

Evaluation of Algorithms to Satisfy Disjunctive Temporal Constraints in Planning and Scheduling Problems

M. A. Salido, A. Garrido, F. Barber

Dpto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Camino de Vera s/n 46071
Valencia, Spain
msalido@dsic.upv.es, agarridot@dsic.upv.es, fbarber@dsic.upv.es

Abstract

The management of constraints either implicit or explicit in planning and scheduling environments is commonly a very hard task. The way to manage them in the proper way is becoming an important area of study. Moreover, the common approach in planning and scheduling problems is that the constraints to satisfy are disjunctions on intervals (i.e., they imply several possible alternatives). Hence, the complexity of their management is vastly increased (NP-complete complexity). If we use a closure process, there exist two possible solutions to manage these constraints: algorithms that maintain the input and derived constraints and algorithms that only maintain the input constraints. The former require large amounts of memory to store all the generated constraints, whereas the latter do not require large amounts of memory. Here, we present an analysis and evaluation of algorithms to satisfy temporal constraints on metric-disjunctive intervals in scheduling environments using this last kind of algorithms.

1 INTRODUCTION

Planning and Scheduling are two active and relevant areas in Artificial Intelligence which have become of great interest to researchers because of their applications in real problems. There exist many problems (manufacturing, transport problems, planning of production processes, etc.) which should be treated as planning problems with temporal constraints and resource usage constraints. Classical methods for solving them are based on resource allocation.

In Operational Research techniques, the partial sequence of actions is already known, and the actions are principally based on resource allocation. In contrast, Artificial Intelligence methods are used to determine the correct plans. The planner builds a plan as a partially-ordered sequence of actions to reach a goal and the scheduler must ensure that this plan is executable (resource allocation and temporal constraint satisfiability). Nevertheless, the increasing complexity of current problems obligates us to use new, more flexible and more powerful approaches.

1.1 Integrated Planning and Scheduling System

In this section, we present a high-level general view of our integrated system (Figure 1). Our work is developed from an integrated architecture of planning and scheduling (Garrido et al. 1999). The main goal of this integration is to guarantee plan executability and satisfy

the problem constraints through the planner and the scheduler in a simultaneous and interactive way. The planning system searches through several alternative partial plans, dispatching the constraints and resource requirements that the scheduler must check. Hence, the scheduler guarantees data constraints and resource assignments are satisfied. Due to this interactive behaviour, the need for improving both planning and scheduling process efficiency becomes more important. In this integrated system, the planning improvements are focused on techniques to reduce the search space, solve conflicts (threats among actions), grouping primitive actions in macro-actions, etc. On the other hand, the main improvements in the scheduler deal with a more efficient management of constraints: resource usage constraints and metric-disjunctive temporal constraints (Barber, 2000). We will focus specifically on the scheduling process and on the temporal constraint management.

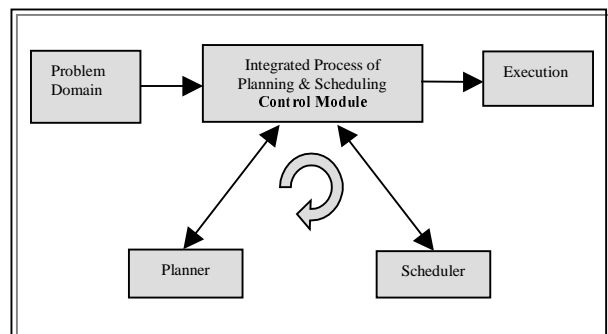


Figure 1. Integrated Planning and Scheduling environment

Since scheduling processes imply temporal constraints about actions and resources, one of the main improvements in the schedulers deals with a more efficient management of constraints. These constraints represent temporal intervals, which can indicate an order of execution, allowing us to represent a wide set of possible solutions. Moreover, these constraints can be disjunctive or non-disjunctive. If the constraints are non-disjunctive, only one order of execution will be possible. However, if the constraints are disjunctive, different (and alternative) orders of execution will be feasible (Baptiste & Le Pape, 1995; Dechter et al., 1991).

Following, we are going to define a simple, typical example in which there exist metric-disjunctive temporal constraints (Dechter et al., 1991). In Figure 2, the input constraints (the explicit ones) of the example are shown in a temporal graph. This example will help us to explain the nature of the problem which is described in this paper:

Michael goes to work either by bus (at least 60') (R_1), or by car [30', 40'] (R_2). Anthony goes to work either by train [40', 50'] (R_3), or by car [20', 30'] (R_4). Today Michael left home (t_1) between 7:10 and 7:20 (R_0), and Anthony arrived (t_2) at work between 8:00 and 8:10 (R_0). We also know Michael arrived (t_2) at work about [10', 20'] (R_0) after Anthony left home (t_3).

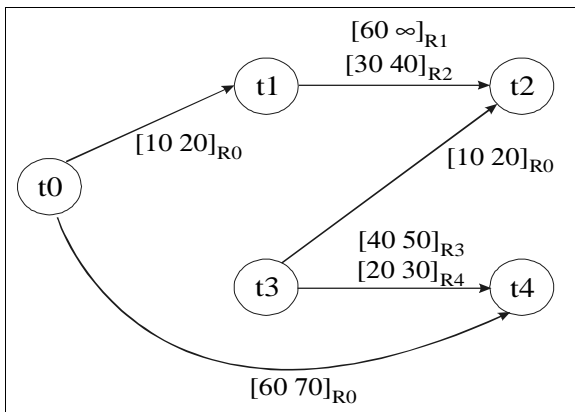


Figure 2. Input constraints of the example

Some queries can be made about the information in the example:

- Who arrives at work earlier? Michael or Anthony?
- Could all these disjunctions be satisfied at the same time?
- Which kinds of vehicles are going to be used in the final solution?
- Is this example's information consistent, i.e., is there a solution?

We might use several techniques in order to answer this last question. One of these techniques is based on traditional methods of *Constraint Satisfaction Problems* (CSPs) that work on a previously known set of constraints obtaining a final solution which answers the

previous questions. Nevertheless, the behaviour of this technique is not adequate enough due to its lack of flexibility. For each new set of constraints (which are the result of including or excluding constraints), a CSP process must resolve the entire problem in order to obtain a new solution. However, when the set of constraints is modified, the previous solution may become invalid.

On the other hand, other techniques that can be used to solve problems with metric-disjunctive temporal constraint are based on closure processes. The main goal of closure techniques is to guarantee the consistency of the existing constraints, which may be included and/or excluded by means of an interactive behaviour. There exist several levels of consistency depending on the solution exigency, from the lowest exigent levels (for instance, path consistency) to the most exigent ones (global consistency) and two alternative to guarantee this consistency:

Algorithms that maintain derived constraints

In this case, derived constraints which are obtained from the set of input constraints are explicitly represented in the temporal network. Consequently, these algorithms require large amounts of memory to store all the derived constraints in the closure process. For instance, in the previous example (Figure 2), the derived constraints would be $t_0 \rightarrow t_2$, $t_0 \rightarrow t_3$, $t_1 \rightarrow t_3$, $t_1 \rightarrow t_4$ and $t_2 \rightarrow t_4$ (see Figure 3). When a constraint between two temporal points is asserted into the system, these algorithms check its consistency with the existing constraint. If the new constraint is consistent, the resulting constraint to maintain will be the most restrictive combination between them, and the input constraint will not be maintained. Due to the fact that the graph is always propagated, retracting a constraint is a very complex task (it is difficult to determine the input constraint) and it obligates us to repeat the closure process for every input constraint.

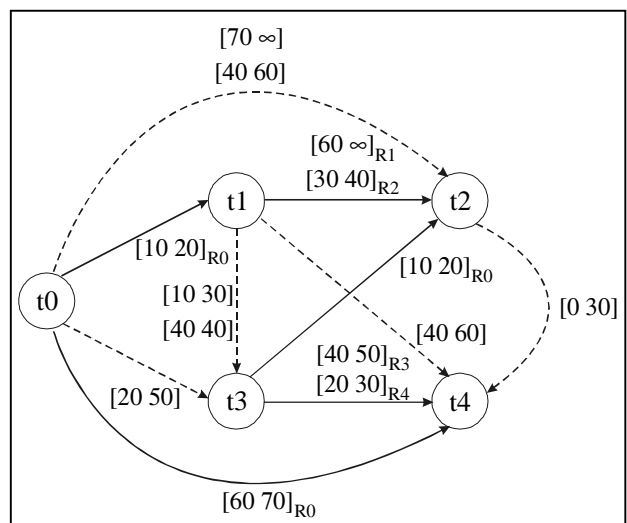


Figure 3. Derived constraints of the example

Algorithms that only maintain input constraints

These algorithms only maintain input constraints (the explicitly asserted ones which appear in Figure 2). The main advantage of these algorithms is that they do not require large amounts of memory to store the derived constraints because they do not carry out any propagation process between the new constraint and the existing ones. For this reason, these algorithms may be used in an attempt to reduce the complexity of asserting new constraints and they ease the process of retracting some asserted constraints into the system. When a new constraint between two temporal points (nodes) is inserted into the system, the algorithm retrieves the most restrictive constraint (the minimal one) between these two points. Next, the algorithm checks whether the new constraint is consistent with the retrieved one. If it is consistent, the new constraint is accepted, and if not, it is rejected. The main difficulty is to calculate the minimal constraint in a nonpropagated disjunctive graph in the most efficient way. It is a complex task because there exists an exponential number of paths that represent the constraints to be calculated. Since the graph may contain both negative arcs in parallel (disjunctions) and circuits, we must find all paths in the associated graph in order to obtain the *shortest path* which represents the minimal temporal constraint (Goldfarb, 1991).

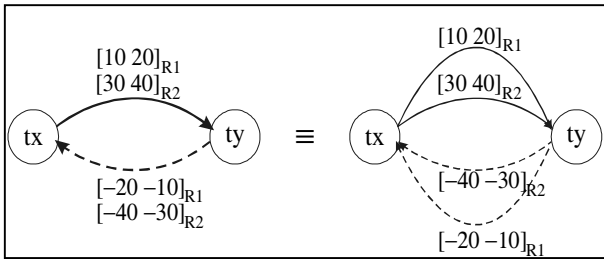


Figure 4. Equivalence between disjunctions and arcs in parallel

As can be observed, problems with metric-disjunctive temporal constraints may be represented by means of a directed graph with negative weights and arcs in parallel (Figure 4). Nodes represent time points, and arcs between nodes represent metric-disjunctive temporal constraints among time points. The negative weights are temporal intervals, and arcs in parallel are denoted by means of disjunctive weights. For instance, let x_i and x_j be two nodes. The metric-disjunctive constraint might be

$$x_i \{ [d_k, D_k], [d_l, D_l], \dots \} x_j : d_k, D_k, d_l, D_l \in \mathfrak{R}.$$

Hence, these problems may be treated as problems over temporal graphs. One of the most studied problems in graph theory is the shortest path problem or the k -shortest paths problem in a network (Ford & Fulkerson, 1974; Shier, 1979; Goldfarb, 1991). Over the last four decades, several algorithms have been proposed for

solving the shortest path problem or the k -shortest paths problems (Ravi et al., 1992). Many of these methods are variants of the well-known Bellman-Ford algorithm (Aho & Ullman, 1992; Ford & Fulkerson, 1974). Our problem attempts to generate algorithms for finding all paths (without circuits) from a node source x_1 to a node sink x_n , which represent time points. Therefore, we have proposed a *unidirectional search algorithm* and a *bidirectional search algorithm*. The *unidirectional search algorithm* generates a spanning tree to find all paths between two nodes. This algorithm obtains all paths by means of a strategy which is similar to the *inorder* strategy. The number of generated nodes is exponential according to the number of nodes of the graph, although its spatial cost is linear. In contrast, the *bidirectional algorithm* does not generate the entire spanning tree to find all paths between two nodes. The search process starts forward from a node (the source node) and backward from the other node (the sink node). When the partial paths meet, a new complete path is found. Thus, the number of generated nodes is decreased with regard to the *unidirectional algorithm* (it is theoretically the square root), but the spatial cost is increased because both forward and backward trees must be simultaneously stored in memory.

In section 2, we present the specification of the *unidirectional* and *bidirectional search algorithm*. The analysis of the algorithms and their evaluation are presented in section 3 and section 4, respectively. Conclusions and future lines of work are discussed in section 5.

2 SPECIFICATION OF THE ALGORITHMS

In this section, both the *unidirectional* and *bidirectional search algorithm* are presented. We begin introducing the necessary notation used in the algorithms.

2.1 Preliminaries. Notation

Definition 1

Let $G = (V, E)$ be a connected graph. V is a finite non-empty set of nodes. E is a set of pairs of nodes called arcs. $V(G)$ and $E(G)$ represent the set of nodes and arcs of graph G respectively. We assume $V = \{x_1, x_2, \dots, x_n\}$, $|V| = n$ and $|E| = l$. Node x_i of the arcs (x_i, x_j) , or (x_j, x_i) , is said to be adjacent to node x_j . We denote $Adj(x_i)$ as the set of x_i adjacent nodes, and we say $deg(x_i)$ is the x_i degree.

Definition 2

A *path* from node x_1 to the node x_n in a connected graph is an ordered sequence

$[(x_1, x_2), (x_2, x_3), \dots, (x_i, x_{i+1}), \dots, (x_{n-1}, x_n)]$ of nodes such that there is an arc from each node to the next one, i.e., (x_i, x_{i+1}) is an arc for $i = 1, 2, \dots, n-1$. Without loss of generality, we denote this sequence by (x_1, x_2, \dots, x_n) . A *valid path* is a path without repeated nodes. The maximal *length* of a valid path (x_1, x_2, \dots, x_n) in a graph of n nodes is $n-1$, i.e., the number of arcs along the path.

Definition 3

Let $G = (V, E)$ be a connected graph and let (x_i, x_j) be an arc. We can say x_i is a *predecessor* of x_j and x_j is a *successor* of x_i .

Definition 4

We denote the ordered list of the x_i forward predecessors as

$$P_f(x_i) = (x_0, x_1, \dots, x_{i-1}) : x_j \neq x_p \quad \forall j \neq p, 0 \leq j, p \leq i-1.$$

Analogously, we denote the ordered list of the x_i backward predecessors as

$$P_b(x_i) = (x_{i+1}, x_{i+2}, \dots, x_n) : x_j \neq x_p \quad \forall j \neq p, i+1 \leq j, p \leq n.$$

Definition 5

$\bar{P}_f(x_i)$ is 'the x_i direct predecessor' in forward exploration, whereas $\bar{P}_b(x_i)$ is 'the x_i direct predecessor' in backward exploration.

Definition 6

The set of x_i valid predecessors in forward exploration (nodes which do not form circuits) is $Suc_f(x_i) = Adj(x_i) \setminus P_f(x_i)$, and the set of x_i backward true predecessors is $Suc_b(x_i) = Adj(x_i) \setminus P_b(x_i)$.

Definition 7

We denote $(a_1, a_2, \dots, a_n)^T$ by (a_n, \dots, a_2, a_1) .

2.2 The unidirectional search algorithm

The *Unidirectional Search Algorithm* is a well-known algorithm which finds all paths between nodes x_1 and x_n by generating the spanning tree by means of a unidirectional strategy. It starts from the node source x_1 in level 0. Level 1 is formed by the adjacent nodes of the node x_1 . Thus, level i is formed by the adjacent nodes of level $i-1$ which have not been processed in that branch (i.e. $Suc(x_i)$). Hence, the algorithm terminates when all paths have been found.

2.3 The bidirectional search algorithm

The *Bidirectional Search Algorithm* is an algorithm which finds all paths between nodes x_1 and x_n and, therefore, it finds the minimum constraint between x_j and x_n . This algorithm works on a bidirectional strategy in which it finds paths starting from the node source x_j in a forward direction and from the node sink x_n in a backward direction and varying the direction of the expansion alternatively. Hence, the complete path will be found when both semi-paths meet.

Notation

F_f = Set of frontier nodes generated in the forward spanning.

F_b = Set of frontier nodes generated in the backward spanning.

H_f = Set of leaf-nodes generated in forward exploration.

H_b = Set of leaf-nodes generated in backward exploration.

C_f = Set of reached nodes in forward exploration.

C_b = Set of reached nodes in backward exploration.

C = Set of found paths.

\oplus = List concatenation operator.

Algorithm

Bidirectional Search Algorithm (Graph, x_1, x_n)

; Initialisation stage

$C_f := \{x_1\}, H_f := \{x_1\}, F_f := \emptyset;$

$C_b := \{x_n\}, H_b := \{x_n\}, F_b := \emptyset; C := \emptyset;$

for $i = 1$ **to** n **do**

$P_f(x_i) := nil, P_b(x_i) := nil$

endfor

if $n \text{ MOD } 2 = 0$ **then** $z = n \text{ DIV } 2$

else $z = (n-1) \text{ DIV } 2$

endif

; Search stage

for $k=1$ **to** z **do**

Search $(f, b, F_m, H_m, C_v, C_m, C; C_m, H_m, F_m, C);$

forward

Search $(b, f, F_m, H_m, C_v, C_m, C; C_m, H_m, F_m, C);$

backward

endfor

; Odd stage

if $n \text{ MOD } 2 = 1$ **then**

Search $(f, b, F_m, H_m, C_v, C_m, C; C_m, H_m, F_m, C);$

return C

endAlgorithm

Algorithm 1. The Bidirectional Search Algorithm

Algorithm Search (**I** : $m, v, F_m, H_m, C_v, C_m, C$; **O** : C_m, H_m, F_m, C)

```

forall  $x_i \in H_m$  do
  if  $x_i \in C_v$  then
    forall  $x_j \in Suc_m(x_i)$  do
      if  $x_j \in \bar{P}_v(x_i)$  then
        if  $\{P_m(x_i) \oplus \{x_i\} \oplus \{P_v(x_i)\}^T\} \not\subset C$  then
           $C \leftarrow C \cup \{P_m(x_i) \oplus \{x_i\} \oplus \{P_v(x_i)\}^T\}$ ; new path has been found
           $F_m \leftarrow F_m \cup Suc_m(x_i)$ ; we update  $F_m$ 
           $C_m \leftarrow C_m \cup Suc_m(x_i)$ ; we update  $C_m$ 
        forall  $x_j \in Suc_f(x_i)$  do
           $P_m(x_j) := P_m(x_j) \oplus \{x_i\}$ ; we update the  $x_j$  predecessors
         $H_m \leftarrow m$ ; we assign to set of leaf-nodes  $H_m$  the set  $m$ 
         $F_m \leftarrow \emptyset$ ; we empty the set of frontier nodes
      end;

```

Algorithm 2. The Search Algorithm

Theorem 1

Let $G = (V, E)$, $|V| = n$, $|E| = l$ be a connected graph. The bidirectional search algorithm finds all paths from a node source x_1 to node sink x_n .

Proof

We assume a path $(x_1, \dots, x_{k-1}, x_k, x_{k+1}, \dots, x_n)$ exists, and the bidirectional search algorithm does not find it. We will reason over this premise and we will demonstrate that this path has been found by the algorithm (arriving at a contradiction).

if $\{x_1, x_2, \dots, x_n\} \not\subset C \Rightarrow \exists x_k \in \{x_1, x_2, \dots, x_n\}$:

- a) $x_k \notin H_f \wedge x_k \notin H_b$ for any H_f, H_b
- or
- b) $x_k \in H_f \vee x_k \in H_b$, but $Suc_{(f,b)}(x_k) = \emptyset$

a) If $x_k \notin H_f \wedge x_k \notin H_b \Rightarrow \neg \exists x_i : x_k \in Suc_{(f,b)}(x_i) \Rightarrow Deg(x_k) = 0$ (contradiction) because the graph is connected.

b) $x_k \in H_f \vee x_k \in H_b$ but $Suc_{(f,b)}(x_k) = \emptyset$

We have assumed that $\{x_1, \dots, x_{k-1}, x_k, x_{k+1}, \dots, x_n\}$ is a path $\Rightarrow Suc_f(x_k) = x_{k+1}$ and

$Suc_b(x_k) = x_{k-1} \Rightarrow Suc_{(f,b)}(x_k) \neq \emptyset$ (contradiction).

All paths are going to be found because the algorithm expands both the forward and the backward tree until the $n/2$ level. Consequently, no path will be lost due to the fact that the algorithm finds all the paths of length $n-l$, the maximal length of a valid path without circuits in a graph with n nodes.

3 ANALYSIS OF THE ALGORITHMS

In this section, we present both a temporal and spatial analysis of the proposed algorithms in section 2. Their respective complexities are based on the number of nodes in the initial graph.

3.1 The unidirectional search algorithm

Let n be the number of nodes in the graph. The branching factor is determined by the number of descendent nodes from each node. Thus, the effective branching factor B_f is the maximum of these numbers. The maximum depth of the generated tree is bounded by the maximum length of all the valid paths. Since the valid paths do not contain circuits, the maximum length is lower than the number of nodes (specifically, the maximum length will be $n-l$). Therefore, the maximum number of generated nodes is given by the following formula:

$$1 + \sum_{i=1}^{n-l} \prod_{j=0}^{i-1} (B_f - j) \in O(B_f^{n-l}) \in O(B_f^n)$$

However, the real number of nodes is lower because a new node can not be generated due to a cycle in the current path. Moreover, as B_f is always lower than n , the temporal complexity in terms of n is $O(n^n)$.

The unidirectional search algorithm obtains all paths by means of a strategy which is similar to the *inorder* strategy. Therefore, it only maintains one complete path each time as maximum. Hence, the spatial complexity is $O(n)$.

3.2 The bidirectional search algorithm

In order to calculate the temporal complexity, the bidirectional algorithm behaves as two unidirectional algorithms. On the one hand, it starts from the source node towards the sink node. On the other hand, it starts from the sink node towards the source node. Thus, the two generated trees have exactly half the depth (instead of generating a n -depth tree, two $n/2$ -depth trees are generated). Consequently, the number of nodes is given by the next formula:

$$2 \cdot \left(1 + \sum_i^{(n-1)/2} \prod_{j=0}^{i-1} (B_f - j) \in O(B_f^{(n-1)/2}) \right) \in O(B_f^n)$$

As can be deduced of this formula, the asymptotic spatial complexity is $O(n^n)$. Its complexity is the same as that of the unidirectional search algorithm, but its practical behaviour is much better (because it approximately generates the square root of nodes).

On the other hand, the spatial complexity is greater than the unidirectional complexity. In order to be able to join the partial paths found, it is necessary to maintain the two whole trees. Hence, the required memory is proportional to the number of generated nodes, and the spatial complexity is

$$O(2 \cdot n^{(n-1)/2}) \in O(n^{n/2}) \in O(n^n)$$

4 EVALUATION OF THE ALGORITHMS

In order to evaluate the performance of the two proposed algorithms, we implemented them in *Common Lisp*. The computer used in our tests was a Sun Ultra 10 Sparc with 256 Mb. of memory and with the SunOS 5.7 operating system.

We evaluated the algorithms by their execution time and the number of nodes generated. Once the graph is defined, the algorithms carry out a search process between two nodes which are randomly chosen. The selected graphs were random graphs which consisted of a set of nodes (from 20 to 100) and a set of arcs, which represented 2-disjunctive and non-disjunctive constraints. We limited the number of disjunctive constraints to 10. This implies having an equivalent number of $2^{10}=1024$ different non-disjunctive graphs for each generated graph. In addition, we varied the branching factor, from 1.1 up to 1.5 which implies varying this factor from 2.2 up to 3.

The mean and variance of the execution time (in hundredths of a second) and the number of nodes generated in a 20-node graph for the unidirectional algorithm are shown in Table 1. We modified the effective branching factor B_f from 1.1 up to 1.5. Table 2 shows the same values for the bidirectional search

algorithm as are shown in Table 1 for the unidirectional search algorithm.

Unidirectional (20 nodes)	Time (hs)		Nodes Generated	
	Mean	Variance	Mean	Variance
$B_f = 1.1$	0	0	111.6	72.4
$B_f = 1.2$	0	0	148.4	46.4
$B_f = 1.3$	40	54.8	636	229.4
$B_f = 1.4$	40	54.8	3225.2	1959.3
$B_f = 1.5$	60	54.8	3927.4	2424.3

Table 1. Execution time and nodes generated in the Unidirectional Search Algorithm (20-node graph).

Bidirectional (20 nodes)	Time (hs)		Nodes Generated	
	Mean	Variance	Mean	Variance
$B_f = 1.1$	0	0	86.8	14.3
$B_f = 1.2$	20	44.7	125.8	26.8
$B_f = 1.3$	20	44.7	471	77.8
$B_f = 1.4$	480	164.3	1378.4	297.7
$B_f = 1.5$	940	634.8	2167.8	1179.2

Table 2. Execution time and nodes generated in the Bidirectional Search Algorithm (20-node graph).

As can be observed in the previous tables, the mean of the execution time is higher in the bidirectional algorithm although the number of nodes generated is lower. This average time is higher due to the fact that, in the computation time in each node is much higher in the bidirectional search algorithm: it is necessary to check whether a path has been found by joining two semi-paths (one from the forward tree and the other semi-path from the backward tree). As analysed in section 3, the number of nodes generated is lower in the bidirectional search algorithm. Theoretically, the bidirectional algorithm asymptotically generates the square root of the nodes generated by the unidirectional search algorithm. However, the number of nodes generated in practice is approximately only half. The variance values are quite high due to the fact that the results obtained are greatly dependent on the topology of the graph.

In Figure 5 and Figure 6, we present the graphics of the nodes generated in the two search algorithms for each graph of our tests.

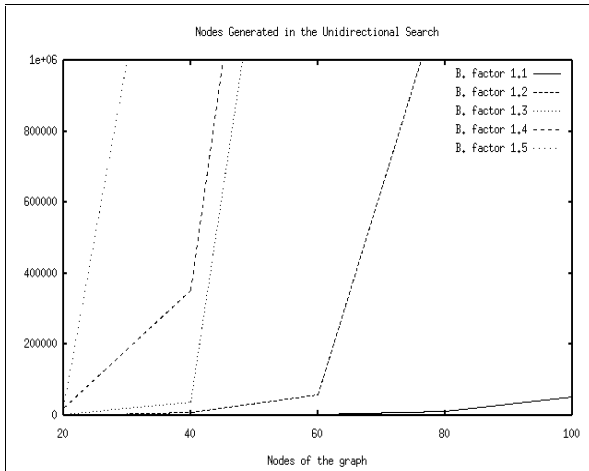


Figure 5. Nodes generated in the unidirectional search

As can be observed in Figure 5, the higher the branching factor, the greater the number of nodes generated. For instance, since a higher branching factor implies a greater number of arcs, the unidirectional algorithm is not able to manage 40-node graphs with a branching factor higher than 1.3.

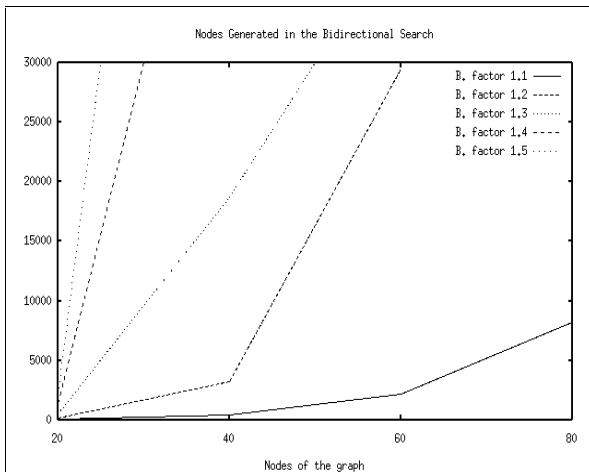


Figure 6. Nodes generated in the bidirectional search

In Figure 6, the number of nodes generated is lower than in Figure 5 because the bidirectional search algorithm generates less nodes for the same graph. In this case, the bidirectional search algorithm has generated less than 3500 nodes for a 40-node graph with branching factor 1.2, whereas the unidirectional search algorithm has generated more than 7000-nodes for the same graph.

The graphics of the execution time are presented in Figure 7 and Figure 8. In these figures, the execution time demonstrates that the bidirectional search algorithm takes much longer than the unidirectional search algorithm for the same graphs and the same branching factors. For instance, the bidirectional search algorithm takes 13420 hundredths of second in the 40-node graph

with branching factor 1.3 and, on the contrary, the unidirectional search algorithm takes only 460.

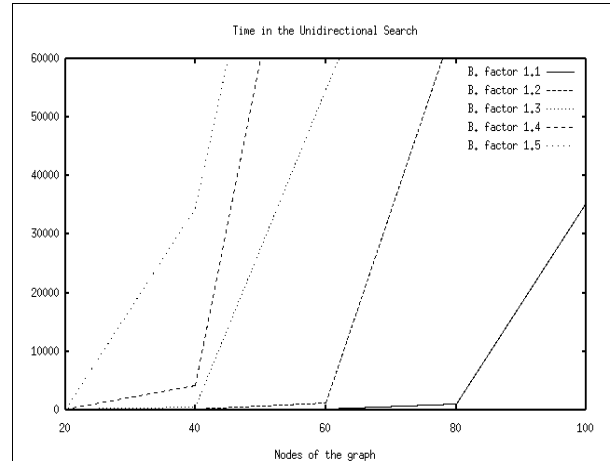


Figure 7. Execution time for the unidirectional search

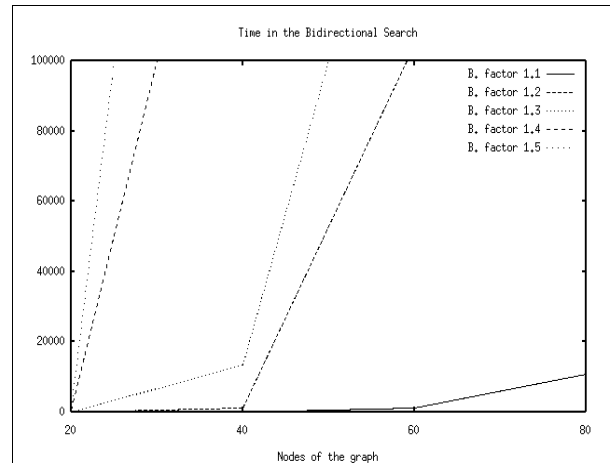


Figure 8. Execution time for the bidirectional search

Finally, it is important to note that these algorithms are not applicable to complete graphs with many nodes (in which there exists an arc between each pair of nodes). Due to the huge number of existing paths, it is impossible to obtain all the paths. For instance, in a graph with only 20 nodes, the number of existing paths is approximately 10^{16} .

5 CONCLUSIONS

In this paper, we have presented two algorithms to find the most restrictive constraint between two temporal points in a temporal graph. Since each arc represents a constraint on temporal disjunctive intervals, the arc's weight can either be positive or negative. Therefore, there does not exist an admissible heuristic that avoids having to find all the paths in order to obtain this minimal constraint.

These proposed algorithms can also be used in other problems based on graph exploration. However, even though the number of generated nodes is decreased by

using the bidirectional search algorithm, the needed storage space is increased.

As can be deduced from the results presented, these kinds of algorithms are not applicable to graphs with many nodes. Graphs with more than 100 nodes become unmanageable, specially if the branching factor is higher than 1.5 or 2. As can be seen in the results of the comparative study, the bidirectional algorithm generates fewer nodes than the unidirectional one. However, the spatial cost of the unidirectional algorithm is lower than the bidirectional search algorithm cost. The results demonstrate the suitability of each algorithm for use in the management of temporal constraints. These methods are not adequate enough in many real problems due to the difficulty of finding the optimal solution. Thus, our work is focused on reducing this complexity by means of the following techniques:

- Heuristic techniques in order to solve real problems in polynomial time. We can reduce the complexity of the search processes by using a heuristic that helps us to decide which path must be generated, and which must be discarded, according to some criteria. These criteria might lead to a path which does not represent the most restrictive constraint and we would have to choose between a nonminimal constraint obtained in a faster way or a minimal constraint obtained in a slower way.
- Combined CSP and closure techniques to improve the behaviour of planning and scheduling processes (Alfonso & Barber, 1999).
- Other techniques that do not guarantee the total consistency, for example path consistency.
- The use of new data structures which might improve the behaviour of these processes allowing us to have more powerful tools for solving real problems of planning and scheduling.

We can also use other methods that diminish the consistency of the generated graph (Freuder, 1982) and, therefore, decrease the complexity of the process.

Another interesting idea for the application of these algorithms to scheduling processes is to work on nondisjunctive graphs, which can be solved in polynomial time. When a disjunctive constraint appears, the method will select one of the disjunctions (according to criteria such as slack, due time, etc.) and it will ignore the other ones (Alfonso & Barber, 1999). If the selection has been appropriate the other disjunctions will be discarded. However, if the selection has been inappropriate, a backtracking process will be necessary in order to continue through another disjunction. In this case, the criteria are focused on minimising the number of backtracking stages.

ACKNOWLEDGEMENTS

This work has been proposed in the *Intelligent Planning & Scheduling Group* of the Polytechnic University of Valencia (<http://www.dsic.upv.es/users/ia/gps>) and

partially supported by the grant CICYT/TAP98-0345 from the Spanish government.

REFERENCES

- A.V. Aho and J.D. Ullman, 'Foundations of Computer Science', *Computer Science Press*, (1992).
- M.I. Alfonso and F. Barber, 'Combinación de Procesos de Clausura y CSP para la Resolución de Problemas de Scheduling', *Proceedings of the VIII Conferencia de la Asociación Española para la Inteligencia Artificial*, 1(3), 35-42 (1999).
- P. Baptiste and C. Le Pape, 'A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling', *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 600-606, Morgan Kaufmann, (1995).
- F. Barber, 'Reasoning on complex disjunctive temporal constraints', *Journal of Artificial Intelligence Research*, (2000).
- R. Dechter, I. Meiri, and J. Pearl, 'Temporal constraint networks', *Artificial Intelligence* 49, 61-95 (1991).
- L.R. Ford, D.R. Fulkerson, 'Flow in Network', *Princeton University Press*, (1974).
- E.C. Freuder, 'A sufficient condition for backtrack-free search', *Journal of ACM* 29(1), 24-32, (1982).
- A. Garrido, E. Marzal, L. Sebastián and F.Barber, 'Un Modelo de Integración de Planificación y Scheduling', *Proceeding of CAEPIA'99* 1(3):1-9, (1999).
- D. Goldfarb, 'Shortest Path Algorithm Using Dynamic Breadth-First Search', *Network*, 21, 29-50 (1991).
- R. Ravi, V. Madhav, and C. Pandu, 'An Optimal Algorithm to Solve the All-Pair Shortest Path Problem on Interval Graphs', *Network*, 22, 21-35 (1992).
- D.R. Shier, 'On Algorithm for Finding the K -Shortest Paths in a Network', *Network*, 9, 195-214 (1979).