh, 'Binary vs. non-binary constraints', *Artificial Intelligence*, **140**, 1–37, (2002).
- [2] R. Bartak and P. Surynek, 'An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems', *Proc. FLAIRS'05*, 161–166, (2005).
- [3] P. Berlandier and B. Neveu, 'Arc-consistency for dynamic constraint satisfaction problems: a rms free approach', *Proc. ECAI'94, Workshop on Constraint Satisfaction Issues Raised by Practical Applications*, (1994).
- [4] C. Bessière, 'Arc-consistency in dynamic constraint satisfaction problems', *Proc. AAAI'91*, 221–226, (1991).
- [5] C. Bessière, 'Arc-consistency for non-binary csp's', *Proc. ECAI'92*, 23–27, (1992).
- [6] C. Bessière, 'Arc-consistency and arc-consistency again', *Artificial Intelligence*, **65**, 179–190, (1994).
- [7] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh, 'The complexity of reasoning with global constraints', *Constraints*, **12**(2), 239–259, (2007).
- [8] R. Debruyne, 'Arc-consistency in dynamic csp's is no more prohibitive', *Proc. ICTAI'96*, 239–267, (1996).
- [9] C. Lecoutre, 'Optimization of simple tabular reduction', *Proc. CP'08*, 128–143, (2008).
- [10] A. K. Mackworth, 'Consistency in networks of relations', *Artificial Intelligence*, **8**, 99–118, (1977).
- [11] A. K. Mackworth, 'On reading sketch maps', *Proc. IJCAI'77*, 598–606, (1977).
- [12] R. Mohr and TC Henderson, 'Arc and path consistency revisited', *Artificial Intelligence*, **28**, 225–233, (1986).
- [13] R. Mohr and G. Masini, 'Good old discrete relaxation', *Proceedings ECAI'88*, 651–656, (1988).
- [14] M. Mouhoub, 'Arc-consistency for dynamic csp's', *Proc. KES'03*, **2773**, 393–400, (2003).
- [15] J. R. Ullmann, 'Partition search for non-binary constraint satisfaction', *Information Science*, **177**, 3639–3678, (2007).
- [16] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre, 'Random constraint satisfaction: Easy generation of hard (satisfiable) instances', *Artificial Intelligence*, **171**, 514–534, (2007).