Dynamic Arc Consistency for CSPs

Malek Mouhoub
University of Regina
Wascana Parkway, Regina, SK, Canada, S4S 0A2
Phone: +1 (306) 585 4700
Fax: +1 (306) 585 4745
Email: mouhoubm@cs.uregina.ca

March 4, 2009

Abstract

Constraint Satisfaction Problems (CSPs) are fundamental in many real world applications such as interpreting a visual image, laying out a silicon chip, frequency assignment, scheduling, planning and molecular biology. A main challenge when designing a CSP-based system is the ability to deal with constraints in a dynamic and evolutive environment. We talk then about on line CSP-based systems capable of reacting, in an efficient way, to any new external information during the constraint resolution process. During the conceptual phase of design, for example, the designers should be able to add/remove constraints at any time after specifying an initial statement describing the desired properties of a required artifact. We propose in this paper a new algorithm capable of dealing with dynamic constraints at the arc consistency level of the resolution process. More precisely, we present a new dynamic arc consistency algorithm that has a better compromise, in practice, between time and space than those algorithms proposed in the literature, in addition to the simplicity of its implementation. Experimental tests on randomly generated CSPs as well as on CSPs involving temporal constraints (that we call TCSPs), have been conducted. The results of the experimentation demonstrate the efficiency, in time and space costs, of our algorithm to deal with large size CSPs in a dynamic environment.

Keywords: Constraint Satisfaction, Temporal Reasoning, Dynamic Arc Consistency.

1 Introduction

Constraint Satisfaction problems (CSPs) [8] are a fundamental concept used in many real world applications such as interpreting a visual image, laying out a silicon chip, frequency assignment, scheduling, planning and molecular biology. This motivates the scientific community from artificial intelligence, operations research and discrete mathematics to develop different techniques to tackle problems of this kind[12]. These techniques become more popular after they were incorporated into constraint programming languages [11, 2, 8]. A main challenge when designing a CSP-based system is the ability to deal with constraints in a dynamic and evolutive environment. We talk then about on line CSP-based systems capable of reacting, in an efficient way, to any new external information during the constraint resolution process. One example, in the case of scheduling problems, is when a solution, corresponding to an ordering of tasks to be processed, has to be reconsidered after a given machine becomes unavailable. We have then to look for another solution (ordering of tasks) satisfying the old constraints and taking into account the new information. Another example, in the area of engineering conceptual design, is when the designers add/remove constraints after specifying an initial statement describing the desired properties of a required artifact during the conceptual phase of design.

A CSP involves a list of variables defined on finite domains of values and a list of relations restricting the values that the variables can take. If the relations are binary we talk about binary CSPs. Solving a CSP consists of finding an assignment of values to each variable such that all relations (or constraints) are satisfied. A CSP is known to be an NP-Hard problem. Indeed, looking for a possible solution to a CSP requires a backtrack search algorithm of exponential complexity in time[1]. To overcome this difficulty in practice, local consistency techniques are used in a pre-processing phase to reduce the size of the search space before the backtrack search procedure. Also, in the case of inconsistent CSPs, the inconsistency can be detected at the pre-processing level. The backtrack phase will then be saved. A k-consistency algorithm removes all inconsistencies involving all subsets of k variables belonging to N. The k-consistency algorithm is polynomial in time, $O(N^k)$, where N is the number of variables. A k-consistency algorithm does not solve the constraint satisfaction problem, but simplifies it. Due to the incompleteness of constraint propagation, in the general case, search is necessary to solve a CSP problem, even to check if a single solution exists. When $k = 2$ we talk about arc consistency. An arc consistency algorithm transforms the network of constraints into an equivalent and simpler one by removing, from the domain of each variable, some values that cannot belong to any global solution.

We propose in this paper a new technique capable of dealing with dynamic constraints at the arc consistency level. More precisely, we present a new dynamic arc consistency algorithm that has a better compromise, in practice, between time and space than those algorithms proposed in the literature [3, 7, 18], in addition to the simplicity of its implementation. We call this algorithm AC3.1|DC. In order to evaluate, in practice, the performance in time and memory space costs of the algorithm we

[1]Note that some CSP problems can be solved in polynomial time. For example, if the constraint graph corresponding to the CSP has no loops, then the CSP can be solved in $O(nd^2)$ where n is the number of variables of the problem and d the domain size of the different variables [8].

propose, experimental tests on randomly generated CSPs and temporal CSPs (CSPs involving temporal constraints) have been performed. The results of the experimentation demonstrate the efficiency in time and memory costs of our method to deal with large size problems. Note that the Temporal CSP (that we call TCSP) is a model we proposed in [16] in order to represent a wide variety of real world applications including scheduling and planning. Often these systems are handled in an evolutive environment and are thus required to be solved in a dynamic manner.

The rest of the paper is organized as follows. In the next section we will present an overview of the arc consistency and dynamic arc consistency algorithms proposed in the literature. Our dynamic arc consistency algorithm is then presented in section 3. Theoretical comparison of our algorithm and those proposed in the literature is covered in this section. The experimental part of our work is presented in section 5. Finally, concluding remarks and possible perspectives are listed in section 5.

2 Dynamic Arc-Consistency Algorithms

2.1 Arc Consistency Algorithms

The key AC algorithm was developed in [13] called AC-3 over twenty years ago and remains one of the easiest to implement and understand today. Figure 1 illustrates the code of the algorithm AC-3. As shown in line 2 of the algorithm, AC-3 starts with a list of all variable pairs (i, j) and enforces the arc consistency for each of these pairs through the function *REVISE* as follows. For each value a from i'domain, *REVISE* looks for a value b in j's domain such that the constraint between i and j holds. If no such value b is found, value a is removed from i's domain (as it has no value in j's domain supporting it). This change will be propagated to all the variables sharing a constraint with variable i. For that, we reconsider all variable pairs (k,i) where k is a variable sharing a constraint with i. This is done in line 6 of the algorithm AC-3.

There have been many attempts to best AC-3 worst case time complexity of $O(ed^3)$ and though in theory these other algorithms (namely AC-4[15], AC-6[4] and AC-7[5]) have better worst case time complexities, they are harder to implement. In fact, the AC-4 algorithm fares worse on average time complexity than the AC-3 algorithm[20]. It was not only until recently when Bessiere et al. [6] proposed an improvement directly derived from the AC-3 algorithm into their algorithm AC2001/3.1. The worst case time complexity of AC-3 is bounded by $O(ed^3)$ [14] where e is the number of constraints and d is the domain size of the variables. In fact this complexity depends mainly on the way the arc consistency is enforced for each arc of the constraint graph. Indeed, if anytime a given arc (i, j) is revised, a support for each value from the domain of i is searched from scratch in the domain of j, then the worst case time complexity of AC-3 is $O(ed^3)$. Instead of a search from scratch, Bessiere et al. [6] proposed a new view that allows the search to resume from the point where it stopped in the previous revision of (i, j). By doing so, the worst case time complexity of AC-3 is achieved in $O(ed^2)$. This new idea is implemented by the function *EXISTb* in Figure 2. This function is used in line 3 of the function *REVISE* in Figure 1. Indeed, in the case of

Function $REVISE(i,j)$
1. $REVISE \leftarrow false$
2. **For** each value $a \in Domain_i$ **Do**
3. **If** there is no $b \in Domain_j$ such that $compatible(a,b)$ **Then**
4. remove a from $Domain_i$
5. $REVISE \leftarrow true$
6. **End-If**
7. **End-For**

Algorithm AC-3
1. Given a constraint network $CN = (E,R)$
   (*E: set of variables, R: set of constraints between variables*)
2. $Q \leftarrow \{(i,j) \mid (i,j) \in R\}$ (list initialized to all relations of CN)
3. **While** $Q \neq Nil$ **Do**
4. $Q \leftarrow Q - \{(i,j)\}$
5. **If** $REVISE(i,j)$ **Then**
6. $Q \leftarrow Q \sqcup \{(k,i) \mid (k,i) \in R \wedge k \neq j\}$
7. **End-If**
8. **End-While**

Figure 1: Pseudo code of the algorithm AC-3.

the standard algorithm the function REVISE works in the same way at each time it is applied on a pair of variables (x,y). More precisely, each time (x,y) is revised, for each value a of x's domain REVISE will start from the beginning looking for a value b in y's domain supporting it. Restarting from the beginning of the variable domain causes the algorithm to run in $O(ed^3)$.

2.2 Dynamic Arc Consistency Algorithms

Before we present the different dynamic arc consistency algorithms, let us define the Dynamic Constraint Satisfaction Problem.

> A dynamic constraint satisfaction problem (DCSP) P is a sequence of static CSPs $P_0, \ldots, P_i, P_{i+1}, \ldots, P_n$ each resulting from a change in the preceding one imposed by the "outside world". This change can either be a constraint restriction (adding a new constraint) or a constraint relaxation (removing a constraint because it is no longer interesting or because the current CSP has no solution). More precisely, P_{i+1} is obtained by performing a restriction (addition of a constraint) or a relaxation (suppression of a constraint) on P_i.

The arc consistency algorithms we have seen in the previous section can easily be adapted to update the variable domains incrementally when adding a new constraint. This simply consists of performing the arc consistency between the variables sharing

Function $EXISTb((i,a),j)$
1. $b \leftarrow ResumePoint((i,a),j)$
($ResumePoint((i,a),j)$ *remembers the first value b such that* $compatible(a,b)$ *is true in the previous revision of* (i,j))
2. **If** $b \in Domain_j$ **Then**
3. return true
4. **Else**
5. **While** $b \leftarrow successor(b, Domain_j^0)$ and $b \neq NIL$
($Domain_j^0$ *denotes the domain of j before arc consistency*)
($successor(b, Domain_j^0)$ *returns the successor of b in* $Domain_j^0$)
6. **If** $b \in Domain_j$ and $compatible(a,b)$ **Then**
7. $ResumePoint((i,a),j) \leftarrow b$
8. return true
9. **End-If**
10. **End-While**
10. return false
11. **End-If**

Figure 2: Pseudo code of AC-3.1 : function for searching b in line 3 of $REVISE(i,j)$.

the new constraint and propagate the change to the rest of the constraint network. However, the way the arc consistency algorithm has to proceed with constraint relaxation is more complex. Indeed, when a constraint is retracted the algorithm should be able to put back those values removed because of the relaxed constraint and propagate this change to the entire graph. Thus, traditional arc consistency algorithms have to be modified so that it will be able to find those values which need to be restored anytime a constraint is relaxed. To illustrate this idea, let us consider the following CSP problem.

Example 1 : the graph coloring problem

Given a graph G (V,E) where :

- V is the set of nodes each defined on a set of colors the node can take,
- and E a set of edges (corresponding the relation $\neq$);

find a color to each node so that no nodes with the same color are adjacent.

This example is illustrated in figure 3. The top left graph corresponds to the graph coloring problem after performing the arc consistency algorithm. Let us assume now that we have the following actions :

- Add the constraint (2 4).
- Add the constraint (3 4).

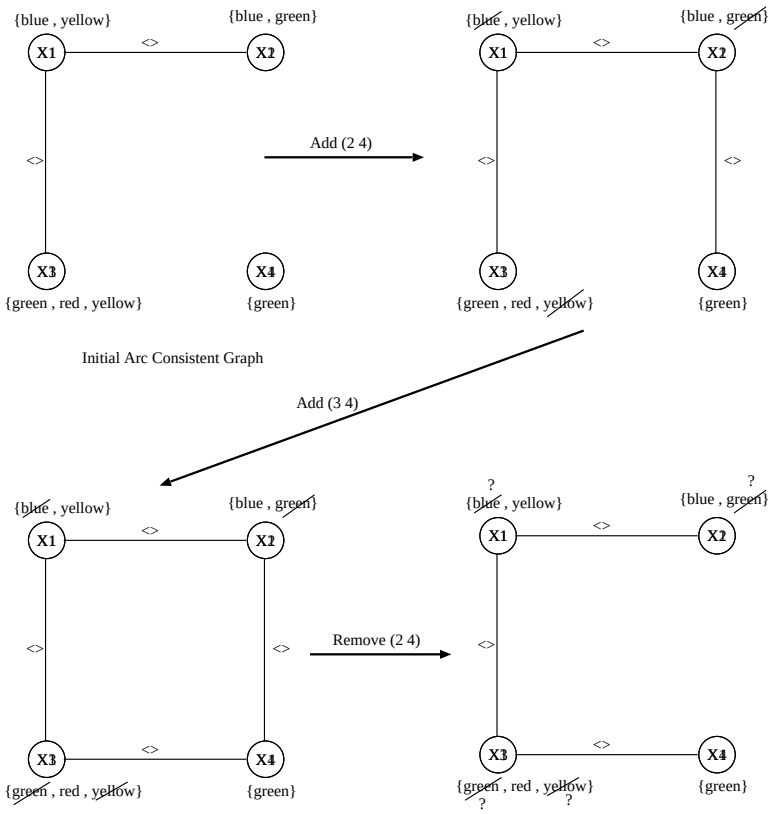

Figure 3: Dynamic Arc Consistency for the Graph Coloring Problem

- Remove the constraint (2 4).

The goal here is to enforce the arc consistency after each action. The first two actions are constraint restrictions and can easily be handled by a traditional arc consistency algorithm. The top right and bottom left graphs of figure 3 show the maintenance of arc consistency respectively after adding the constraints (2 4) and (3 4).

The third action is however more complex. Indeed we need to know which of the suppressed values should be restored.

Bessière has proposed DnAC-4[3] which is an adaptation of AC-4[15] to deal with constraint relaxations. This algorithm stores a justification for each deleted value. These justifications are then used to determine the set of values that have been removed because of the relaxed constraint and so can process relaxations incrementally. DnAC-4 inherits the bad time and space complexity of AC-4. Indeed, comparing to AC-3 for example, AC-4 has a bad average time complexity[20]. The worst-case space complexity of DnAC-4 is $O(ed^2 + nd)$ (e, d and n are respectively the number of constraints, the domain size of the variables and the number of variables). To work out the drawback of AC-4 while keeping an optimal worst case time complexity, Bessière has proposed AC-6[4]. Debruyne has then proposed DnAC-6 adapting the idea of AC-6 for dynamic CSPs by using justifications similar to those of DnAC-4[7]. While keeping an optimal worst case time complexity ($O(ed^2)$), DnAC-6 has a lower space requirements ($O(ed + nd)$) than DnAC-4. To solve the problem of space complexity, Neveu and Berlandier proposed AC|DC[18]. AC|DC is based on AC-3 and does not require data structures for storing justifications. Thus, it has a very good space complexity ($O(e + nd)$) but is less efficient in time than DnAC-4. Indeed, with its $O(ed^3)$ worst case time complexity, it is not the algorithm of choice for large dynamic CSPs.

Our goal here is to develop an algorithm that has a better compromise between running time and memory space than the above three algorithms. More precisely, our ambition is to have an algorithm with the $O(ed^2)$ worst case time complexity of DnAC-6 but without the need of using complex and space expensive data structures to store the justifications. We have then decided to adapt the new algorithm proposed by Bessiere et al. [6] in order to deal with constraint relaxations. The details of the new dynamic arc consistency algorithm we propose that we call AC-3.1|DC, are presented in the next section.

3 AC-3.1|DC

The basic idea we took was to integrate the AC-3.1 into the AC|DC algorithm since that algorithm was based on AC-3. The problem with the AC|DC algorithm was that it relied solely on the AC-3 algorithm and did not keep support lists like DnAC4 or DnAC6 causing the restriction and relaxation of a constraint to be fairly time consuming. This is also the reason for its worst case time complexity of $O(ed^3)$. If AC-3.1 was integrated into the AC|DC algorithm, then by theory the worst case time complexity for a constraint restriction should be $O(ed^2)$. The more interesting question is whether this algorithm's time complexity can remain the same during constraint retractions. Following the same idea of AC|DC, the way our AC3.1|DC algorithm deals

with relaxations is as follows (see pseudo-code of the algorithm in figure 4). For any retracted constraint (k,m) between the variables k and m, we perform the following three phases :

1. An estimation (over-estimation) of the set of values that have been removed because of the constraint (k,m) is first determined by looking for the values removed from the domains of k and m that have no support on (k,m). Indeed, those values already suppressed from the domain of k (resp m) and which do have a support on (k,m), do not need to be put back since they have been suppressed because of another constraint. This phase is handled by the procedure **Propose**. The over-estimated values are put in the array *propagate_list*[k] (resp *propagate_list*[m]). In the procedure **Propose**, $dom[i]$ and $D[i]$ denote respectively the initial domain and current domain of i.

2. The above set is then propagated to the other variables. In this phase, for each value (k,a) (resp (m,b)) added to the domain of k (resp m) we will look for those values removed from the domain of the variables adjacent to k (resp m) supported by (k,a) (resp (m,b)). These values will then be propagated to the adjacent variables. The array *propagate_list* is used to contain the list of values to be propagated for each variable. After we propagate the values in *propagate_list*[i] of a given variable i, these values are removed from the array *propagate_list* and added to the array *restore_list* in order to be added later to the domain of the variable i. This way we avoid propagating the values more than once.

3. Finally a filtering procedure (the function **Filter**) based on AC-3.1 is then performed to remove, from the estimated set, the values which are not arc-consistent with respect to the relaxed problem.

The worst case time complexity of the first phase is $O(d^2)$. AC-3.1 is applied in the third phase and thus the complexity is $O(ed^2)$. Since the values in *propagate_list* are propagated only once, then the complexity of the second phase is also $O(ed^2)$. Thus the overall complexity of the relaxation is $O(ed^2)$.

In terms of space complexity, the arrays *propagate_list* and *restore_list* require $O(nd)$. AC-3.1 requires an array storing the resume point for each variable value with respect to any related constraint (in order to have $O(ed^2)$ time complexity). The space required by this array is $O(ed)$. If we add to this the $O(e+nd)$ space requirement of the traditional AC-3 algorithm, the overall space requirement is $O(ed+nd)$. Comparing to the three dynamic arc consistency algorithms we mentioned in the previous section, ours and DnAC-6 have a better compromise, in theory, between time and space costs as illustrated by table 1.

4 Experimentation

Theoretical comparison of the four dynamic arc consistency algorithms, presented in table 1, shows that AC3.1|DC has a better compromise between time and space costs than the other three algorithms. In order to see if the same conclusion can be said in practice, we have performed comparative tests respectively on randomly generated

```
Function Relax(k,m)
1. propagate_list ← nil
2. Remove (k,m) from the set of constraints
3. Propose(k,m,propagate_list)
4. Propose(m,k,propagate_list)
5. restore_list ← nil
6. Propagate(k,m,propagate_list,restore_list)
7. Filter(restore_list)
8. for all i ∈ V do
9.    domain_i ← domain_i ∪ restore_list[i]

Function Propose(i,j,propagate_list)
1. for all value a ∈ dom[i] − D[i] do
2.    support ← false
3.    for all b ∈ D[j] do
4.       if ((i a),(j b)) is satisfied by (i,j) then
5.          support ← true
6.       exit
7.       if support ← false then
8.          propagate_list[i] ← propagate_list[i] ∪ {a}

Function Propagate(k,m,propagate_list,restore_list)
1. L ← {k,m}
2. while L ≠ nil do
3.    i ← pop(L)
4.    for all j such that (i,j) ∈ the set of constraints
5.       S ← nil
6.       for all b ∈ dom[j] − (D[j] ∪ restore_list[j] ∪ propagate_list[j]) do
7.          for all a ∈ propagate_list[i] do
8.             if ((i a),(j b)) is satisfied by (i,j) then
9.                S ← S ∪ {b}
10.               exit
11.      if S ≠ nil do
12.         L ← L ∪ {j}
13.   propagate_list[j] ← propagate_list[j] ∪ S
14.   restore_list[i] ← restore_list[i] ∪ propagate_list[i]
15.   propagate_list[i] ← nil
```

Figure 4: Pseudo code of AC3.1|DC.

	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	$O(ed^2)$	$O(ed^2)$	$O(ed^3)$	$O(ed^2)$

Table 1: Comparison in terms of time and memory costs of the four algorithms.

dynamic CSPs and Temporal CSPs (that we call TCSPs). The criteria used to compare the algorithms are the running time needed and the memory space required by each algorithm to achieve the arc consistency. The experiments are performed on a Sun Sparc 10 station and all procedures are coded in C|C++.

4.1 Experimental tests on randomly generated dynamic CSPs

Given n the number of variables and d the domain size, each CSP instance is randomly obtained by generating n sets of d natural numbers. $\frac{n(n-1)}{2}$ constraints are then picked randomly from a set of arithmetic relations $\{=,\neq,<,\leq,>,\geq,\ldots\}$. The generated CSPs are characterized by their tightness, which can be measured, as shown in [19], as the fraction of all possible pairs of values from the domain of two variables that are not allowed by the constraint.

Figure 5 shows the performance in time performed by each arc consistency algorithm to achieve the arc consistency in a dynamic environment, as follows. Starting from a CSP having $n = 100$ variables, $d = 50$ and 0 constraints, restrictions are done by adding the relations from the random CSP until a complete graph (number of constraints=$\frac{n(n-1)}{2}$) is obtained. Afterwards, relaxations are performed until the graph is 50% constrained (number of constraints=$\frac{n(n-1)}{4}$). These tests are performed on various degrees of tightness to determine if one type of problem, (over-constrained, middle-constrained or under-constrained) favored any of the algorithms. As we can easily see, the results provided by AC-3.1|DC fares better than that of AC|DC and DnAC-4 in all cases. Also AC-3.1|DC algorithm is comparable if not better than DnAC6 (that has the best running time of the three dynamic arc consistency algorithms) as it can be seen in figure 5.

Table 2 shows the comparative results of DnAC-6 and AC3.1|DC in terms of memory space. The tests are performed on randomly generated CSPs in the same way as for the previous ones. As we can easily see, AC3.1|DC requires much less memory space than DnAC-6 especially for large problems with large domain size.

4.2 Experimental tests on randomly generated dynamic TCSPs

Before we present the tests on dynamic TCSPs let us first define TCSPs and dynamic TCSPs.

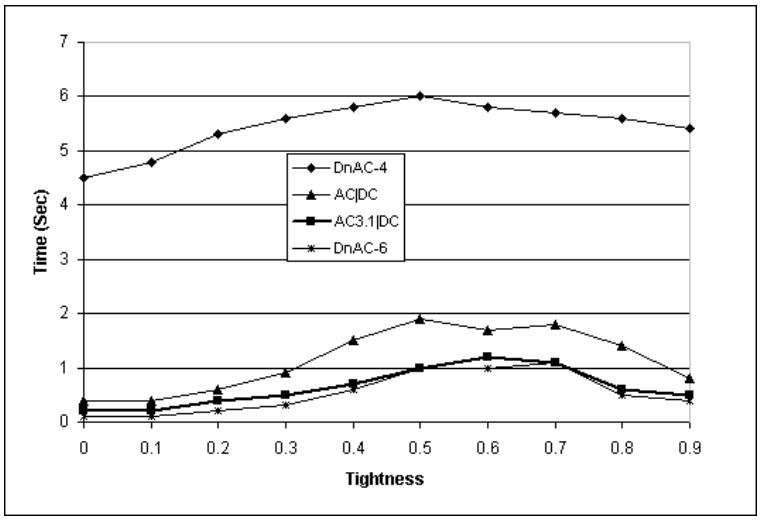

Figure 5: Comparative tests of the dynamic arc-consistency algorithms

4.2.1 TCSPs and dynamic TCSPs

We define a TCSP[2] as a CSP where constraints are disjunctions of Allen primitives [1] (see table 3 for the definition of the thirteen Allen primitives) and variables are temporal events defined on domains of numeric intervals. Indeed, in order to define a unique model for managing both numeric and symbolic temporal information, we have proposed the model TemPro [17, 16] based on Allen's interval Algebra and a discrete representation of time. TemPro transforms a problem involving numeric and symbolic time information into a TCSP. CSP search techniques can then be applied to check the consistency of the TCSP and to provide a solution in the case where the TCSP is consistent. A solution corresponds here to an assignment of temporal intervals to temporal events such that all the constraints (disjunction of Allen primitives) are satisfied.

Example 2

Consider the following typical temporal reasoning problem[3] :

1. *John, Mary and Wendy* **separately** *rode to the soccer game.*
2. *It takes John* **30 minutes**, *Mary* **20 minutes** *and Wendy* **50 minutes** *to get to the soccer game.*

[2]Note that this name and the corresponding acronym was used in [9]. The TCSP, as defined by Dechter et al, is a quantitative temporal network used to represent only numeric temporal information. Nodes represent time points while arcs are labeled by a set of disjoint intervals denoting a disjunction of bounded differences between each pair of time points.

[3]This problem is basically taken from an example presented by Ligozat, Guesgen and Anger at the tutorial : Tractability in Qualitative Spatial and Temporal Reasoning, IJCAI'01. We have added numeric constraints for the purpose of our work.

n	d	DnAC6	AC|DC3.1
500	100	354MB	58MB
500	90	312MB	54MB
500	80	276MB	49MB
500	70	221MB	43MB
500	60	186MB	34MB
500	50	154MB	28MB
500	40	112MB	25MB
500	30	75MB	17MB
500	20	54MB	14MB
500	10	27MB	10MB
500	5	16MB	8MB

Table 2: Comparative results in terms of memory cost

3. *John either* **started or arrived** *just as Mary* **started**.
4. *John either* **started or arrived** *just as Wendy* **started**.
5. *John left home* **between 7:00 and 7:10**.
6. *Mary arrived at the soccer game* **between 7:55 and 8:00**.
7. *Wendy left home* **between 7:00 and 7:10**.
8. *John's trip* **overlapped** *the soccer game.*
9. *Mary's trip took place* **during** *the game or else the game took place* **during** *her trip.*
10. *The soccer game* **starts** *at* **7:30** *and* **lasts 105 minutes**.

The above story includes numeric and qualitative information (words in boldface). Given this information, one important task is to represent and reason about such knowledge and answer queries such as : "is the above problem consistent ?", "what are the possible times at which Wendy arrived at the soccer game ?", ... etc. To do that, we first transform the above story to the TCSP shown in figure 6. There are four main events : John, Mary and Wendy are going to the soccer game respectively and the soccer game itself. These events are denoted respectively by J, M, W and Sc. Some numeric constraints specify the duration of the different events, e.g. *30 minutes is the duration of John's event*. Other numeric constraints describe the temporal windows in which the different events occur, e.g. *John left home between 7:00 and 7:10*. The numeric constraints of a given event are represented by the fourfold $[Start, End, Duration, Step]$ where $Start$, End, $Duration$ and $Step$ denote respectively the earliest start time, latest end time, duration and discretisation step of the event. In the case of John's event (J), this fourfold is equal to $[0, 40, 30, 1]$ (*7:00* is the time origin and is denoted by *0*). Given that the discretisation step is 1, the fourfold can be converted to the domain $\{(0\ 30), (1\ 31), \ldots, (10\ 40)\}$ as shown in figure 6. Symbolic constraints state the relative positions between events e.g. *John's trip overlapped the soccer game* which is denoted by $J\ O\ Sc$ in the figure. The information *John either started or arrived just*

as Wendy started is denoted by the relation $J \ \ ESS-M \ \ W$ which means that the start time of John and Wendy are equal or that the end time of John equals the start time of Wendy.

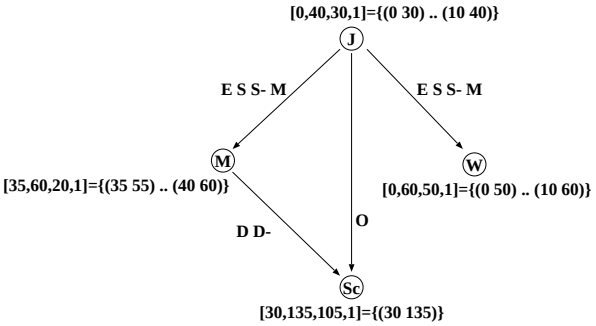

Figure 6: TCSP corresponding to example 2.

After the translation to the TCSP, consistency check is then enforced using a resolution method based on CSP techniques described as follows.

1. Arc consistency is first applied in order to reduce the size of the search space and to detect possible inconsistencies. Indeed, if the TCSP is not arc consistent then it is not consistent.

2. A backtrack search algorithm is then performed in order to look for a possible solution.

As we can see in figure 7, when applying arc consistency on the TCSP representing example 2, the domains of John's and Wendy's events are reduced. Indeed values $\{(0\,30)\ldots(4\,34)\}$ from the domain of John's event are deleted since they do not have support in Mike's event domain. As a consequence (after constraint propagation) values $\{(0\,50)\ldots(4\,54)\}$ from the domain of Wendy's event are deleted since they do not have support in John's event domain

In the case of dynamic CSPs, a constraint restriction corresponds to the addition of a constraint while a constraint relaxation is a retraction of a constraint. This is not exactly the case of Dynamic TCSPs (that we call DTCSPs). Indeed, a restriction of a temporal constraint is obtained by removing one or more Allen primitive from the disjunctive relation. A particular case is when the constraint is equal to the disjunction of the 13 primitives (we call it the universal relation I) which means that the constraint does not exist (there is no information about the relation between the two involved events). In this particular case, removing one or more Allen primitives from the universal relation is equivalent to adding a new constraint. Using the same way, a relaxation of a temporal constraint is obtained by adding one or more Allen primitives to a given constraint. A particular case is when the new constraint has 13 Allen primitives which is equivalent to the suppression of the constraint.

Relation	Symbol	Inverse	Meaning
X precedes Y	P	P-	X Y
X equals Y	E	E	X / Y
X meets Y	M	M-	X Y
X overlaps Y	O	O-	X Y
X during Y	D	D-	X Y
X starts Y	S	S-	X Y
X finishes Y	F	F-	Y X

Table 3: Allen Primitives

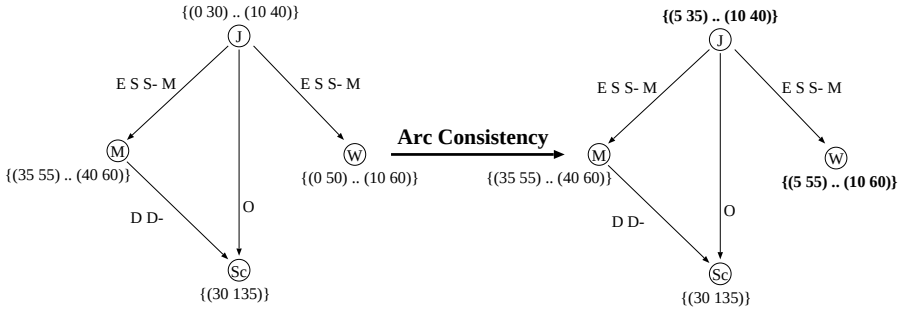

Figure 7: Applying Arc Consistency to the TCSP of Example 2.

Example 3

Starting from the arc consistent graph of figure 7, we assume now that we have the following constraint restrictions and relaxations :

1. Mary and Wendy arrived together but started at different times.
2. Retract the constraint between John and Mary.
3. Mary's event can also be overlapped by Wendy's one.

The first operation is a constraint restriction and corresponds to the addition of the relation $F \vee F^{\smile}$ between Mary and Wendy. Arc consistency is performed after this constraint restriction but the domains of the events are not changed. The second operation is a constraint relaxation and corresponds to the suppression of the constraint between John and Mary. Figure 8 shows the application of the above three actions to

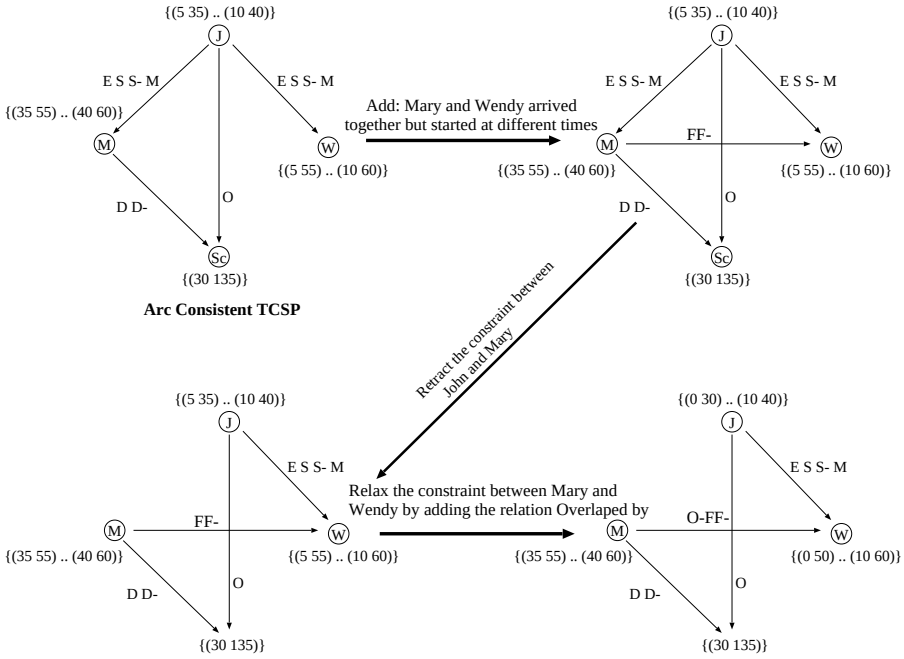

Figure 8: Restrictions and relaxations of the TCSP corresponding to example 3.

the arc consistent graph of example 2 presented in figure 7. The third operation is a constraint relaxation and corresponds to the addition of the primitive `Overlapped by` to the constraint between Mary's and Wendy's events. As a consequence, the values removed earlier from the domains of John's and Wendy's events are put back after applying our dynamic arc consistency algorithm.

As we stated in introduction, TCSPs enable the representation and solving of a wide variety of planning and scheduling applications. The following shows how to use a TCSP for solving a typical scheduling problem.

Example 4

The following example is taken from [16].

> *The production of five items A,B,C,D and E requires three mono processor machines M_1,M_2 and M_3. Each item can be produced using two different ways depending on the order in which the machines are used. The process time of each machine is variable and depends on the task to be processed. The following lists the different ways to produce each of the five items (the process time for each machine is mentioned in brackets) :*
>
> *item A:* $M_2(3),M_1(3),M_3(6)$ *or*
> $M_2(3),M_3(6),M_1(3)$

item B: $M_2(2), M_1(5), M_2(2), M_3(7)$ *or*
$M_2(2), M_3(7), M_2(2), M_1(5)$
item C: $M_1(7), M_3(5), M_2(3)$ *or*
$M_3(5), M_1(7), M_2(3)$
item D: $M_2(4), M_3(6), M_1(7), M_2(4)$ *or*
$M_2(4), M_3(6), M_2(4), M_1(7)$
item E: $M_2(6), M_3(2)$ *or*
$M_3(2), M_2(6)$

The above problem can easily be represented by a TCSP. A temporal event corresponds here to the contribution of a given machine to produce a certain item. For example, AM_1 corresponds to the use of machine M_1 to produce the item A, ... etc. 16 events are needed in total to produce the five items. Most of the qualitative information can easily be represented by the disjunction of Allen primitives. For example, the constraint (disjunction of two sequences) needed to produce item A is represented by the following three relations :

$AM_2 \ \ P \vee M \ \ AM_1$
$AM_2 \ \ P \vee M \ \ AM_3$
$AM_1 \ \ P \vee M \vee P^{\smile} \vee M^{\smile} \ \ AM_3$

However the translation to Allen relations of the disjunction of the two sequences required to produce item B needs an additional event (EVT_{17}) and is represented by the following seven binary relations :

$BM_{21} \ \ P \vee M \ \ BM_1$
$BM_{21} \ \ P \vee M \ \ BM_3$
$BM_{21} \ \ P \vee M \ \ BM_{22}$
$BM_1 \ \ P \vee P^{\smile} \ \ BM_3$
$BM_1 \ \ S \vee F \ \ EVT_{17}$
$BM_3 \ \ S \vee F \ \ EVT_{17}$
$BM_{22} \ \ D \ \ EVT_{17}$

56 binary relations are needed in total to represent all the qualitative information.

The following (see figure 9) is the solution to the above problem provided by our solving method. Note that this solution is optimal[4] but not unique.

4.2.2 Tests on Consistent and Inconsistent DTCSPs

All tests are performed on randomly generated DTCSPs. Three classes of instances, corresponding to 3 types of tests, are generated as follows.

[4]The total processing time of all machines needed to produce the five items, 26 seconds, is minimal

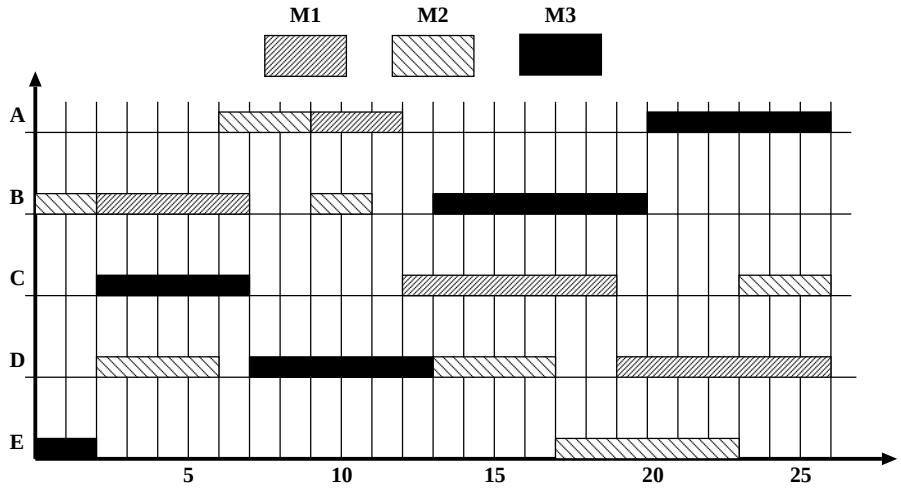

Figure 9: Optimal solution provided by the CSP based method.

case 1 : actions correspond to additions of constraints. $C = N(N-1)/2$ (constraints are added until a complete graph is obtained). N and C are respectively the number of variables and the number of constraints.

case 2 : Actions can be additions or relaxations of constraints.
$C = N(N-1)/2$ additions $+N(N-1)/4$
retractions (the final TCSP will have $N(N-1)/4$ constraints).

case 3 : This case is similar to case 1 but with inconsistent DTCSPs. Indeed, in the previous two cases the generated DTCSPs are consistent. In this last case, constraints are added until an arc inconsistency and thus a global inconsistency is detected (the inconsistency is detected if one variable domain becomes empty). Retractions are then performed until the arc-consistency is restored.

In each of the above three cases the constraints added are taken from TCSPs (consistent or inconsistent) randomly generated as described by the following subsections. Each generated problem is characterized by two parameters : N the number of events and *Horizon* the time before which all events must be processed. In the following, we will describe the generation of consistent and inconsistent problems.

Generation of Consistent TCSPs

Consistent problems of size N are those having at least one complete numeric solution (set of N numeric intervals satisfying all the constraints of the problem). Thus, to generate a consistent problem we first randomly generate a numeric solution and then add other numeric and symbolic information to it. More precisely, the generation is performed by the following three steps.

Step 1. Generation of the numeric solution

Randomly pick N pairs (x,y) of integers such that $x < y$ and $x,y \in [0, \ldots, Horizon]$. This set of N pairs forms the initial solution where each pair corresponds to a time interval.

Step 2. Generation of numeric constraints

For each interval (x,y) randomly pick an interval contained within $[0, \ldots, Horizon]$ and containing the interval (x,y). This newly generated interval defines the temporal window of the corresponding variable. From this temporal window, we generate the domain of the corresponding event.

Step 3. Generation of symbolic constraints

Compute the basic Allen primitives that can hold between each interval pair of the initial solution. Add to each relation a random number belonging to the interval $[0,Nr]$ $(1 \leq Nr \leq 13)$ of chosen Allen primitives.

Example 4

Let us assume we want to generate a consistent problem with $N = 3$ and $Horizon = 10$.

1. First a numeric solution is generated : S = $\{$(1 4), (2 8), (5 7)$\}$.
2. Numeric constraints (domains of the three events) are then randomly generated from the numeric solution.

Numeric Interval		Temporal Window		Domains of events
(1 4)	→	[2,9]	→	{(2 5)…(6 9)}
(2 8)	→	[2,10]	→	{(2 8)…(4 10)}
(5 7)	→	[3,8]	→	{(3 5)…(6 8)}

3. Allen primitives are then computed from the pairs of intervals of the numeric solution :

   (1 4) *and* (2 8) → Overlaps (O)
   (1 4) *and* (5 7) → Preceeds (P)
   (2 8) *and* (5 7) → During inverse ($D^{\smile}$)

   and finally other Allen primitives are randomly chosen from the list of the 13 basic relations and added to the above primitives.

   $O + PM \rightarrow POM$
   $P + DD^{\smile}EO \rightarrow DD^{\smile}EOP$
   $D^{\smile} + DEFSP \rightarrow FSDD^{\smile}PE$

Generation of Inconsistent TCSPs

Each inconsistent problem of size N is generated using the following steps.

Step 1. Generation of numeric constraints

Randomly pick N pairs of ordered values (x,y) such that $x,y \in [0, \ldots, Horizon]$. x and y are respectively considered the earliest start time and the latest end time of a given event. For each pair of value (x,y), randomly pick a number $d \in [1 \ldots y-x]$. d is considered the duration of the event.

Step 2. Generation of symbolic constraints

Randomly generate C constraints between the N events where $C \in [1 \ldots \frac{n(n-1)}{2}]$ ($C = \frac{n(n-1)}{2}$ in the case of a complete graph of constraints). Each constraint C is a disjunction of a random number nb ($nb \in [1 \ldots 13]$) of relations chosen randomly from the set of the 13 Allen primitives.

Step 3. Consistency check of the generated problem

Perform a backtrack search algorithm to check the consistency of the problem. If a solution exists then the generated problem is consistent otherwise goto step 1.

4.2.3 Test Results

Figure 10 shows the results of tests corresponding to case 1. As we can easily see, the results provided by DnAC-6 and AC-3.1|DC are better than the ones provided by AC|DC and DnAC-4 (which is too slow to appear on the chart). This can be explained by the worst case time complexity AC|DC has ($O(ed^3)$ comparing to $O(ed^2)$ for AC-3.1|DC and DnAC-6 as shown in table 1). On the other hand, while DnAC-4 has the same worst case time complexity as AC-3.1|DC and DnAC-6, it inherits the bad running time in practice of AC-4 as we mentioned in Section 2.1.

Since DnAC-6 requires much more memory space than AC-3.1|DC in practice as we have shown in table 2, this latter is the algorithm of choice in the case of constraint additions.

Figure 11 and 12 correspond to case 2 and case 3 respectively. DnAC-4 and DnAC-6 have better performance in this case than AC3.1|DC and AC|DC (the running time of AC|DC is very slow comparing to the other 3 algorithms due to its worst case time complexity as we reported above, thus it does not appear on the chart). However, since AC3.1|DC does not require a lot of memory space as shown in 1 and 2, it has less limitations than DnAC-4 and DnAC-6 in terms of space requirements especially in the case of problems having large domain sizes.

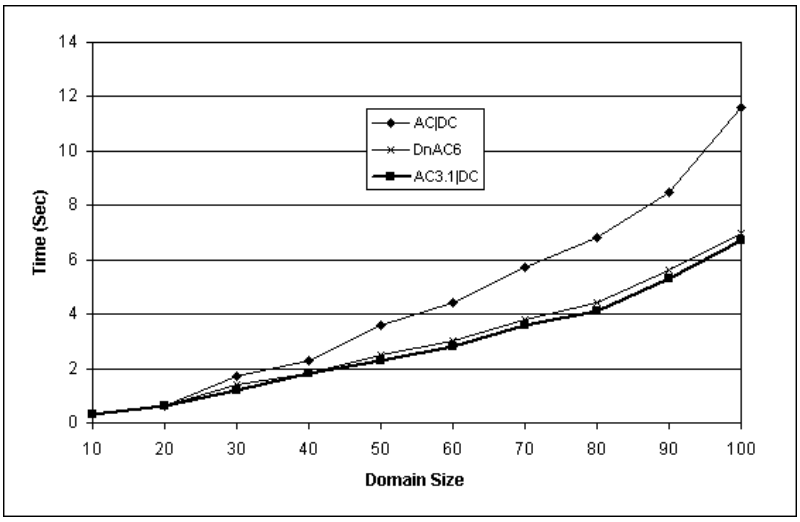

Figure 10: Experimental Tests on random DTCSPs : case 1

5 Conclusion and future work

In this paper we have presented a new algorithm, that we call AC3.1|DC, for maintaining the arc consistency of a CSP in a dynamic environment. Our proposed algorithm maintains the worst case time and space cost, in theory, of the best dynamic arc consistency algorithm DnAC-6. In addition, experimental results on randomly generated CSPs show that AC3.1|DC has comparable running time to DnAC-6 but outperforms this latter algorithm in the case of space cost. This is very significant especially for large size problems where space can become a serious issue. We have also conducted experimental tests on randomly generated dynamic temporal CSPs. These particular case of CSPs represent a wide variety of applications including dynamic scheduling and planning, Geographic Information Systems (GIS) and temporal databases. Our goal here is to see how does AC3.1|DC behave in the case of numeric and symbolic temporal constraints. The result of the tests show that AC3.1|DC has comparable running time to DnAC-6 in the case of constraint addition but is less efficient in the case of constraint relaxation.

In the near future we are looking to integrating our dynamic arc consistency algorithm during the backtrack search phase in order to handle the addition and relaxation of constraints dynamically during the search. For instance, if a value from a variable domain is deleted during the backtrack search, would it be worthwhile to use a DAC algorithm to determine its effect or would it be more costly than just continuing on with the backtrack search. Another perspective is to use our dynamic arc consistency algorithm for solving conditional CSPs. Conditional CSPs are CSPs containing variables whose existence depends on the values chosen for other variables. In this case our algorithm AC3.1|DC will have to maintain the arc consistency of the CSP any time new variables and their corresponding constraints are added.

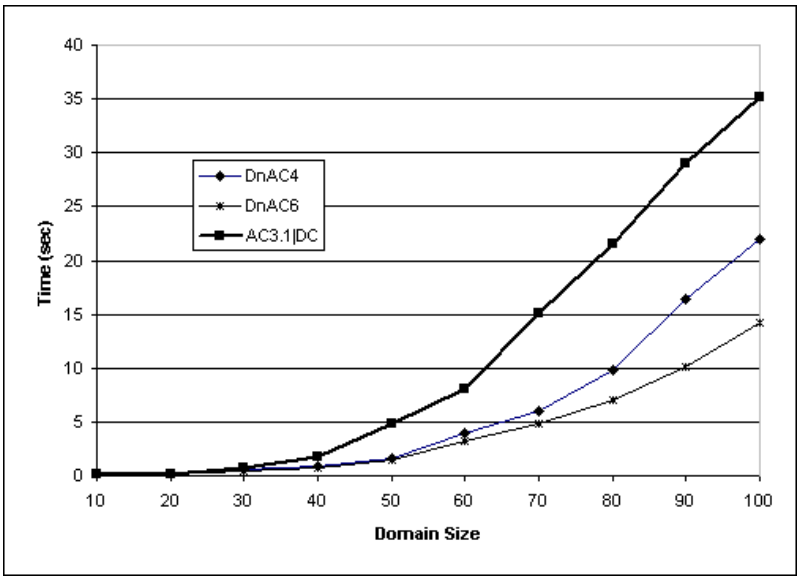

Figure 11: Experimental Tests on random DTCSPs : case 2

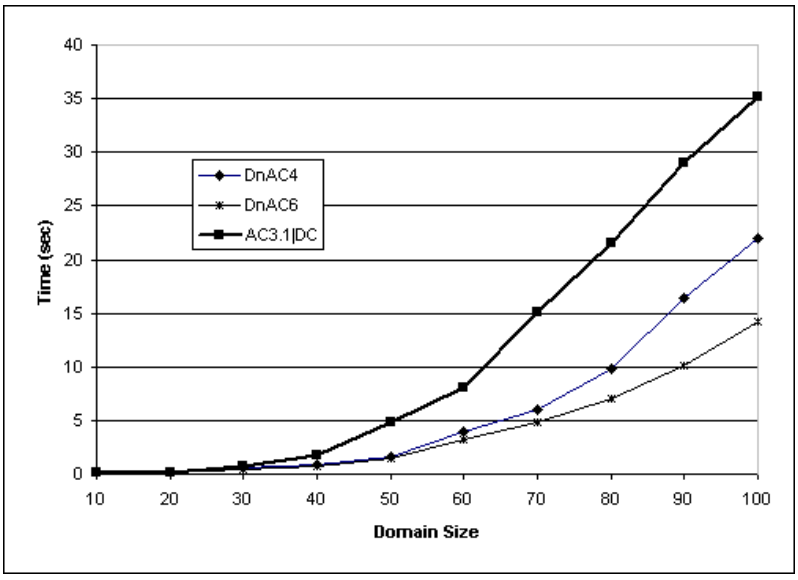

Figure 12: Experimental Tests on random DTCSPs : case 3

References

[1] J.F. Allen. Maintaining knowledge about temporal intervals. *CACM*, 26(11):832–843, 1983.

[2] K.R. Apt. ewblock *Principles of Constraint Programming*. ewblock Cambridge University Press, 2003.

[3] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *AAAI'91*, pages 221–226, Anaheim, CA, 1991.

[4] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[5] C. Bessière, E. Freuder, and J.C. Regin. Using inference to reduce arc consistency computation. In *IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[6] C. Bessière, J. C. Régin, R.H.C. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[7] R. Debruyne. Les algorithmes d'arc-consistance dans les csp dynamiques. *Revue d'Intelligence Artificielle*, 9:239–267, 1995.

[8] R. Dechter. ewblock *Constraint Processing*. ewblock Morgan Kaufmann, 2003.

[9] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[10] Thom Fruhwirth and Slim Abdennadher. ewblock *Essentials of Constraint Programming*. ewblock Springer, 2003.

[11] P Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[12] P Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[13] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[14] A. K. Mackworth and E. Freuder. The complexity of some polynomial network-consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[15] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[16] M. Mouhoub. Reasoning with numeric and symbolic time information. *Artificial Intelligence Review*, 21:25–56, 2004.

[17] M. Mouhoub, F. Charpillet, and J.P. Haton. Experimental Analysis of Numeric and Symbolic Constraint Satisfaction Techniques for Temporal Reasoning. *Constraints: An International Journal*, 2:151–164, Kluwer Academic Publishers, 1998.

[18] B. Neuveu and P. Berlandier. Maintaining Arc Consistency through Constraint Retraction. In *ICTAI'94*, pages 426–431, 1994.

[19] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. 11th ECAI*, pages 125–129, Amsterdam, Holland, 1994.

[20] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *IJCAI'93*, pages 239–245, Chambery, France, 1993.