
Artificial Intelligence Review **21:** 25–56, 2004.

Reasoning with Numeric and Symbolic Time Information

MALEK MOUHOUB
Department of Computer Science, University of Regina, 3737 Wascana Parkway, Regina, Saskatchewan, Canada S4S 0A2
(E-mail: mouhoubm@cs.uregina.ca)

Abstract. Representing and reasoning about time is fundamental in many applications of Artificial Intelligence as well as of other disciplines in computer science, such as scheduling, planning, computational linguistics, database design and molecular biology. The development of a domain-independent temporal reasoning system is then practically important. An important issue when designing such systems is the efficient handling of qualitative and metric time information. We have developed a temporal model, TemPro, based on the Allen interval algebra, to express and manage such information in terms of qualitative and quantitative temporal constraints. TemPro translates an application involving temporal information into a Constraint Satisfaction Problem (CSP). Constraint satisfaction techniques are then used to manage the different time information by solving the CSP. In order for the system to deal with real time applications or those applications where it is impossible or impractical to solve these problems completely, we have studied different methods capable of trading search time for solution quality when solving the temporal CSP. These methods are exact and approximation algorithms based respectively on constraint satisfaction techniques and local search. Experimental tests were performed on randomly generated temporal constraint problems as well as on scheduling problems in order to compare and evaluate the performance of the different methods we propose. The results demonstrate the efficiency of the MCRW approximation method to deal with under constrained and middle constrained problems while Tabu Search and SDRW are the methods of choice for over constrained problems.

Keywords: approximation algorithms, constraint satisfaction techniques, planning, scheduling, temporal reasoning

1. Introduction

Time representation and reasoning is fundamental in many applications of artificial intelligence as well as of other disciplines of computer science, such as computational linguistics (Song and Cohen 1991; Hwang and Shubert 1994), database design (Orgun 1996), computational models for molecular biology (Golumbic and Shamir 1993), scheduling (Pape and Smith 1987; Baptiste and Pape 1995) and planning (Laborie and Ghallab 1995b). The development of a domain-independent temporal reasoning system is thus practically important. An important issue when designing such systems is the efficient handling of qualitative and metric time information. Indeed, the

separation between the two aspects does not exist in the real world. In our daily life activities, for example, we combine the two type of information to describe different situations. In the case of scheduling problems, we can have qualitative information such as the ordering between tasks and quantitative information describing the temporal windows of the tasks i.e., earliest start time, latest end time and the duration of each task.

In a previous work (Mouhoub et al. 1998), we have developed a temporal model, TemPro, based on the interval algebra, to express such information in terms of qualitative and quantitative temporal constraints. TemPro translates an application involving temporal information into a binary Constraint Satisfaction Problem (CSP).[1] We call it a Temporal Constraint Satisfaction Problem (TCSP).[2] Managing temporal information consists then of solving the TCSP using a resolution method based on local consistency and backtrack search. Local consistency is enforced at both the symbolic and the numeric levels using respectively path and arc consistency algorithms. Local consistency is also used during the backtrack search to reduce the number of consistency checks by detecting early later failures. In this paper we present an updated version of the resolution method including an additional preprocessing phase called numeric $\rightarrow$ symbolic conversion used to reduce the size of the search space. We have also improved the arc consistency, path consistency and backtrack search techniques by reducing the number of consistency checks necessary by each algorithm.

In order to deal with real time applications where a solution should be provided within a given deadline or those applications where it is impossible or impractical to solve these problems completely, we have studied different methods offering a trade off between search time and solution quality. These methods are exact and approximation algorithms based respectively on branch and bound techniques and local search. The idea behind the exact methods is to seek to partially solve the problem by satisfying a maximal number of constraints. We call this latter a maximal temporal constraint satisfaction problem (MTCSP). Local consistency and backtrack search methods we use to solve the TCSP can be adapted to cope with, and take advantage of, the differences between partial and complete constraint satisfaction. The method is based on branch and bound techniques and has the advantage to provide a solution that is guaranteed to be optimal. However, as we will see in the following, this method is impractical for large size problems and is in general useful to verify the optimality and, therefore, the quality of the solution returned by approximation methods. Approximation methods, on the other hand, do not guarantee the optimality of the solution provided but are obviously of interest when they provide near optimal solutions.

In the following section, we will present through an example, the different components of the model TemPro. The description of the different methods we propose for solving TCSPs and MTCSPs are then presented respectively in sections 3 and 4. Experiments evaluating and comparing the methods we describe in this paper are reported in section 5. Concluding remarks and possible perspectives of our work are presented in section 6.

2. Background and Basic Concepts

Consider the following typical temporal reasoning problem:

> *John, Mike and Lisa work for a company in Calgary. It takes John* **20 minutes***, Mike* **25 minutes** *and Lisa* **30 minutes** *to get to work. Every day, John left home* **between 7:20 and 7:26***. Mike arrives at work* **between 7:55 and 8** *and Lisa arrives at work* **between 7:50 and 8***. We also know that John and Mike* **meet** *on their way to work, that Mike arrives to work* **before** *Lisa and that Lisa and John go to work* **at the same time**.

The above story includes numeric and qualitative information (words in boldface). There are three main events: John, Mike and Lisa are going to work respectively. Some numeric constraints specify the duration of the different events, e.g., *15 minutes is the duration of Mike's event*. Other numeric constraints describe the temporal windows in which the different events occur. And finally, symbolic constraints state the relative positions between events e.g., *John and Mike meet on their way to work*.

Given these kind of information, our main goal 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 Lisa arrived at work?", . . . , etc.

To reach this goal, we first translate the temporal problem into a constraint satisfaction problem. Our model TemPro is used for this purpose. Resolution techniques are then used to check for the consistency of the problem and to look for possible solutions. In the following, we will present the different objects of our model TemPro.

2.1. *Time-line*

Since our model TemPro translates the different numeric and symbolic temporal information into a constraint satisfaction problem, we use a discrete model of time. We define a temporal reference *Tr* (see Figure 1) as the maximal set of discrete and adjacent temporal units u_i. Each unit represents

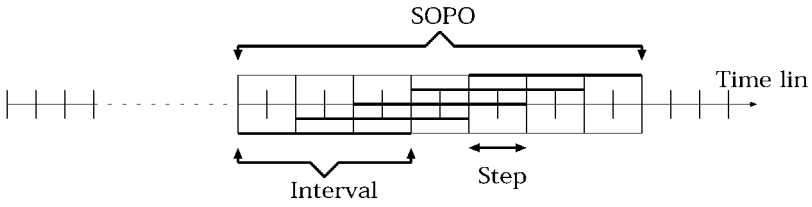

Figure 1. The SOPO of a given event.

the smallest discrete portion of time that can be obtained over the temporal reference.

2.2. *Interval*

An interval I is a finite set of adjacent units u_k represented by a couple of units (u_i, u_j), where u_i and u_j are respectively the begin and end times of I.

2.3. *Events*

In TemPro, temporal objects are called events. Events have a uniform reified representation made up of a proposition and its temporal qualification: *Evt* = *OCCUR*(*p, I*) defined by Allen (Allen 1983) and denoting the fact that the proposition p occurred over the interval I. For the sake of notation simplicity, an event is used in this paper to denote its temporal qualification. According to our model, in our example we have three main events: J, L and M representing the fact that John, Lisa and Mike are going to work respectively.

2.4. *Qualitative constraints*

Qualitative constraints specify the relative temporal position of an event with respect to other events. The qualitative constraint between two events ev_1 and ev_2 can take the following forms: $ev_1\ r_1\ r_2\ \ldots\ r_n\ ev_2$ where each of the r_i's is one of the thirteen Allen primitives (Allen 1983): *precedes, during, overlaps, meets, starts, finishes* noted respectively P, D, O, M, S and F; their converses $P^{\smile}$, $D^{\smile}$, $O^{\smile}$, $M^{\smile}$, $S^{\smile}$ and $F^{\smile}$; and the equality relation E. Table 1 presents the definition of the thirteen Allen primitives.

2.5. *Quantitative constraints*

Additional information about an event can be expressed, thus restricting its temporal qualification to belong to a given SOPO, i.e., the Set Of Possible Occurrences where the given event can take place. In our case, a SOPO is restricted to be a finite set of intervals with constant duration. We represent a

Table 1. Allen primitives.

Relation	Symbol	Inverse	Meaning
X precedes Y	P	$P^{\smile}$	XXX YYY
X equals Y	E	E	XXX YYY
X meets Y	M	$M^{\smile}$	XXXYYY
X overlaps Y	O	$O^{\smile}$	XXXX YYYY
X during y	D	$D^{\smile}$	XXX YYYYYY
X starts Y	S	$S^{\smile}$	XXX YYYYY
X finishes Y	F	$F^{\smile}$	XXX YYYYY

SOPO by the fourfold [*begintime, endtime, duration, step*] where *begintime* and *endtime* are respectively the earliest start time and the latest end time of the corresponding event, *duration* is the duration of the event and *step* defines the distance between the starting time of two adjacent intervals within the SOPO. Thus, if e_i is an event numerically constrained by the SOPO [inf_i, sup_i, d_i, s_i], then the set of possible occurrences of e_i is defined as follows:

$$I = \{occ_j \mid begin(occ_j) = inf_i + k * s_i, end(occ_j) = begin(occ_j) + d_i,$$
$$end(occ_j) <= sup_i, k \in [0, \frac{sup_i - inf_i - d_i}{s_i}] \cap \aleph\}.$$

begin and *end* are functions on intervals that return the begin and the end points of a given interval, respectively. Figure 1 illustrates the SOPO of a given event.

2.6. *TemPro based temporal constraint satisfaction*

Using the concepts described before, TemPro transforms a temporal problem involving numeric and symbolic information into a temporal constraint satisfaction problem including:

- a set of variables $\{EV_1, \ldots, EV_n\}$, each defined on a discrete domain D_i standing for the set of possible occurrences (SOPO) in which the corresponding event can hold,
- and a set of binary constraints, each representing a qualitative disjunctive relation between a pair of events and thus restricting the values that the events can simultaneously take.

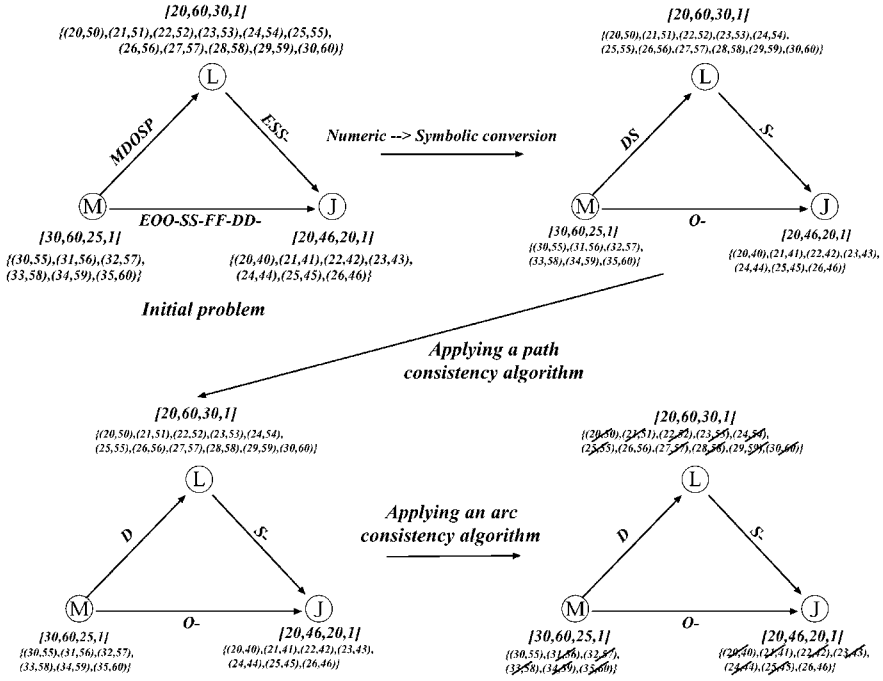

Figure 2. Numeric → symbolic conversion + local consistency phases.

Using TemPro, our example can be represented by the following temporal constraint network (see initial problem of Figure 2).

A **solution to a TCSP** is an assignment of a value (numeric interval) from its domain to every variable, in such a way that every constraint is satisfied. We may want to find:

- just one solution, with no preference as to which one,
- all solutions,
- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.

In the real world, a main challenge for temporal constraint problems is the ability to run on-line, that is to give a solution quickly enough so that the system can use it without missing the deadline of the first events. The provided solution will have a quality (number of satisfied constraints) depending on the time allocated for computation. We are dealing here with a Maximal Temporal Constraint Satisfaction Problem (MTCSP) which is an optimization problem consisting of looking for an assignment that satisfy the maximal number of temporal constraints. Methods for solving an MTCSP include exact algorithms based on branch-and-bound techniques and approximation methods based on local search.

3. Solving TCSPs

Given a TemPro constraint-based network, one important task of reasoning is to determine the consistency of the network, and therefore to find a solution of this network if the latter is consistent. Since we are dealing with a constraint satisfaction problem, deciding consistency is in general NP-hard.[3] Indeed, looking for a complete solution requires a backtracking algorithm of complexity $O((Max(\frac{sup_i - inf_i - d_i}{s_i}))^N)$ where inf_i, sup_i, d_i and s_i are respectively the earliest start time, latest end time, duration and step of a given event e_i and N is the number of events. In order to overcome this difficulty in practice, we propose a resolution method based on local consistency techniques. Indeed, these algorithms transform 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. A k-consistency algorithm removes all inconsistencies involving all subsets of k variables belonging to N. For $k = 2$ and $k = 3$, the solutions are called arc and path consistent respectively. The k-consistency problem 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. In (Mouhoub et al. 1998) we have proposed a method based on local consistency and backtrack search to look for a possible solution of a CSP involving numeric and symbolic temporal constraints. In this paper we present an improved version of the method. In the updated version we have added a new pre-processing phase that we call numeric $\rightarrow$ symbolic conversion. This phase reduces the search space by removing some symbolic relations that are inconsistent with the numeric constraints. We have also modified the way arc consistency, path consistency and backtrack search work. More precisely the new resolution method we propose is described by the following steps:

3.1. *Numeric $\rightarrow$ symbolic conversion*

From the numeric information of two given events, we can extract the symbolic relation they share. The intersection of this relation with the current qualitative relation between the two events will reduce the size of the latter which simplifies the size of the original problem. The naive algorithm that converts the information from numeric to symbolic requires $O(c(Max_{1 \leq i \leq n}(\frac{sup_i - inf_i - d_i}{s_i}))^2)$ in time where c is the number of qualitative constraints, n is the number of events, inf_i, sup_i, d_i and s_i are respectively the earliest start time, latest end time, duration and step of the events. We have defined a method that extracts most of the primitives within a relation between

each pair of events in constant time, thus reducing the complexity to $O(c)$. The method consists of using the information concerning the lower bound, upper bound and duration of the event SOPO instead of its occurrences. The pseudo-code of the method is presented in Figure 3.

3.2. *Local consistency processing*

In this phase arc and path consistency algorithms are performed respectively at the numeric and symbolic levels in order to reduce the size of the search space. Path consistency is first performed on disjunctive relations to reduce their size by removing some inconsistent Allen primitives (those which do not respect the path consistency). This will decrease considerably the number of tests performed later by the arc consistency and the backtrack search algorithms. More precisely, the path consistency algorithm (called also transitive closure algorithm) works as follows:

> *Choose any three nodes I, J and K of the temporal network and checks whether* $R_{IK} = R_{IK} \cap (R_{IJ} \otimes R_{JK})$. *If* R_{IK} *is updated then this update should be propagated to the rest of the network. The algorithm iterates until no more such changes are possible.*

R_{IK} (respectively R_{IJ} and R_{JK}) is the binary relation between node I and node K (respectively between nodes I and J; and between nodes J and K). $\cap$ is the intersection operator between two relations (the result of the intersection between two relations is the common Allen primitives the two relations share). $\otimes$ is the composition operator between two relations. Allen (Allen 1983) has defined a 13×13 composition table between Allen primitives (see Table 2). The path consistency algorithm assumes that the constraint graph is complete. If the initial graph is not complete then it is transformed to a complete one by adding arcs labeled with the universal relation I which corresponds to the disjunction of the thirteen Allen primitives.

Our implementation of the path consistency algorithm (see Figure 4) differs from the above transitive closure algorithm. Indeed, we have used the following improvements:

- Since only the triangle of edges whose labels have changed in the previous iteration need to be computed, we use, as reported by Mackworth (Mackworth 1977), Allen (Allen 1983) and Van Beek (van Beek and Manchak 1996), a queue data structure for maintaining the triangles that must be recomputed. The computation proceeds until the queue is empty. This will considerably reduce the number of triangles to be processed.

```
Function NumSymb(list)
// list: contains all the constraints of the graph
// sopo: array containing the SOPOs of the events
// each entry of the array has 4 fields
//       inf: earliest start time
//       sup: latest end time
//       duration: duration of the event
//       step: discretization step
 1. relation ← E P P⌣ S S⌣ F F⌣ D D⌣ O O⌣ M M⌣
    // set relation to the universal relation I (disjunction of the 13 primitives)
 2. while (not_empty(list)) do
 3.    elt ← get_elt(list)
 4.    i ← elt.i, j ← elt.j
 5.    if (sopo[i].inf > sopo[j].sup) then
 6.       return relation ← P⌣
 7.    else
 8.       if (sopo[i].sup < sopo[j].inf) then
 9.          return relation ← P
 10.      if ( (sopo[i].duration ≠ sopo[j].duration)
          OR ((sopo[j].sup - sopo[i].inf) < sopo[i].duration)
          OR ((sopo[i].sup - sopo[j].inf) < sopo[i].duration) )
          {
 11.         relation ← relation − E
 12.         if (sopo[i].duration > sopo[j].duration) then
 13.            relation ← relation − {S, F, D}
 14.         else
 15.            if (sopo[i].duration < sopo[j].duration)
 16.               relation ← relation − {S⌣, D⌣, F⌣}
          }
 17.      else
 18.         if (sopo[i].duration = sopo[j].duration) then
 19.            relation ← relation − {D, D⌣, S, S⌣, F, F⌣}
 20.         if ((sopo[i].inf + sopo[i].duration) > (sopo[j].sup - sopo[j].duration)) then
 21.            relation ← relation − {M, P}
 22.         if ((sopo[i].sup - sopo[i].duration) < (sopo[j].inf + sopo[j].duration))
 23.            relation ← relation − {M⌣, P⌣}
 24.         if (sopo[i].inf > (sopo[j].sup - sopo[j].duration)) then
 25.            relation ← relation − {S, S⌣, O, D⌣}
 26.         if ((sopo[i].inf + sopo[i].duration) > sopo[j].sup) then
 27.            relation ← relation − {F, f, D}
 28.         if (sopo[i].sup < (sopo[j].inf + sopo[j].duration)) then
 29.            relation ← relation − {F, f}
 30.         if ((sopo[i].sup - sopo[i].duration) < (sopo[j].inf + sopo[j].duration)) then
 31.            relation ← relation − {O⌣}
 32.         if (sopo[i].inf < sopo[j].inf ) then
 33.            relation ← relation − {D}
 34. return relation
```

Figure 3. Numeric to symbolic conversion algorithm.

Table 2. Allen's composition table.

	E	P	$P^{\smile}$	D	$D^{\smile}$	O	$O^{\smile}$	M	$M^{\smile}$	S	$S^{\smile}$	F	$F^{\smile}$
E	E	P	$P^{\smile}$	D	$D^{\smile}$	O	$O^{\smile}$	M	m	S	s	F	$F^{\smile}$
P	P	P	I	u	P	P	u	P	u	P	P	u	P
$P^{\smile}$	p	I	$P^{\smile}$	$v^{\smile}$	$P^{\smile}$	$v^{\smile}$	$P^{\smile}$	$v^{\smile}$	$P^{\smile}$	$v^{\smile}$	$P^{\smile}$	$P^{\smile}$	$P^{\smile}$
D	D	P	$P^{\smile}$	D	I	u	$v^{\smile}$	P	$P^{\smile}$	D	$v^{\smile}$	D	u
$D^{\smile}$	$D^{\smile}$	v	$u^{\smile}$	n	$D^{\smile}$	$z^{\smile}$	$y^{\smile}$	$z^{\smile}$	$y^{\smile}$	$z^{\smile}$	$D^{\smile}$	$y^{\smile}$	$D^{\smile}$
O	O	P	$u^{\smile}$	y	v	x	n	P	$y^{\smile}$	O	$z^{\smile}$	y	x
$O^{\smile}$	$O^{\smile}$	v	$P^{\smile}$	z	$u^{\smile}$	n	$x^{\smile}$	$z^{\smile}$	$P^{\smile}$	z	$x^{\smile}$	$O^{\smile}$	$y^{\smile}$
M	M	P	$u^{\smile}$	y	P	P	y	P	a	M	M	y	P
$M^{\smile}$	$M^{\smile}$	v	$P^{\smile}$	z	$P^{\smile}$	z	$P^{\smile}$	b	$P^{\smile}$	z	$P^{\smile}$	$M^{\smile}$	$M^{\smile}$
S	S	P	$P^{\smile}$	D	v	x	z	P	$M^{\smile}$	S	b	D	x
$S^{\smile}$	s	v	$P^{\smile}$	z	$D^{\smile}$	$z^{\smile}$	$O^{\smile}$	$z^{\smile}$	m	b	s	$O^{\smile}$	$D^{\smile}$
F	F	P	$P^{\smile}$	D	$u^{\smile}$	y	$x^{\smile}$	M	$P^{\smile}$	D	$x^{\smile}$	F	a
$F^{\smile}$	$F^{\smile}$	P	$u^{\smile}$	y	$D^{\smile}$	O	$y^{\smile}$	M	$y^{\smile}$	O	$D^{\smile}$	a	$F^{\smile}$

$x = P \vee O \vee M$
$y = D \vee O \vee S$
$z = D \vee O^{\smile} \vee F$
$a = E \vee F \vee F^{\smile}$
$b = E \vee S \vee S^{\smile}$
$u = P \vee O \vee M \vee D \vee S$
$v = P \vee O \vee M \vee D^{\smile} \vee F^{\smile}$
$n = E \vee F \vee D \vee O \vee S \vee F^{\smile} \vee D^{\smile} \vee O^{\smile} \vee S^{\smile}$

– We changed as reported in (Bessière et al. 1996) the way composition and intersection of relations are achieved during the path consistency process (following the principle "one support is sufficient").

After the path consistency phase, arc consistency is applied on each pair of variables sharing a qualitative constraint to reduce the size of variable domains by removing some inconsistent values (those which do not satisfy the arc consistency). AC3.1 (Zhang and Yap 2001; Bessière and Régin 2001) is the algorithm we have chosen to achieve the arc consistency. This is justified by the interesting space and time complexity this algorithm offers in addition to the simplicity of its implementation. Indeed AC-3.1 has the best compromise between time and memory costs comparing to the other arc consistency algorithms proposed in the literature. Our implementation of AC-3.1 for TCSPs is presented in Figure 5.

Function PC()
// $INVERSE$: returns the inverse of a disjunctive relation
// Exp: $INVERSE(PF^{\smile}OD^{\smile}) = P^{\smile}FO^{\smile}D$
1. $PC \leftarrow false$
2. $L \leftarrow \{(i, j) | 1 \leq i < j \leq n\}$
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. $t \leftarrow C_{xk} \bigcap C_{xy} \otimes C_{yk}$
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. $t \leftarrow C_{ky} \cup C_{kx} \otimes C_{xy}$
12. **if** $(t \neq C_{ky})$ **then**
13. $C_{yk} \leftarrow INVERSE(t)$
14. $L \leftarrow L \cup \{(k, y)\}$

Figure 4. Path consistency algorithm.

Function AC3.1()
// sopo: is an array of SOPOs
// R: set of disjunctive relations of the TCSP
1. $Q \leftarrow \{(i, j) \mid (i, j) \in R\}$
2. $AC \leftarrow true$
3. **While** $Q \neq Nil$ **Do**
4. $Q \leftarrow Q - \{(x, y)\}$
5. **If** $REVISE(x, y)$ **then**
6. **if** $Dom(x) \neq \emptyset$ **then**
7. $Q \leftarrow Q \sqcup \{(k, x) \mid (k, x) \in R \wedge k \neq y\}$
8. **else**
9. return $AC \leftarrow false$

Function REVISE(x, y)
// $compatible$: checks if two intervals are compatible
// regarding the symbolic relation they share
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. $REVISE \leftarrow true$

Figure 5. Arc consistency algorithm.

3.3. *Backtrack search*

After the local consistency phase is achieved, the backtrack search algorithm is performed to look for a possible numeric solution. Arc consistency is also used during this phase following the principle of the forward check strategy (Haralick and Elliott 1980) in order to allow branches of the search tree that will lead to failure to be pruned earlier than with simple backtrack. More precisely the backtrack search algorithm works as follows:

> *Choose a node and instantiate the corresponding variable (that we call current variable) to a value (numeric interval) belonging to its domain. Discard from the variable domain the remaining values and run the arc consistency algorithm between the current variable and the non instantiated variables (called future variables). If the network is arc consistent, fix a value on another variable and run again the arc consistency algorithm until each value is fixed on the domain of every variable of the network. We obtain a solution corresponding to the set of numeric intervals fixed on the domain of each variable. If the network does not succeed (is not arc consistent) at some point, choose another value of the domain of the last selected variable. If there is no value to be considered. Backtrack and choose another value from the domain of the previous variable.*

During the backtrack search, we use the following properties to select the different variables and values:

- Choose the most constrained variable to assign next. This can be determined by the number and type of the different relations connected to each variable. For example, the primitive relation E is more constraining than S, $S^{\smile}$, F and $F^{\smile}$ which are more constraining than O, $O^{\smile}$, D, $D^{\smile}$, P and $P^{\smile}$.
- Choose the least constraining value for each variable.

Figure 2 shows the application of the numeric $\rightarrow$ symbolic conversion and local consistency phases to the temporal reasoning problem described in the previous section. Note that for this particular example a numeric solution is obtained without the need of a backtrack search phase (which is not the case in general).

4. Solving MTCSPs

In this section we present 2 different ways for solving an MTCSP. The first one is an exact algorithm based on the branch and bound technique. The solution provided by this method is guaranteed to be optimal. The second

way concerns approximation algorithms based on local search. The exact algorithm is the procedure of choice if the problem is small enough that all or most can be solved quickly through this approach. And as we will see in section 5, this method can also be used to evaluate the goodness of the results returned by the approximation methods. Approximation algorithms are in general used for large size problems (Minton et al. 1992; Selman and Kautz 1993) and, obviously, are of interest when they provide near optimal solutions with a polynomial computational complexity.

4.1. *Branch and bound method*

The method we present here uses partial constraint satisfaction techniques (Freuder and Wallace 1992; Wallace 1995), capable of solving temporal constraint problems by giving a solution with a quality depending on the time allocated for computation. It consists of using a branch and bound variant of the backtrack search algorithm in order to satisfy the maximum number of constraints. More precisely, we have transformed the propagation techniques we have seen in the previous section, as follows:

1. In the pre-processing phase of the resolution method, instead of performing arc consistency algorithms to remove some values that do not belong to any solution, we use direct arc consistency algorithms (Dechter and Peal 1988; Wallace 1995) to count the number of inconsistencies counts associated with each value (number of domains that offer no support for that value). Note that, with direct arc consistency algorithms, checking is unidirectional so inconsistency counts are non-redundant. In order to perform direct consistency checking, a variable ordering should be first established. The ordering heuristic we use consists of sorting variables by decreasing number of constraints shared with those already instantiated. The DAC-3 algorithm (see Figure 6) is a modification of the algorithm AC-3 to perform direct consistency checking in the case of temporal constraints. Note that the checking is in backward direction, i.e., each value is tested for support in domains of past variables. This means that the inconsistencies counts associated to each variable value correspond to the number of domains of past variables having no support for the value.
2. In the search phase, a cost function corresponding to the number of violated constraints is associated with each path of the search tree. The branch and bound algorithm starts by setting a lower bound value on the cost function. Search is then performed down on each path until all variables are instantiated and in this case a new lower bound is found, or when a partial assignment of values to variables is at least as great as the actual lowest cost. More precisely, the branch and bound algorithm

Begin
Set the counters of variable values to 0.
For each event ev_i
For each interval a in the domain of ev_i
For each event ev_j before ev_i such that
ev_i and ev_j share a disjunctive relation r_{ij}
If there is no interval b in the domain of ev_j
such that $a\ r_{ij}\ b$
increment the counter of a
End

Figure 6. Algorithm DAC-3.

works in the same fashion as the backtrack search algorithm (used to look for a complete solution) except that when the current variable is instantiated to a chosen value (interval), it will not perform a local consistency algorithm to check if there is an inconsistency (because of the chosen value) but computes the estimated cost (minimum quality) of the solution obtained with the chosen value. This estimation of the minimum quality is computed using the inconsistency counts associated to each variable value. If the estimation is greater than the lower bound we have found so far, another value is chosen.

The pseudo-code of the branch and bound algorithm is presented in Figure 7.

4.2. *Approximation methods*

We will use the following terms:

State: one possible assignment of 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; the number of states is equal to the product of domains sizes.

State or solution quality: the number of constraint violations of the state or the solution.

Neighbor: the state which is obtained from the current state by changing one event value.

Local-minimum: the state that is not a solution and the evaluation values of all of its neighbors are larger than or equal to the evaluation value of this state.

Strict local-minimum: the state that is not a solution and the quality of all of its neighbors are larger than the evaluation value of this state.

The different algorithms that we will consider in the following are based on a common idea known under the notion of local search. In local search,

```
Begin
  // count_ix: inconsistency count for value x of event i
  cost_function = 0, lower_bound = MAX_VALUE
  while lower_bound > 0 do
    if all values have been tried for first event chosen then
      exit
    else
      choose future event ev_j as the current variable
        for each interval intv_x in the domain of ev_j do
          if cost_function + count_jx +
            Σ_k is a future event ≠j min(count_ky) < lower_bound
            if all events are now instantiated then
              lower_bound = cost_function,
              save solution
            else if consistency_check(ev_j, intv_x,
            cost_function, Σ_k is a future event ≠j min(count_ky)) then
              add intv_x to partial solution
              set distance to value returned by consistency_check
              exit and go to while loop
        // all intervals have been tested for the chosen event
        return ev_j to the list of future events and backtrack to previous event
End

Function consistency_check(current_event,
                           current_interval,
                           cost_function,
                           sum_min_count)
  add_to_cost = 0
  for each past event evt_i that shares a relation with current_event
    if interval assigned to evt_i is inconsistent with current_interval then
      increment add_to_cost
      if (cost_function + add_to_cost + sum_min_count) ≥ lower_bound then
        return false
  return (cost_function + add_to_cost)
```

Figure 7. Pseudo-code of the branch and bound algorithm.

an initial configuration (assignment of events) is generated randomly and the algorithm moves from the current configuration to a neighborhood configurations until a complete solution (TCSP problems) or a good one (MTCSP problems) has been found or the resources available are exhausted.

4.2.1. *Min-Conflict-Random-Walk method (MCRW)*

After an initial configuration is randomly generated, the Min-conflicts method chooses randomly any conflicting event, i.e., the event that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks

```
procedure MCRW(Max_Moves,p)
  s <- random valuation of events;
  nb_moves <- 0;
  while eval(s) > 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 s
end MCRW
```

Figure 8. Pseudo-code of the MCRW method.

randomly one value that does not increase the number of violated constraints (the current value of the event is picked only if all the other values increase the number of violated constraints). The problem of this method is that it is not able to leave local-minimum. In addition, if the algorithm achieves a strict local-minimum it does not perform any move at all and, consequently, it does not terminate. Thus, noise strategies should be introduced. Among them, the random-walk strategy that works as follows: for a given conflicting event, the random-walk 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. This time is of order $O(NMax_{1\leq i\leq N}(\frac{sup_i - inf_i - d_i}{s_i}))$ where N is the number of variables and sup_i, inf_i, s_i and d_i are respectively the latest end time, earliest start time, duration and step of the different events. Figure 8 presents the pseudo-code of the MCRW method for solving TCSPs.

4.2.2. *Steepest-Descent-Random-Walk (SDRW)*

In the Steepest-Descent method, instead of selecting the event in conflict randomly, this algorithm explores the whole neighborhood of the current configuration and selects the best neighbor (neighbor with the best quality).

```
procedure SDRW(Max_Moves,p)
  s <- random valuation of variables;
  nb_moves <- 0;
  while eval(s) > 0 & nb_moves < Max_Moves do
    if probability p verified then
      choose randomly a variable evt in conflict;
      choose randomly a value intv for evt;
    else
      choose a move <evt,intv> with the best performance
    endif
    if intv # current value of evt then
      assign intv to evt;
      nb_moves <- nb_moves+1;
    endif
  endwhile
  return s
end SDRW
```

Figure 9. Pseudo-code of the SDRW method.

This algorithm can be randomized by using the random-walk strategy in the same manner as for Min-Conflicts to avoid getting stuck at "local optima". The time cost required in each iteration corresponds to the time needed to find the best neighbor and is of order $O(N^2 Max_{1\leq i\leq N}(\frac{sup_i - inf_i - d_i}{s_i}))$ in the worst case. The pseudo-code of the SDRW method is presented in Figure 9.

4.2.3. *Tabu Search (TS)*

The pseudo-code of Tabu search method is illustrated in Figure 10. This method is based on the notion of Tabu list used to maintain a selective history, composed of previously encountered configurations in order to prevent Tabu from being trapped in short term cycling and allows the search process to go beyond local optima. In each iteration of the algorithm, a couple $<$ *event, intv* $>$ that does not belong to the Tabu list and corresponding to the best performance is selected and considered as an assignment of the current configuration. $<$ *event, intv* $>$ will then replace the oldest move in the Tabu list. The time cost required in each iteration is the same as for SDRW, i.e., $O(N^2 Max_{1\leq i\leq N}(\frac{sup_i - inf_i - d_i}{s_i}))$ in the worst case.

Figure 11 illustrates the application of MCRW to the example presented in section 2. The algorithm starts by randomly generating an initial configuration (potential solution). One of the events in conflict is then chosen randomly (L corresponding to Lisa's event) and a value (interval) minimizing the number of conflicts is assigned to the chosen event. The algorithm is now

```
procedure Tabu-Search(Max_Iter)
  s <- random valuation of variables;
  nb_iter <- 0;
  initialize randomly the tabu list of size tl_size;
  while eval(s)>0 & nb_iter<Max_Iter do
   choose a move <evt,intv> with the best performance
   among the non-tabu  moves;
    remove the oldest move from the tabu list;
    introduce <evt,intv> in the tabu list,
              where intv is the current values of evt;
    assign intv to evt;
    nb_iter <- nb_iter+1;
  endwhile
  return s
end tabu-search
```

Figure 10. Pseudo-code of the tabu search method.

trapped in a local optima. Indeed, there is only one conflicting constraint and none of the alternate values for the three events can improve this situation. As we stated before, the only way to escape the local optima is to choose an interval randomly (even if it increases the number of conflicting variables) for a selected event. In our case, L is selected and an interval is chosen randomly for this event. This leads to a situation with two conflicting constraints. This is worse than the previous state but allows us to escape the local optima. Events and values are then selected until a complete solution (with no conflicting constraints) is obtained.

5. Experimentation

In this section, we present comparative tests concerning the different exact and approximation methods we have seen in the previous sections. Subsection 5.4 is dedicated to tests performed on general consistent and inconsistent TCSPs randomly generated while subsection 5.5 concerns job|machine scheduling problems.

5.1. *Comparison criteria*

We use two criteria to compare the different exact and approximation methods. The first one is the quality of the solution, i.e., the minimum number of violated constraints of the solution provided by the method. The second

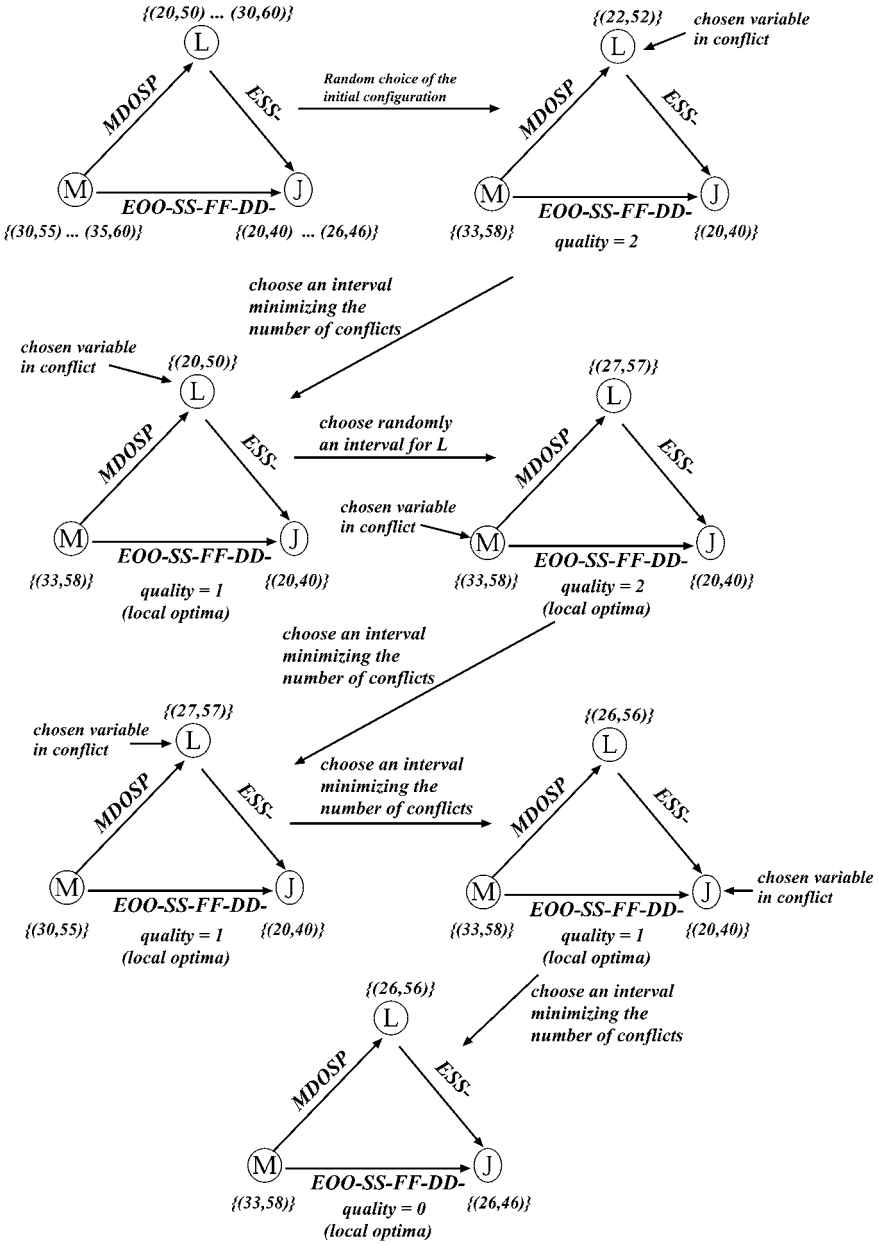

Figure 11. Solving a TCSP using MCRW.

criterion is the computing effort needed by an algorithm to find its best solution. This last criterion is measured by the number of moves and the running time in seconds required by each algorithm. The experiments are performed on a SUN SPARC Ultra 5 station. All the procedures are coded in C|C++.

5.2. *Parameter tuning for MCRW, SDRW and Tabu search*

The performance of the approximation algorithms we use is greatly influenced by the following parameters: the size of the Tabu list, *tl_size*, in the case of Tabu search; and the random walk probability, p, in the case of MCRW and SDRW. Preliminary tests determined the following ranges of parameter values:

- $10 \leq tl\_size \leq 20$
- $0.05 \leq p \leq 0.20$

Different discrete values between these ranges were further tested and the best values were identified for each approximation method as follows.

- For MCRW, $p = 5$.
- For SDRW, $p = 5$.
- For Tabu Search, $tl\_size = 10$.

5.3. *TCSP instances*

The tests presented in this subsection are performed on consistent and inconsistent problems generated as follows.

5.3.1. *Generation of consistent TCSPs*

Consistent TCSPs are those containing at least one numeric solution. Thus, to generate a consistent TCSP we first start by generating a numeric solution (a set of numeric intervals) and then we randomly add other numeric and symbolic information to it. More precisely, the random generation process is as follows.

1. **Generation of the numeric solution:** *Randomly pick n (n is the number of variables of the problem to generate) pairs (x, y) of integers such that $x < y$ and $x, y \in [0, \ldots, Horizon]$ (Horizon is the parameter before which all events must be processed). This set of n pairs forms the initial solution where each pair corresponds to a time interval.*
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 SOPO of the corresponding variable.*
3. **Generation of symbolic constraints:** *Compute the basic relations that can hold between each interval pair of the initial solution. Add to each*

relation a random number in the interval [*0, Nr*] *(1 ≤ Nr ≤ 13) of chosen basic Allen relations.*

5.3.2. *Generation of inconsistent TCSPs*

A TCSP is inconsistent if it does not contain a solution that satisfies all the temporal constraints. Randomly generated large size TCSPs are more likely to be inconsistent (this is why for generating consistent TCSPs we start by generating a solution and then add some "noise" to it). Thus the easiest way we found to generate inconsistent TCSPs is simply to generate the TCSP and check with the CSP based method that the TCSP is inconsistent (which is the case in general). Indeed, with the CSP based method, the inconsistency is detected very quickly (in general at the preprocessing level). Also our goal is to have a generation method that is not biased by any parameter.

More precisely, to randomly generate an inconsistent TCSP we generate a list of symbolic and numeric constraints (disjunctive relations and SOPOs) and check that a solution cannot be found. Each inconsistent problem of size n (n is the number of variables) is generated using the following steps:

1. **Generation of numeric constraints:** *Randomly pick n pairs of ordered values (x, y) such that x, y* ∈ [*0, . . ., 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* ∈ [*1 . . . y – x*]. *d is considered the duration of the event.*
2. **Generation of symbolic constraints:** *Randomly generate c constraints between the n events where c* ∈ [*1 . . .* $\frac{n(n-1)}{2}$] *(c =* $\frac{n(n-1)}{2}$ *in the case of a complete graph of constraints). Each constraint is a disjunction of a random number nb (nb* ∈ [*1 . . . 13*]*) of relations chosen randomly from the set of the 13 Allen primitives.*
3. **Consistency check of the generated problem:** *Perform the CSP based method on the generated problem. If the problem is consistent* **goto 1** *and generate another problem.*

The generated problems (consistent or inconsistent) can be characterized by their tightness, which can be measured, as shown in (Sabin and Freuder 1994) using the following definition:

> *The tightness of a CSP problem is the fraction of all possible pairs of values from the domain of two variables that are not allowed by the constraint.*

The tightness depends in our case on the parameters *Horizon, Nr* and the density of the problem.

Table 3. Comparative results for consistent problems.

Tightness	MCRW			SDRW			Tabu Search			BB
of the problem	qual	time	# moves	qual	time	# moves	qual	time	# moves	time
0.0002	0	0.12	5	0	2.67	80	0	0.17	4	0.10
0.0004	0	0.28	18	0	4.95	136	1	185	10000	12.7
0.001	0	0.46	28	0	8.24	193	0	0.6	16	217
0.002	0	0.95	68	0	11.22	212	2	294	10000	1615
0.0037	0	1.74	145	0	126	712	1	270	10000	1250
0.006	0	4	255	0	33	336	3	286	10000	1540
0.03	0	86	3713	33	33802	10000	12	349	10000	2730
0.044	0	73	1633	4	9595	10000	25	355	10000	3240
0.045	0	72	1633	4	9614	10000	16	376	10000	4536
0.058	0	15	433	74	12333	10000	12	364	10000	7765
0.1	0	12	332	0	34	225	0	112	211	22455
0.14	0	8.47	304	0	39	243	0	112	193	37600
0.35	0	181	2009	0	66	210	68	714	10000	87680
0.44	0	137	1291	220	8346	10000	63	646	10000	127000
0.55	0	315	2505	0	66	210	0	262	190	250000
0.67	372	13945	100000	0	130	297	0	422	224	480000

5.4. *Experiments on randomly generated temporal problems*

5.4.1. *Results*

Table 3 presents tests performed on randomly generated consistent problems of size 200 each, characterized as shown in the table. It gives a summary of the best results of MCRW, SDRW, Tabu Search and Branch and Bound for the chosen instances in terms of quality of the solutions. In the case of the approximation methods, the results correspond to the average time, number of moves and quality of the solution provided by each method. To obtain these results, the algorithms were run 100 times on each instance, each run being given a maximum of 100,000 moves in the case of MCRW and 10,000 moves in the case of SDRW and Tabu search. The parameter of each algorithm (the size of the Tabu list *tl_size* and the random-walk probability p) is fixed according to the best value found during the parametric study. Note that, as mentioned before in section 3, the cost in time of a move in the case of Tabu Search and SDRW is equal to N times the cost of a move in the case of the MCRW method, where N is the number of variables (events).

Table 4. Comparative results for non consistent problems.

Tightness	MCRW			SDRW			Tabu Search			BB
of the problem	qual	time	# moves	qual	time	# moves	qual	time	# moves	time
0.0002	**8**	0.44	32	**8**	4.5	107	**8**	0.28	6	1.12
0.001	**10**	0.7	53	**10**	10.26	199	**10**	242	3298	1.6
0.002	**2**	0.68	43	**2**	7.77	183	**2**	194	5812	21
0.0037	**14**	1237	45630	**14**	14.62	238	**14**	230	6543	2250
0.006	**20**	5.83	425	**20**	33	336	22	377	10000	1320
0.03	**21**	190	5406	32	3663	10000	85	341	10000	1220
0.044	**43**	853	25	46	4827	10000	45	255	10000	17400
0.1	**41**	10	318	**41**	106	233	91	257	10000	19880
0.14	**208**	10.14	279	**208**	37	215	230	434	10000	42300
0.35	**141**	259	3015	**141**	439	554	**141**	201	415	85600
0.44	**531**	105	271	**531**	82	216	**531**	48	195	122000
0.67	**858**	156	315	**858**	98	206	**858**	58	224	234000

From the data of Table 3, we can make the following observations regarding the approximation methods. For under-constrained and middle-constrained problems, the MCRW method always provides the best results. It almost always finds a complete solution within a reasonable amount of time which is not the case of the other two methods. It is also faster than the other two methods to find solutions of the same quality. However for over-constrained problems (see last row of Table 3) SDRW and Tabu Search have better performance. We can explain this by the fact that, for under constrained problems the initial configuration is in general of good quality. A complete solution can be obtained in this case by only changing the values of some conflicting variables (case of MCRW) instead of looking for the best neighbor which is much more expensive.

Table 4 presents tests performed on randomly generated inconsistent problems of size 200 each. For each instance, the exact method based on branch and bound is first performed in order to get the optimal solution (solution with the minimum number of violated constraints). The three algorithms are then run 100 times on each instance, each run being given a maximum of 100,000 moves in the case of MCRW and 10,000 moves in the case of SDRW and Tabu search.

From Table 4 we can make the same observations as for Table 3 i.e., the MCRW method is the algorithm of choice if we have to deal with under-constrained or middle-constrained problems. The effort made by SDRW

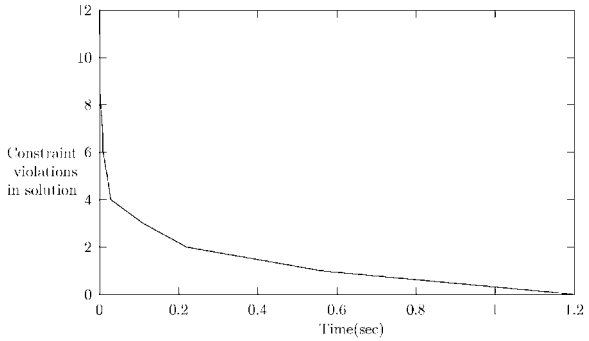

Figure 12. Averaged anytime curve. 100-variable problems, tightness = 0.10.

and Tabu search methods to look for the best neighbor helps only in the case of over constrained problems. When the quality value is in boldface, this means that it is the optimal one i.e., the approximation method has succeeded to get the optimal solution. As we can see on Table 4, comparing to Tabu Search and SDRW, MCRW succeeds in each case to get the optimal solution.

As we can easily see from Tables 3 and 4, the branch and bound method is slower than the approximation methods especially for middle-constrained and over-constrained problems. This is due to the branching factor during the backtrack search. Indeed, although using the cost function the branch and bound method avoids exploring many branches entirely, some backtrack (with exponential cost in time) is still needed to find paths of better quality. This is not the case of the approximation methods based on the iterative greedy algorithm (polynomial complexity in time). However, despite of this disadvantage, as we have seen before, the solution provided by the exact method is guaranteed to be optimal and thus can be used to check the goodness of the solutions returned by the approximation methods.

5.4.2. *Anytime curve*

As we said before, the method based on branch and bound is an exact method that has the ability to solve a temporal problem by giving a solution with a quality depending on the time allocated for computation. This can be shown using the results of tests that we have performed on 100 problems of size 100 each, with complete solutions, randomly generated as shown is subsection 5.3. The "anytime curve" reporting the results is presented in Figure 12. This curve is based on the average number of constraint violations found after each period of time. The CSP based method (resolution method) we have seen in section 3 took 0.44 seconds in average to get a complete solution when tested on the above randomly generated problems.

Note that the CSP based algorithm is faster than the branch and bound method in getting a complete solution (in case the problem is consistent) because of the following two reasons:

1. **The search space in the case of the CSP based method is reduced.** Indeed, the goal of the CSP based method is to check for the global consistency of the problem by looking for a complete solution (assignment of values to variables such that all constraints are satisfied). Two possible answers are provided by the CSP based algorithm:
   - yes (true) if the problem is consistent (does have a complete solution). The solution can be output as well.
   - no (false) otherwise.

   Thus local consistency techniques are first applied (in the preprocessing phase) to reduce the size of the search space (which will improve the running time of the backtrack search) by removing those values which have no chance to appear in a complete solution.
   In the other hand, the branch and bound method does not return a yes|no but a partial solution (with an optimal quality) anytime the program is interrupted. The partial solution provided may contain some inconsistent values. Thus all values of the initial problem should be kept and the branch and bound method has to deal with a very large search space (comparing to the CSP based one). This makes the branch and bound algorithm slower than the CSP based method.
2. **When exploring a given branch of the search space, the inconsistency is detected earlier in the case of the CSP based method.** Indeed, with the help of the local consistency techniques during the backtrack search, the inconsistency is detected early and the algorithm backtracks to explore another branch. In the case of the branch and bound method, even if the inconsistency is detected the algorithm keep exploring the branch as long as the estimated quality is better than the lower bound.

5.5. *Experiments on scheduling problems*

In this section we will present the results of experiments performed on job| machine scheduling problems. A particular example is described as follows:[4]

> *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 TemPro. 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 13) is the solution to the above problem provided by the CSP based method we have seen in section 3. Note that this solution is optimal[5] but not unique.

In order to evaluate and compare the performance of our methods on job|machine scheduling problems, we have performed experimental tests on job|machine scheduling problems each characterized by the parameters *nb_art* and *nb_mach* which correspond respectively to the number of articles and machines each problem has. Table 5 presents the running time in seconds

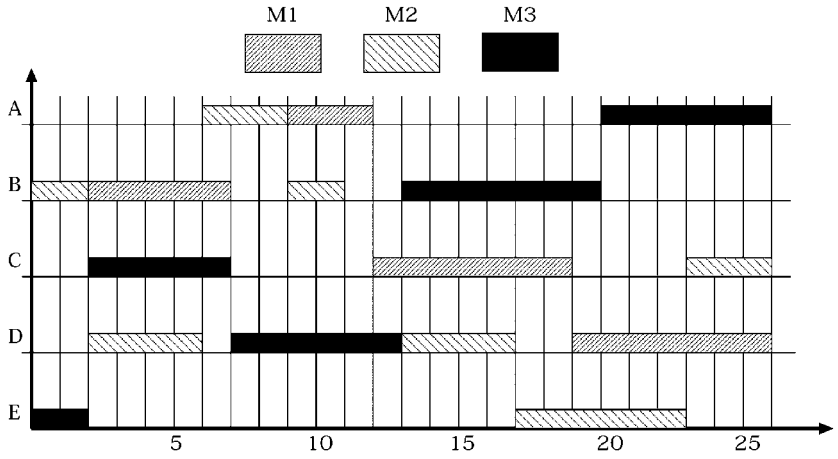

Figure 13. Optimal solution provided by the CSP based method.

needed by each of the exact and approximation methods to obtain the optimal solution for each problem. The first three columns correspond respectively to the number of articles, machines and generated events characterizing each problem. After performing preliminary tests on different job|machine scheduling problems, we have identified the best values for the parameters of the approximations methods, as follows.

- For MCRW, $p = 10$.
- For SDRW, $p = 18$.
- For Tabu Search, $tl\_size = 10$.

As we can easily see in Table 5 the running time of the MCRW method is better than the other approximation (SDRW and Tabu search) and exact (Branch and bound) methods especially for large size problems. This is mainly because the problems are in general not over-constrained and as we have seen before, this is the case where the MCRW presents the best results.

Table 6 presents the results of tests performed on scheduling problems when the value of *Horizon* (time before which all events should be achieved) is strictly smaller than the optimal value (26 in the case of the scheduling problem we have seen in example). In this case the problems become inconsistent. The parameter q corresponds to the quality of the solution provided by the method. The result provided by the CSP based method corresponds to the time needed to detect the inconsistency of the problem. This can either be achieved at the local consistency level (if the problem is not arc or path consistent than it is not consistent) or at the backtrack search level by exploring the entire search tree before deciding that the problem is inconsistent. However, as we have seen before, this method does not have the ability to provide the optimal solution (solution with minimum number of

Table 5. Comparative results for job|machine scheduling problems.

Problem			CSP based	MCRW	SDRW	Tabu Search	BB
nb_art	*nb_mach*	*nb_events*	Time	Time	Time	Time	Time
5	3	17	0.65	0.8	1.2	1.34	4.45
10	5	54	1.90	1.4	1.80	1.90	6.05
20	7	148	5.25	2.24	3.20	3.88	10.12
30	8	252	12.37	3.44	4.67	4.77	16.78
50	10	520	37	6.12	7.44	8.72	89
70	12	858	112	12.52	14	16.25	230
100	15	1600	12210	27	34	32	24560

Table 6. Comparative results for the inconsistent scheduling problem.

Problem			CSP based	MCRW		SDRW		Tabu Search		BB	
nb_art	*nb_mach*	*nb_events*	Time	Time	q	Time	q	Time	q	Time	q
5	3	17	0.92	1.10	2	1.2	5	1.34	8	3.93	2
10	5	54	1.90	2.12	5	2.44	7	3.02	13	5.54	5
20	7	148	3.25	3.56	9	3.88	12	3.78	15	11.12	9
30	8	252	8.25	6.34	13	7.02	22	6.66	17	21	13
50	10	520	238	11.17	22	12.02	40	14.12	37	1389	22
70	12	858	1512	78	30	112	58	428	47	23227	30
100	15	1600	24345	227	54	830	94	324	112	44578	54

unsolved constraints). Concerning the comparison of the other methods, we can argue in the same way as for Table 5 i.e., the MCRW method presents the best results and is the only one (comparing to the other approximation methods) that succeeds to have a solution with the optimal quality. Note that constraint violations (quality of the solution) correspond here to the overlapping between events processed by the same mono processor machine. The branch and bound method is used to guarantee the optimality of the solution returned by the approximation methods.

6. Conclusion and future work

In this paper we have presented a system that handles temporal problems involving both numeric and symbolic information (the architecture of the

system is presented in Figure 14). One of the main tasks of the system is to check and to maintain, in real time, the consistency of a given temporal constraint problem. This is very important for many industrial applications such as on line scheduling, planning and natural language processing where an incomplete solution should be provided without missing a given deadline. For this purpose, we use exact and approximation methods based respectively on branch and bound and local search techniques. Experimental tests on large size randomly generated temporal constraint problems and scheduling problems demonstrated that the Min Conflict Random Walk method is the algorithm of choice in the case of under constrained and middle constrained problems while Tabu Search and Steepest Descent Random Walk are the methods of choice in the case of over constrained problems. The exact method based on branch and bound is useful to verify the optimality and, therefore, the quality of the solution returned by the approximation methods.

In the near future, we intend to integrate our model TemPro into a planning system. Indeed, Planning systems based on the state space approach, for example, use a sequential notion of time (systems like TWEAK (Chapman 1987) and SNLP (McAllester and Rosenblitt 1991) are based on a ordering relation between operators). Our goal is to enrich the time representation of the planning system especially by handling numeric and symbolic constraints. A problem to plan will include, in addition to the implicit temporal relations (orderings generated during the planning process to solve some resource conflicts), a set of explicit temporal constraints. One of the main roles of the planner is to maintain, during the construction of the plan, the consistency of the latter given the temporal constraints. This will be handled by the resolution techniques we have developed.

Another perspective consists of solving temporal constraint problems in a dynamic environment. We may, for example, add new information or relax some constraints when there are no more solutions. In those cases we need to check if there still exist solutions to the problem every time a constraint has been added or removed. Adding temporal constraints can easily be handled by TemPro, we have just to put in this case the new constraint in the lists of constraints to be checked. However, constraint relaxation can not be handled by our techniques. Indeed, when we remove a constraint, our algorithms cannot find which value, that has been already removed, must be put back and which one must not. We must then transform our satisfaction techniques so that they will be able to save the reason why a certain value has been removed.

We also intend to extend our model TemPro in order to manage incomplete information, that is:

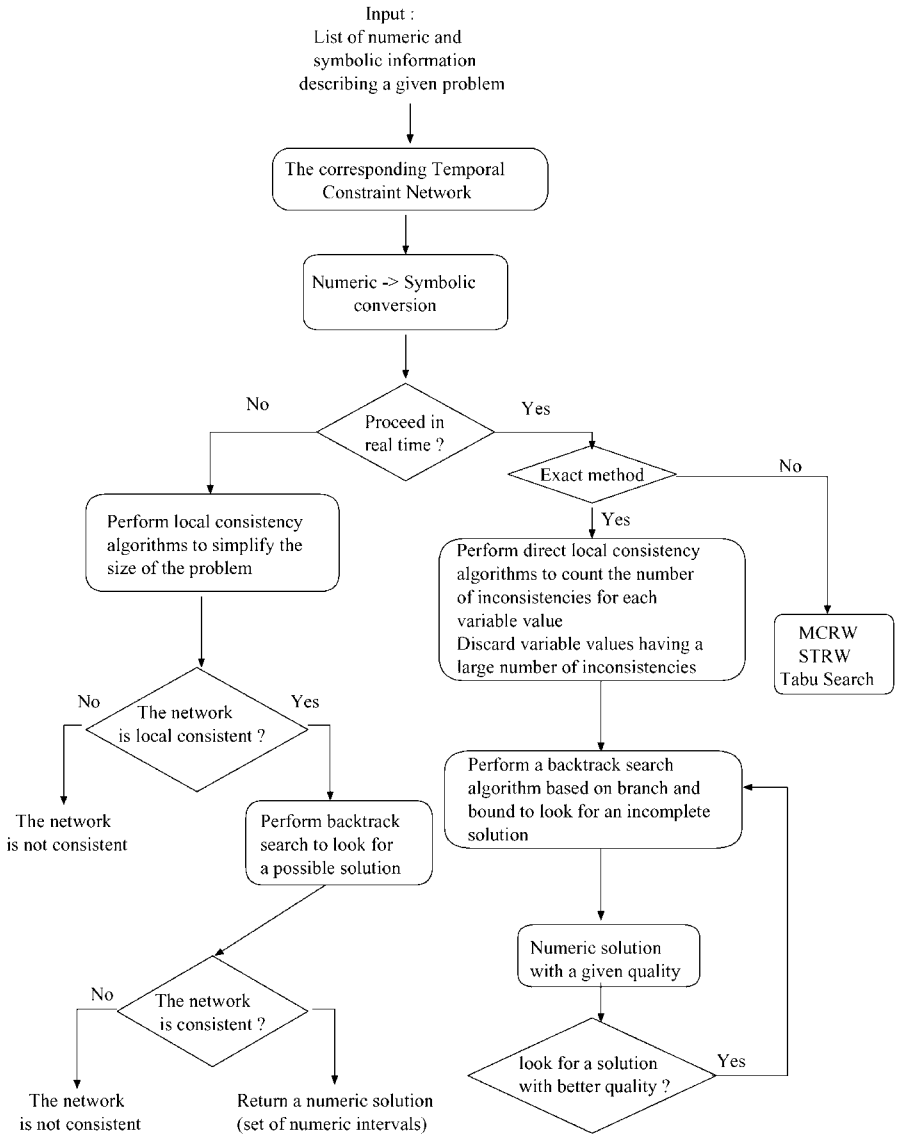

Figure 14. System solving temporal constraint satisfaction problems.

1. To handle estimated duration and temporal windows bounds. If we have to plan different tasks in an operating system for example, the corresponding processes will have an estimated duration depending on certain parameters. In this case we have to think of a new method of propagating estimated constraints.

2. To manage assumptions in the case where a given time information is incomplete. Managing assumptions can be handled by an assumptions truth maintenance systems (ATMS) (de Kleer 1986), however we may also use constraint satisfaction techniques in this case. Indeed, ATMS and CSP techniques embody the same computational trade-offs. CSP local consistency algorithms, for example, are motivated by exactly the same set of problems solving concerns as the ATMS algorithms. Also, most of the CSP local consistency algorithms themselves have almost identical analogs in the ATMS framework.

Notes

1. A binary CSP involves a list of variables defined on finite domains of values and a list of binary relations between variables.
2. Note that this name and the corresponding acronym was used in (Dechter et al. 1991). A comparison of the approach proposed in this later paper and our model TemPro is described in (Mouhoub et al. 1998).
3. 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 is the domain size of the different variables.
4. This problem is taken from (Laborie 1995).
5. The total processing time of all machines needed to produce the five items, 26 seconds, is minimal.

References

Allen, J. (1983). Maintaining Knowledge about Temporal Intervals. *CACM* **26**(11): 832–843.

Baptiste, P. & Pape, C. L. (1995). Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions. In *Third International Conference on Computer Integrated Manufacturing*. Singapore.

Bessière, C., Isli, A. & Ligozat, G. (1996). Global Consistency in Interval Algebra Networks: Tractable Subclasses. *ECAI'96*. Budapest, Hongrie.

Bessière, C. & Régin, J. C. (2001). Refining the Basic Constraint Propagation Algorithm. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 309–315. Seattle, WA.

Chapman, D. (1987). Planning for Conjunctive Goals. *Artificial Intelligence* **32**: 333–377.

de Kleer, J. (1986). An Assumption-based Truth Maintenance System. *Artificial Intelligence* **28**: 127–162.

Dechter, R., Meiri, I. & Pearl, J. (1991). Temporal Constraint Networks. *Artificial Intelligence* **49**: 61–95.

Dechter, R. & Pearl, J. (1988). Network-based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence* **34**: 1–38.

Freuder, E. C. & Wallace, R. J. (1992). Partial Constraint Stisfaction. *Artificial Intelligence* **58**: 21–70.

Golumbic, C. & Shamir, R. (1993). Complexity and Algorithms for Reasoning about Time: A Graphic-theoretic Approach. *Journal of the Association for Computing Machinery* **40**(5): 1108–1133.

Haralick, R. & Elliott, G. (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* **14**: 263–313.

Hwang, C. & Shubert, L. (1994). Interpreting Tense, Aspect, and Time Adverbials: A Compositional, Unified Approach. In Proceedings of *The First International Conference on Temporal Logic, LNAI, Vol 827*, 237–264. Berlin.

Laborie, P. (1995). *Une approche intégrée pour la gestion de ressources et la synthèse de plans*. Ph.D. thesis, École Nationale Supérieure des Télécommunications.

Laborie, P. & Ghallab, M. (1995). Planning with Sharable Resource Constraints. *IJCAI-95*: 1643–1649.

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

McAllester, D. & Rosenblitt, D. (1991). Systematic Nonlinear Planning. *AAAI-91*: 634–639.

Minton, S., Johnston, M. D., Philips, A. B. & Laird, P. (1992). Minimizing conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* **58**: 161–205.

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

Orgun, M. (1996). On Temporal Deductive Databases. *Computational Intelligence* **12**(2): 235–259.

Pape, C. L. & Smith, S. (1987). Management of Temporal Constraints for Factory Scheduling. *Temporal Aspects in Information Systems Conference*, 165–176. Sophia Antipolis, France.

Sabin, D. & Freuder, E. C. (1994). Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. 11th ECAI*, 125–129. Amsterdam, Holland.

Selman, B. & Kautz, H. A. (1993). An Empirical Study of Greedy Local Search for Satisfiability Testing. *AAAI'93*: 46–51.

Song, F. & Cohen, R. (1991). Tense Interpretation in the Context of Narrative. *AAAI'91*: 131–136.

van Beek, P. & Manchak, D. W. (1996). The Design and Experimental Analysis of Algorithms for Temporal Reasoning. *Journal of Artifical Intelligence Research* **4**: 1–18.

Wallace, R. J. (1995). Partial Constraint Satisfaction. *Lecture Notes in Computer Science* **923**: 121–138.

Zhang, Y. & Yap, R. H. C. (2001). Making AC-3 an Optimal Algorithm. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 316–321. Seattle, WA.