AI Communications 00 (2004) 1–11
IOS Press

Systematic versus non systematic techniques for solving temporal constraints in a dynamic environment

Malek Mouhoub
Department of Computer Science, University of Regina, 3737 Wascana Parkway, Regina SK, Canada, S4S 0A2
E-mail: mouhoubm@cs.uregina.ca

Abstract. A main challenge when designing constraint based systems in general and those involving temporal constraints in particular, is the ability to deal with constraints in a dynamic and evolutive environment. That is to check, anytime a new constraint is added, whether a consistent scenario continues to be consistent when a new constraint is added and if not, whether a new scenario satisfying the old and new constraints can be found. We talk then about on line temporal constraint based systems capable of reacting, in an efficient way, to any new external information during the constraint resolution process. In this paper, we will investigate the applicability of systematic versus approximation methods for solving incremental temporal constraint problems. In order to handle both numeric and symbolic constraints, the systematic method is based on constraint propagation performed at both the qualitative and quantitative levels. The approximation methods are respectively based on stochastic local search and genetic algorithms. Experimental evaluation of the performance in time and the quality of the solution returned (number of violated constraints) of the different techniques has been performed on randomly generated temporal constraint problems. The results favour the exact method for problems with reasonable size while the approximation techniques are the methods of choice for very large problems in the case where we want to trade the quality of the solution for the process time. Indeed, while the approximation methods are faster for large problems, they do not guarantee, in general, the completeness of the solution returned.

Keywords: Temporal reasoning, constraint propagation, stochastic local search, genetic algorithms

1. Introduction

In any constraint satisfaction problem (CSP) there is a collection of variables which all have to be assigned values from their discrete domains, subject to specified constraints. Because of the importance of these problems in so many different fields, a wide variety of techniques and programming languages from artificial intelligence, operations research and discrete mathematics are being developed to tackle problems of this kind. An important issue when dealing with a constraint satisfaction problem, in the real world, is the ability to maintaining the consistency of the problem anytime a new constraint is added. Indeed, this change may affect the solution already obtained and respecting the old constraints. Our goal, in this paper, is to maintain the consistency in a dynamic environment of a constraint satisfaction problem involving qualitative and quantitative temporal constraints. This is of practical relevance since it is often required to check whether a solution to a CSP involving temporal constraints continues to be a solution when a new constraint is added and if not, whether a new solution satisfying the old and new constraints can be found. In scheduling problems, for example, a solution corresponding to an ordering of tasks to be processed can no longer be consistent if a given machine becomes unavailable. We have then to look for another solution satisfying the old constraints and taking into account the new information. In a previous work [1,2], we have developed a temporal model, TemPro, based on Allen's interval algebra [3] and a discrete representation of time, to express numeric and symbolic time information in terms of qualitative and quantitative temporal constraints. More precisely, TemPro translates an application involving temporal information into a binary Constraint Satisfaction Problem[1] where variables are temporal events defined on domains of numeric intervals and binary constraints between variables correspond to disjunctions of Allen

[1] A binary CSP involves a list of variables defined on finite domains of values and a list of binary relations between variables.

primitives. We call it Temporal Constraint Satisfaction Problem (TCSP).[2] The solution method for solving the TCSP is based on constraint propagation and requires two stages. In the first stage, local consistency is enforced by applying the path consistency on symbolic relations and the arc consistency on variable domains. A backtrack search algorithm is then performed in the second stage to check the consistency of the TCSP by looking for a possible solution. Note that for some TCSPs local consistency implies the consistency of the TCSP network [5]. The backtrack search phase can be avoided in this case.

In order to maintain the consistency of a TCSP (existence of a solution) in a dynamic environment, we propose three different resolution techniques. The first one is an exact method based on the above constraint propagation techniques that we have adapted in order to handle the addition of constraints in an efficient way. The second technique is based on stochastic local search. Indeed, the underlying local search paradigm is well suited for recovering solutions after local changes (addition of constraints) of the problem occur. The third method, based on genetic algorithms, is similar to the second one except that the search is multi-directional and maintains a list of potential solutions (population of individuals) instead of a single one. This has the advantage to allow the competition between solutions of the same population which simulates the natural process of evolution. The main difference between the three methods is that the first one is a systematic search technique that guarantees the completeness of the solution provided which is not the case of the other two approximation methods. This however makes the approximation methods faster for large size problems, as shown by the experimental comparison we conducted and that we present in this paper. Note that related work on using local search methods for solving temporal constraint problems has been reported in [6]. This latter work consists of solving static Interval Algebra (IA) networks (representing symbolic temporal information) using a local search approach while our goal is to solve temporal networks involving numeric and symbolic information, in a dynamic environment.

[2]Note that this name and the corresponding acronym was used in [4]. 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.

The rest of the paper is organized as follows. In the next section, we will present through an example, the different components of our model TemPro. The three methods for maintaining the consistency of TCSPs in a dynamic environment are then presented respectively in Sections 3, 4 and 5. Section 6 is dedicated to the experimental evaluation, of the methods we propose, on randomly generated dynamic TCSPs. Concluding remarks and possible perspectives of our work are then presented in Section 7.

2. CSP-based representation of numeric and symbolic temporal constraints: the model TemPro

TemPro [1,2] transforms any problem under qualitative and quantitative constraints into a binary CSP where constraints are disjunctions of Allen primitives [3] (see Table 1 for the definition of the Allen primitives) and variables, representing temporal events, are defined on domains of time intervals. We call this later a Temporal Constraint Satisfaction Problem (TCSP). Each event domain (called also temporal window) contains the Set of Possible Occurrences (SOPO) of numeric intervals the corresponding event can take. The SOPO is the numeric constraint of the event. It is expressed by the fourfold: [*earliest_start*, *latest_end*, *duration*, *step*] where: *earliest_start* is the earliest start time of the event, *latest_end* is the latest end time of the event, *duration* is the duration of the event and *step* is the discretization step corresponding to the number of time units between the start time of two adjacent intervals belonging to the event domain. The discretization

Table 1
Allen primitives

Relation	Symbol	Inverse	Meaning
X before Y	B	Bi	X Y
X equals Y	E	E	X / Y
X meets Y	M	Mi	X Y
X overlaps Y	O	Oi	X Y
X during Y	D	Di	X Y
X starts Y	S	Si	X Y
X finishes Y	F	Fi	Y X

step is a parameter provided by the user. For some applications, the consistency of the problem can depend on the discretization step. In this particular case, if the solution is not found, the user can decrease the value of the step and run again the solving algorithm. Decreasing the discretization step will however increase the complexity of the problem. Indeed, the total number of combinations (potential solutions) of a TCSP is D^N where N is the number of variables and D their domain size.

$$D = Max_{1 \leqslant i \leqslant N} \left(\frac{sup_i - inf_i - d_i}{s_i} \right),$$

where sup_i, inf_i, d_i and s_i are respectively the latest end time, earliest start time, duration and step of a given event Evt_i. As we can easily see, decreasing the value of s_i will increase the domain size d which increases the total number of possibilities of the search space. Note that *begintime*, *endtime*, *duration* and *step* can be constant values or variables taking values from a discrete and finite domain. We can also use constraints, in the form of equations or inequalities, in order to restrict the values these variables can take. To illustrate the different components of the model TemPro, let us consider the following scheduling problem.[3]

Example 1. 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 goal here is to find a possible schedule of the different machines to produce the five items and respecting all the constraints of the problem. In the following, we will describe how is the above problem transformed into a TCSP using our model TemPro. Figure 1 illustrates the graph representation of the TCSP corresponding the constraints needed to produce items A and B. We assume that items A and B should be produced within 25 and 30 units of time respectively. 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. In the particular case of item B, machine M_2 is used twice. Thus there are two corresponding events: BM_{21} and BM_{22}. 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:

$AM_2\ B \vee M\ AM_1$
$AM_2\ B \vee M\ AM_3$
$AM_1\ B \vee M \vee Bi \vee Mi\ AM_3$

However the translation to Allen relations of the disjunction of the two sequences required to produce item B needs a 3-ary relation involving BM_1, BM_{22} and BM_3. This relation states that BM_{22} should occur between BM_1 and BM_3. Since our temporal network handles only binary relations, the way we use to represent this kind of 3-ary relation is as follows: we create an additional event (Evt_1) and represent the constraints for producing item B as shown in Fig. 1. The duration X of Evt_1 is greater (or equal) than the sum of the durations of BM_1, BM_{22} and BM_3. Figure 2 illustrates the solution to the above problem provided

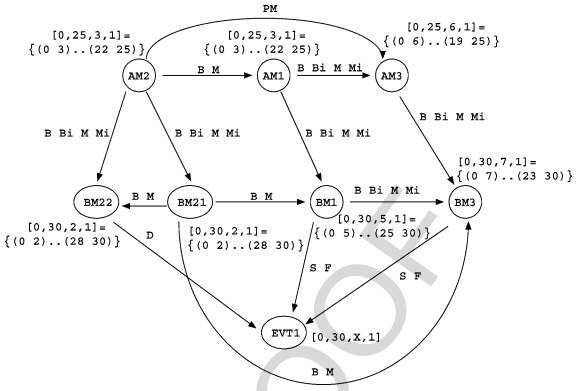

Fig. 1. TCSP corresponding to a subset of the problem presented in Example 1.

[3]This problem is taken from [7].

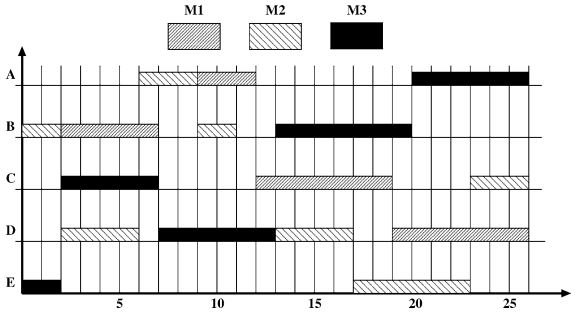

Fig. 2. Optimal solution provided by the constraint propagation based method.

by the constraint propagation based method we have described in introduction. Note that this solution is optimal[4] but not unique.

3. Dynamic maintenance of the consistency using constraint propagation techniques

Before we present the solution method for maintaining the consistency of temporal constraints in a dynamic environment, let us introduce the notion of dynamic temporal constraint satisfaction.

3.1. Dynamic temporal constraint satisfaction problem (DTCSP)

A dynamic temporal constraint satisfaction problem (DTCSP) is a sequence of static TCSPs: $TCSP_0, \ldots, TCSP_i, TCSP_{i+1}, \ldots, TCSP_n$ each resulting from a change in the preceding one imposed by the "outside world". This change corresponds to a constraint restriction or relaxation. In this paper we will focus only on constraint restrictions. More precisely, $TCSP_{i+1}$ is obtained by performing a restriction on $TCSP_i$. We consider that $TCSP_0$ (initial TCSP) has an empty set of constraints. A restriction can be obtained by removing one or more Allen primitives from a given constraint. A particular case is when the initial 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.

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

3.2. The solution method

The pseudo-code of the solution method is presented in Fig. 3. Given that we start from a consistent TCSP, the goal of the method we present here consists of maintaining the consistency (existence of a solution) anytime a new constraint is added. The method works as follows. We first compute the intersection of the new constraint with the corresponding constraint in the consistent graph. We call *updated constraint* the result of the intersection. If *updated constraint* is an empty relation then the new constraint cannot be added (as it will violate in this case the consistency of the constraint graph) otherwise we replace the current constraint of the graph by *updated constraint*. If *updated constraint* is inconsistent with the current solution obtained for the problem then we perform the following steps.

1. **Numeric → symbolic conversion:** check the compatibility of the updated constraint and the numeric domains of the two variables involved by the constraint. This is accomplished by performing the numeric → symbolic conversion on the updated constraint. If the updated constraint becomes empty then it cannot be added. This procedure works as follows: from the domains of the two variables involved by the constraint, we can extract the corresponding symbolic relation. An intersection of this relation with the updated constraint will reduce the size of the latter which simplifies the size of the original problem. We can get the symbolic relation between two events by extracting Allen primitives from all possible pairs of intervals belonging to the Cartesian product of the domains of both events. This, however, requires

$$O\left(Max_{1 \leqslant i \leqslant N}\left(\frac{sup_i - inf_i - d_i}{s_i}\right)^2\right)$$

in time where N is the number of events and

$$Max_{1 \leqslant i \leqslant N}\left(\frac{sup_i - inf_i - d_i}{s_i}\right)$$

is the size of the largest domain. This method can be very expensive for large size domains of events. Alternately, we have defined an incomplete method that extracts most of the primitives within a relation between each pair of events in constant time reducing the complexity to $O(C)$ where C is the total number of constraints. The

Function Restrict(i,j)
1. $t \leftarrow new\_constraint \bigcap C_{ij}, updated\_list \leftarrow \{(i,j)\}$
2. **if** $(t = \emptyset)$ **then**
3. return "Constraint cannot be added"
4. **else**
5. $C_{ij} \leftarrow t$
6. **if** $\neg ConsistentWithCurrentSol(updated\_list)$ **then**
7. **if** $\neg NumSymb(updated\_list)$ **then**
8. return "Constraint cannot be added"
9. **if** $\neg DPC(updated\_list)$ **then**
10. return "Constraint cannot be added"
11. **if** $\neg DAC(updated\_list)$ **then**
12. return "Constraint cannot be added"
13. **if** $\neg DSearch(updated\_list)$ **then**
14. return "Constraint cannot be added"

Function DAC(updated_list)
1. $Q \leftarrow updated\_list$
2. $AC \leftarrow true$
3. (list initialized to the constraints updated after PC)
4. **While** $Q \neq Nil$ **Do**
5. $Q \leftarrow Q - \{(x,y)\}$
6. **if** $Revise(x,y)$ **then**
7. **if** $Dom(x) \neq \emptyset$ **then**
8. $Q \leftarrow Q \sqcup \{(k,x) \mid (k,x) \in R \wedge k \neq y\}$
9. **else**
10. return $AC \leftarrow false$
8. **End-While**

Function $Revise(x,y)$
1. $REVISE \leftarrow false$
2. **For** each interval $a \in SOPO_x$ **Do**
3. **If** $\neg compatible(a,b)$ **for each interval** $b \in SOPO_y$ **Then**
4. remove a from $SOPO_x$
5. $Revise \leftarrow true$
6. **End-If**
7. **End-For**

Function $DPC(updated\_list)$
1. $PC \leftarrow false$
2. $L \leftarrow updated\_list$
3. **while** $(L \neq \emptyset)$ **do**
4. select and delete an (x,y) from L
5. **for** $k \leftarrow 1$ to n, $k \neq x$ and $k \neq y$ **do**
6. $t \leftarrow C_{xk} \bigcap C_{xy}.C_{yk}$
7. **if** $(t \neq C_{xk})$ **then**
8. $C_{xk} \leftarrow t$
9. $C_{kx} \leftarrow INVERSE(t)$
10. $L \leftarrow L \cup \{(x,k)\}$
11. $updated\_list \leftarrow updated\_list \cup \{(x,k)\}$
12. $t \leftarrow C_{ky} \bigcap C_{kx}.C_{xy}$
13. **if** $(t \neq C_{ky})$ **then**
14. $C_{yk} \leftarrow INVERSE(t)$
15. $L \leftarrow L \cup \{(k,y)\}$
16. $updated\_list \leftarrow updated\_list \cup \{(y,k)\}$

Fig. 3. Dynamic consistency algorithm.

method consists of using the information concerning the lower bound, upper bound and duration of the event temporal window instead of its occurrences. Let us consider e_i and e_j two events, r_{ij} the symbolic relation between them (initially set to the disjunction of the 13 Allen primitives), and inf_i, inf_j, sup_i, sup_j, d_i and d_j respectively the earliest start time of e_i, earliest start time of e_j, latest end time of e_i, latest end time of e_j, duration of e_i and duration of e_j. Our method, denoted $NumSymb$ in Fig. 3, is defined by the following rules:

1. if $inf_i > sup_j$ then $r_{ij} \leftarrow$ Bi,
2. if $sup_i < inf_j$ then $r_{ij} \leftarrow$ B,
3. if $d_i < d_j$ then remove { E, Si, Fi, Di } from r_{ij},
4. if $d_i > d_j$ then remove { E, S, F, D } from r_{ij},
5. if $d_i = d_j$ then remove { D, Di, S, Si, F, Fi } from r_{ij},
6. if $inf_i + d_i > sup_j - d_j$ then remove { M, B } from r_{ij},
7. if $sup_i - d_i < inf_j + d_j$ then remove { Mi, Bi } from r_{ij},
8. if $inf_i > sup_j - d_j$ then remove { E, B, M, S, Si, O, Di } from r_{ij},
9. if $inf_i + d_i > sup_j$ then remove { E, B, M, F, Fi, D } from r_{ij},
10. if $sup_i < inf_j + d_j$ then remove { F, Fi } from r_{ij},
11. if $sup_i - d_i < inf_j$ then remove { S, Si, E } from r_{ij}.

2. **Dynamic path consistency:** perform dynamic path consistency (DPC) in order to propagate the update of the constraint to the rest of the graph. If the resulting graph is not path consistent then the new constraint cannot be added.
3. **Dynamic arc consistency:** perform dynamic arc consistency (DAC) starting with the updated constraints. If the new graph is not arc consistent then the new constraint cannot be added.
4. **Incremental backtracking:** perform the backtrack search algorithm in order to look for a new solution to the problem. The backtrack search will start here from the point (resume point) it stopped in the previous search when it succeeded to find a complete assignment satisfying all the constraints. This way the part of the search space already explored in the previous searches will be avoided. The search will explore the rest of the

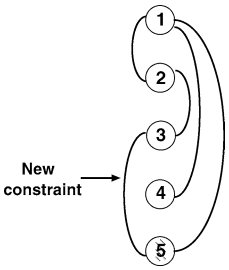

Fig. 4. Dynamic backtracking after adding a new constraint.

search space. If a solution is found then the point where the backtrack search stopped is saved as new resume point and the new solution is returned. Otherwise the graph is inconsistent (when adding the new constraint). The new constraint cannot be added. Note that when the backtrack start from the resume point, it first processes the source of conflict and proceed further using the dynamic backtracking method [8] as follows. Let us consider the example illustrated in Fig. 4. Here we have the first 5 events of the search space and we assume that we are adding a new constraint between events 3 and 5. We also assume that this constraint is violating the assignment given to both events. The backtrack search algorithm will start first by assigning a new value to event 5 that is consistent with the values assigned to events 3 and 1. If such value is found we proceed further to event 6 otherwise we backtrack ("backjump") to event 3, assign a new value to this event and proceed forward. Note that, before backtracking to event 3, event 4 will be moved above event 3 as both events do not share a constraint. This will avoid any superfluous work.

DPC it the path consistency algorithm PC-2 [9] we have adapted to handle constraint additions in an incremental way [10]. A similar dynamic path consistency algorithm has been proposed [11]. DAC is the new arc consistency algorithm AC-3 [12,13] we have adapted for temporal constraints in a dynamic environment. A detailed description of DAC can be found in [14]. The intersection binary operator [3], denoted by $\bigcap$ in the Restrict and DPC functions of Fig. 3, is the ordinary set intersection of the Allen primitives composing each symbolic constraint. In other words, the intersection of two symbolic relations such as $P \vee D \vee M \vee O$ and $D \vee M \vee S$ is simply the Allen primitives shared by both, namely $D \vee M$. The composition of two symbolic relations R_i and R_j, denoted by $R_i . R_j$ as shown in Fig. 3, is computed using the distributive law and the composition table of Allen primitives [3] as follows. Let us assume $R_i = Mi \vee Oi$ and $R_j = B \vee M$.

$$\begin{aligned}(Mi \vee Oi) . (B \vee M) \\ = (Mi.B) \vee (Mi.M) \vee (Oi.B) \vee (Oi.M) \\ = E \vee B \vee M \vee S \vee Si \vee O \vee Di \vee Fi\end{aligned}$$

$(Mi.B)$, $(Mi.M)$, $(Oi.B)$ and $(Oi.M)$ are computed using the composition table of the 13 Allen primitives [3].

4. Dynamic maintenance of the consistency using stochastic local search

In this section we present the way to solve dynamic TCSPs using a stochastic local search method. This technique is based on a common idea known under the notion of local search. In local search, an initial configuration (potential solution) is generated randomly and the algorithm moves from the current configuration to a neighborhood configurations until a complete solution (solution satisfying all the constraints) has been found or a maximum number of iterations is reached.

In the case of a dynamic environment, anytime a new constraint is added, the stochastic local search algorithm works as follows:

1. If the new constraint does not conflict with the solution obtained so far, return "problem consistent when adding the new constraint".
2. Restart the search from the configuration corresponding to the last solution obtained, and iterates until a new solution respecting the old constraints and the new one is found, or the maximum number of iterations is reached (in which case the new constraint is rejected).

The pseudo-code in Fig. 5 illustrates the local search strategy using the Min-Conflict-Random-Walk technique (MCRW). Anytime a new constraint is added, this technique restarts the search by choosing randomly a conflicting event in the configuration, i.e., the event that is involved in any unsatisfied constraint. In this particular case, the choice is done among the two events involved by the new constraint. The method will then pick a value (numeric interval) which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of vi-

```
procedure DYN_MCRW(Max_Moves,p)
Begin
  while there is a new constraint to be processed do
    // sol is the current solution
    // (set of pairs <event,interval>)
    if the new constraint does not conflict with sol then
      return sol
    endif
    nb_moves ← 0;
    while eval(sol) > 0 & nb_moves < Max_Moves do
      if probability p verified then
        choose randomly an event evt in conflict;
        choose randomly an interval intv for evt;
      else
        choose randomly an event evt in conflict;
        choose an interval intv that minimizes
        the number of conflicts for evt;
      endif
      if intv ≠ current value of evt then
        assign intv to evt;
        nb_moves ← nb_moves+1;
      endif
    endwhile
  return sol
  endwhile
End
```

Fig. 5. Pseudo-code of the MCRW method.

olated constraints (the current value of the event is picked only if all the other values increase the number of violated constraints). In order to go beyond a local optimum, we use a random-walk strategy as follows: for a given conflicting event, this strategy picks randomly a value with probability p, and apply the Min Conflict heuristic with probability $1 - p$. In the worst case, the time cost required in each move corresponds to the time needed to determine the value that minimizes the number of violated constraints. To do so, we have to compute the number of conflicts for each value and take the value having the minimum number of conflicts. Computing the number of conflicts for a given value is done by checking the consistency of the value with all the other event values. This costs $O(N)$ in the worst case where N is the total number of variables. Thus, the total cost for computing the number of conflicts for all the values is $O(ND)$ where D the size of the largest event domain. Since the number of moves, denoted by Max_Moves in 5, is a constant number given in input then the total complexity in time of MCRW is also $O(ND)$. In our case,

$$D = Max_{1 \leqslant i \leqslant N} \left(\frac{sup_i - inf_i - d_i}{s_i} \right),$$

where sup_i, inf_i, d_i and s_i are respectively the latest end time, earliest start time, duration and step of a given event Evt_i. The evaluation function of a complete assignment (or potential solution), denoted by $eval(sol)$ where sol is the potential solution, corresponds to the quality of the solution and is defined by the number of violated constraints.

5. Dynamic maintenance of the consistency using genetic algorithms

Genetic algorithms (GAs) perform multi-directional searches by maintaining potential solutions or scenarios (called also population of individuals) and encouraging information formation and exchange between these directions. It is an iterative procedure that maintains a constant size population of candidate solutions. Each iteration is called a generation and it undergoes some changes. *Crossover* and *mutation* are the two primary genetic operators that generate or exchange information in GAs. Under each generation, *good solutions* are expected to be produced and *bad solutions* die. It is the role of the objective (evaluation or fitness) function to distinguish the goodness of the solution. In the case of TCSPs, we define the following concepts.

Individual (potential solution): one possible assignment of numeric intervals to all events i.e set of couples (ev_i, occ_j), where ev_i is an event and occ_j is a possible interval belonging to the domain of ev_i. In other words, the individual represents a potential solution to the problem.

Population: a set of individuals (potential solutions).

Mutation: unary operator that returns a new individual (child) by assigning a new value (numeric intervals) to an event of a given individual (parent).

Crossover: n-ary operator that takes as arguments two individuals and returns a two new individuals with assignments belonging to parent individuals.

Fitness (evaluation) function: returns a measure of an individual. The measure corresponds here to the quality of the solution. The quality is defined by the number of satisfied constraints.

To illustrate the above concepts let us consider the following example.

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.*

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

Figure 6 shows an arc and path consistent graph corresponding to the above problem. A possible individual is presented here with the value of fitness equal to 3. Examples of mutation and crossover operators are respectively presented in Figs 7 and 8.

The pseudo code of the GA based method for dynamic TCSPs is illustrated in Fig. 9. Anytime a new constraint is added, this method starts from the population in which the solution of the TCSP (before adding the new constraint) has been found. The method will then iterate until the termination condition is satisfied. At each iteration, the method maintains a population of n individuals, $P(1) = \{ind_1^1, \ldots, ind_n^1\}$ for iteration 1, ... $P(t) = \{ind_1^t, \ldots, ind_n^t\}$ for iteration t, ..., etc. Each individual (potential solution) ind_i^t is evaluated using the fitness function. A new population at iteration $t+1$ is then formed by selecting the individuals with a better fitness value (*select* step in line 12) from the population of iteration t. Some of the selected individuals will be transformed (*alter* step in line 13) by the mutation and crossover operators. The algorithm is executed until it is running out of time or a solution with the best quality (quality = total number of satisfied constraints) is found.

The complexity in time of the GA method presented in Fig. 9 corresponds to the time cost needed at each iteration (since the number of iterations is constant). Each iteration involves the procedures *select*, *alter* and *evaluate* respectively in lines 12, 13 and 14. The procedure *select* consists of selecting a new population from the old one using the fitness value of the individuals.

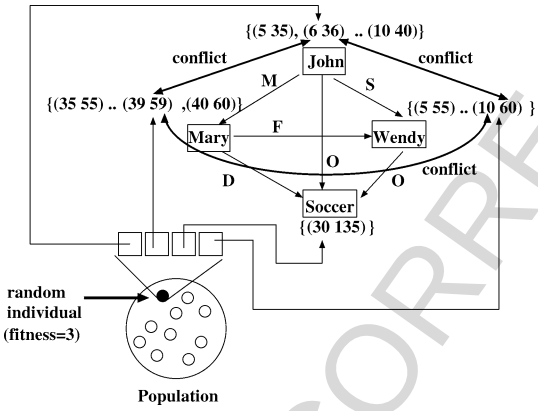

Fig. 6. GA representation of a TCSP.

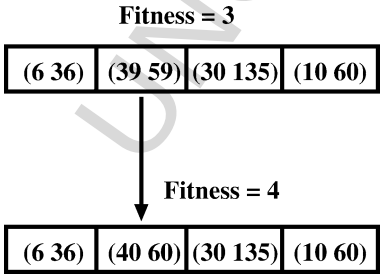

Fig. 7. Mutation operator.

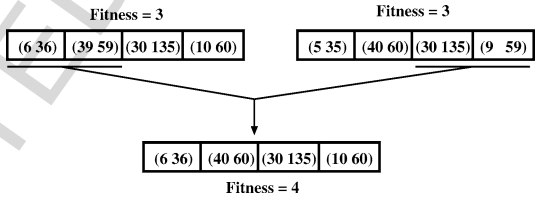

Fig. 8. Crossover operator.

```
1.  begin
2.    t ← 1
3.    while there is a new constraint to process do
4.      // P(t) denotes the population containing
        // the current solution
5.      if new constraint does not conflict with
        the solution in P(t) then
6.        return P(t)
7.      endif
8.      eval ← evaluate P(t)
9.      // the evaluation is performed when adding
        // the new constraint
10.     while termination condition is not satisfied do
11.       t ← t + 1
12.       select P(t) from P(t − 1)
13.       alter P(t)
14.       evaluate P(t)
15.     endwhile
16.     if solution found then
17.       return P(t)
18.   endwhile
19. end
```

Fig. 9. Genetic algorithm for dynamic constraints.

This requires a constant time cost (since the size of the population is a constant number). The procedure *alter* consists of applying a constant number of mutations and crossover on the individuals. The mutation operator changes the value of an individual event with a new one minimizing the fitness function. This new value is chosen in the same way as for the MCRW method and thus requires $O(ND)$ time cost where N is the number of variables and D their domain size. Similarly the crossover operator requires also $O(ND)$ time cost in order to select the good genes (events) from both individuals. The *evaluate* function costs $O(N^2)$ in the worst case (case of a complete graph where the number of constraints is equal to $[N(N-1)]/2$). The total cost of the GA procedure is thus $O(ND + N^2)$.

6. Experimentation

In order to evaluate and compare the performance of the three methods we propose, we have performed experimental tests on randomly generated consistent DTCSPs. The criteria used to evaluate the three different methods is the running time needed to maintain the consistency of the DTCSP and the percentage of success of each of the three methods. The experiments are performed on a SUN SPARC Ultra 5 station. All the procedures are coded in C/C++. A consistent TCSP of size N (N is the number of variables) has at least one complete numeric solution (set of N numeric intervals satisfying all the constraints of the problem). Thus, to generate a consistent TCSP we first randomly generate a numeric solution and then randomly 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 $x, y \in [0, \ldots, \mathit{Horizon}]$. This set of N pairs forms the initial solution where each pair corresponds to a time interval. *Horizon* is the time value before which all the events are processed.

Step 2. Generation of numeric constraints

For each interval (x, y) randomly pick an interval contained within [0..*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 hold between each interval pair of the initial solution. Add to each relation a random number belonging to the interval $[0, Nr]$ $(1 \leqslant Nr \leqslant 13)$ of chosen Allen primitives.

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

1. First a numeric solution is generated:
   S = {(14), (28), (57)}.
2. Numeric constraints (domains of the three events) are then randomly generated from the numeric solution.

Interval		SOPO		Domain
(1 4)	→	[0,9]	→	{(0 3) . . . (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)	→	O
(1 4) *and* (5 7)	→	B
(2 8) *and* (5 7)	→	Di

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

$O + BM$	→	BOM
$B + DDiEO$	→	$DiEOB$
$Di + DEFSP$	→	$FSDDiBE$

After generating the TCSP, the solving algorithm will process the list of temporal relations in an incremental way (in order to simulate a dynamic TCSP). More precisely, we start with a DTCSP having N variables and 0 constraints. Constraints are then added one by one, in an arbitrary order, from the randomly generated TCSP to the DTCSP until a given number of constraints, C, is reached. After adding each constraint, the solving algorithm will check the consistency of the new DTCSP.

Table 2 presents the results of tests performed on DTCSP instances defined by the number of variables N, the domain size D and the number of constraints C. For each method, the time parameter corresponds to the total running time in seconds needed to process all the constraints in an incremental manner. For each test, the three methods are executed on 100 instances and the average running time is taken. For each problem instance, the approximation methods are allocated a maximum time of 1000 seconds to find the solution. For large problems, these methods fail sometimes to

Table 2
Comparative tests on randomly generated DTCSPs

Problem			Dynamic CSPs	MCRW			GAs		
N	C	D	Time	Time	% of success	Unsolv cons	Time	% of success	Unsolv cons
20	95	50	0.10	0.12	100%	0	0.27	100%	0
40	390	50	0.27	0.24	100%	0	0.34	100%	0
60	885	50	0.65	0.82	100%	0	0.95	100%	0
80	1580	50	1.52	1.45	100%	0	1.32	100%	0
100	2475	50	1.89	1.57	98%	3	1.73	98%	3
200	9950	100	7.14	3.12	92%	17	3.16	94%	15
300	22425	100	43.46	7.18	88%	22	6.44	90%	18
400	39900	100	215.34	11.4	83%	35	10.36	87%	34

provide a solution satisfying all the constraints. This is expressed by the percentage of success which is the number of times the method succeeded to solve the problem. Note that, when an approximation method fails to solve a given instance, the corresponding running time is not considered to compute the average time. The column "Unsolv cons" contains the average number of non solved constraints reported in the case where the approximation method fails to return a complete solution.

The running times of the three methods are comparable for small and medium size problems ($N \leqslant 100$). This makes the exact method the technique of choice in this case since the incomplete methods fail sometimes (98% success for $N = 100$ in the case of both approximation methods) to solve the problem. For large problems the CSP-based method becomes very slow, comparing to the approximation methods. This is due to the exponential running time of the exact method (comparing to the polynomial time cost of the approximation methods). However, the percentage of success of the approximation methods decreases when the size of the problem increases. In this case, if the running time is not an issue, the exact method can be used. Indeed, the approximation methods fail to find a complete solution even when given 1000 seconds to solve the problem which exceeds the total running time required by the exact method to solve any problem tested. If, however, the running time is important, then the approximation methods are the methods of choice. This is, for example, the case of solving constraint problems in real time, where a solution (even incomplete) needs to be returned within a given deadline. Indeed, in the case of approximation methods, the running time is proportional to the quality of the solution returned (number of violated constraints) as we can see in Fig. 10. The figure illustrates here the execution times, in seconds, obtained by MCRW versus the corresponding quality

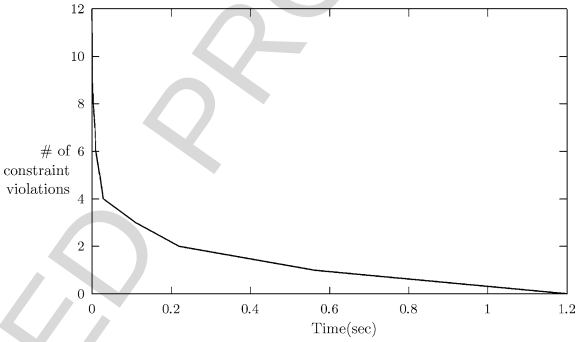

Fig. 10. Running time of MCRW versus quality of the returned solution for a 100-variable DTCSP.

of the solution returned. As we can see, on the figure, if we interrupt the program at anytime before 1.2 seconds (total time needed to solve the problem completely) we will obtain a solution with a given quality. For this particular problem instance, it took the exact method 1.54 seconds to solve the problem. Also, the exact method does not have the ability to return a solution with a quality proportional to the execution time.

The performance of the stochastic local search method and the genetic algorithms are comparable for both small size and large size problems. This suggests further study of the search space in order to see when to expect genetic algorithms to outperform stochastic local search and vise versa.

7. Conclusion and future work

In this paper we have presented three different methods for maintaining the consistency of a temporal constraint satisfaction problem in an incremental way. The methods are of interest for any application where qualitative and numeric temporal information should be managed in an evolutive environment. This can be

the case of real world applications such as reactive scheduling and planning where any new information corresponding to a constraint restriction should be handled in an efficient way.

One perspective of our work is to handle the relaxation of constraints during the resolution process. For example, suppose that during the search, a given constraint is removed. Would it be worthwhile to find those values removed previously because of this constraint and to put them back in the search space or would it be more costly than just continuing on with search.

Another perspective is to use the dynamic methods we propose for solving conditional TCSPs [15]. Conditional TCSPs are TCSPs containing temporal variables whose existence depends on the values chosen for other temporal variables. In this case we will have to maintain the consistency of the TCSP any time new temporal variables are added.

References

[1] 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.

[2] M. Mouhoub, Reasoning about numeric and symbolic time information, in: *Twelfth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'2000)*, Vancouver, IEEE Computer Society, 2000, pp. 164–172.

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

[4] R. Dechter, I. Meiri and J. Pearl, Temporal constraint networks, *Artificial Intelligence* **49** (1991), 61–95.

[5] I. Meiri, Combining qualitative and quantitative constraints in temporal reasoning, *Artificial Intelligence* **87** (1996), 343–385.

[6] J. Thornton, M. Beaumont, A. Sattar and M. Maher, A local search approach to modelling and solving interval algebra problems, *Journal of Logic and Computation* **14**(1) (2004), 93–112.

[7] P. Laborie, Une approche intégrée pour la gestion de ressources et la synthèse de plans, PhD thesis, École Nationale Supérieure des Télécommunications, 1995.

[8] M.L. Ginsberg, Dynamic backtracking, *Artificial Intelligence Research* (1993), 25–46.

[9] P. van Beek and D.W. Manchak, The design and experimental analysis of algorithms for temporal reasoning, *Journal of Artificial Intelligence Research* **4** (1996), 1–18.

[10] M. Mouhoub, Dynamic path consistency for interval-based temporal reasoning, in: *21st International Conference on Artificial Intelligence and Applications (AIA'2003)*, 2003, pp. 10–13.

[11] A. Gerevini, Incremental tractable reasoning about qualitative temporal constraints, in: *Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.

[12] Y. Zhang and R.H.C. Yap, Making ac-3 an optimal algorithm, in: *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA, 2001, pp. 316–321.

[13] C. Bessière and J.C. Régin, Refining the basic constraint propagation algorithm, in: *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA, 2001, pp. 309–315.

[14] M. Mouhoub and J. Yip, Dynamic CSPs for interval-based temporal reasoning, in: *Fifteenth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-2002)*, Cairns, Australia, 2002 (to appear).

[15] M. Mouhoub and A. Sukpan, Managing conditional and composite temporal constraints, in: *ECAI-04 Workshop on Spatial and Temporal Reasoning*, 2004 (to appear).