Managing Efficiently Temporal Relations Through Indexed Spanning Trees

Malik GHALLAB and **Amine MOUNIR ALAOUI**
Laboratoire d'Automatique et d'Analyse des Systèmes du CNRS
7, avenue du Colonel Roche, 31077 Toulouse, France

Abstract

We are interested here in the design of a very efficient Time-Map Manager, able to deal with a large knowledge-base of several thousand time-tokens in demanding applications such as reactive planning and execution control. A system, called *IxTeT*, aiming at that goal is described. It uses an original representation of a lattice of time-points that relies for efficiency on a maximum spanning tree of the lattice together with a particular indexing of its nodes.

The *IxTeT* system is sound and complete, it has the same expressive power as the restricted Interval Algebra that permits completeness in polynomial time. Its average complexity is shown experimentally to be linear, with a low overhead constant, for both operations: retrieval in and updating of a set of temporal relations.

1. Introduction

Temporal knowledge is essential in several AI tasks, such as planning and reasoning on actions, on past and future events, and on dynamic situations. Temporal knowledge requires a specific representation if one wants to correctly grasp the notion of causality. As for other types of knowledge, this representation should be consistent and complete, and should provide a good trade-off between expressive power and complexity. The efficiency criteria is very important for real-time applications, such as those of reactive planning and execution control that motivated this work [Ghallab], where time is an object of representation and reasoning as much as a constraint.

Acknowledgement: This work was supported by the EEC under the ESPRIT project P1560 (SKIDS) and by the French National Research Program GRECO/PRC-IA

A particular set of representations were found to have some nice properties: *reified logic* approaches proposed in [McDermott] and [Allen 84] and formally studied in [Shoham] and [Tsang]. The common feature of these representations is the use of couples: <logical formula Φ, temporal qualification of Φ>. They require a so-called Time-Map Manager (TMM) that is in charge of retrieval in and updating of a knowledge base of temporal relations. In a typical application, a TMM will be put at a very low level and in heavy use, it has to be very efficient.

Of the above set of representations, the Algebra of Temporal Intervals [Allen 83] is the most popular representation: it is appealing for its expressive power and ease of implementation. It has however a major drawback: the consistency problem for a set of Interval Algebra relations is NP-complete [Vilain]. But of resorting to exponential algorithms, this leads to the use of a transitive closure propagation algorithm that is defective because of:

- a completeness problem: it may accept an inconsistent set of relations as being consistent; and
- a complexity problem: it runs in $O(n^3)$, a too high complexity for a real application.

As it was argued in [Vilain] one can solve the completeness problem by restricting the expressive power to a sub-class of Interval Algebra, that class equivalent to the Time Point Algebra. For those two representations the propagation algorithm is complete.

This paper addresses the complexity problem. It describes a representation and a set of algorithms for a TMM, called Indexed Time Table (IxTeT), that has the following properties:

- it relies on a particular Time Point Algebra that has the same expressive power as the restricted sub-class of [Vilain],
- it is guaranteed to be complete,
- it permits non-monotonic updatings of the knowledge base, *i.e.* both addition and removal of relations, and
- it leads to a very efficient TMM that is shown

experimentally to be of linear time and space complexity for both operations, retrieval and updating.

The proposed representation and approach are described in the next section. Main algorithms are detailed and discussed in section 3. Extensive empirical results, that characterize the performances of the proposed TMM and permit to favorably compare it to those described in the litterature, are summarized in section 4. Some extensions to this TMM for dealing with numerical constraints are finally considered.

2. Representation

2.1. Approach

The role of a TMM in a temporal reasoning system is the achievement of the following two tasks:

(i) retrieval: find whether two events in the knowledge base are temporally related and how, and

(ii) updating: add or remove events and temporal relations to the knowledge base while maintaining its consistency.

As for other types of binary constraints, a natural representation for a set of temporal relations is a network where nodes are time tokens (*i.e.* intervals or instants) and arcs are labeled by the constraints relating two nodes. Two directions can be pursued:

- either using a complete graph where all possible relations between all pairs of nodes are propagated and explicitly maintained: this makes task (i) trivial in O(1), and requires a costly propagation algorithm in $O(n^3)$ for monotonic updatings;
- or using a network where the only arcs are those of the explicit knowledge of the problem: this simplifies task (ii) but requires for task (i) a costly search through possible pathes of the network.

The approach proposed here is a trade-off between these two directions. It relies on the efficient combination of 2 principles:

- adding to the time-network a particular data structure, a maximal spanning tree with an adequate indexing scheme, that permits a very efficient computation of ancestral information, this greatly simplifies task (i), and
- restricting the propagation of new relations to a small subset of nodes in the network in order to perform task (ii) efficiently.

2.2. Time lattice

Of the two possible and equivalent representations that lead to a tractable polynomial problem, *i.e.* restricted interval algebra and time-point algebra, we choose the second one. Although this choice was motivated by efficiency considerations, it has other advantages:

- there is no simple characterization of the subset of relations that falls into the restricted interval algebra: input data has to be compared to the complete list of 187 such relations compiled in [Granier]); but, for the user's convenience, one may provide intervals and some useful relations as primitives of the input language and translate them internally into time-points relations;
- the particular intervals and relations that are responsible for an inconsistency are difficult to locate even when the propagation algorithm detects the inconsistency; this is straightforward in the proposed representation;
- numerical constraints are more easily taken into account by the time-point representation;
- it is easy to represent open intervals such as properties that become true at a known moment and remain until a contradiction is found ("persistance" in [Dean] ontology).

In a time-point algebra 3 elementary relations, *before*, *equal* and *after*, and their 5 disjunctive combinations relate a finite set of instants or time-points. Instead of a network with arcs labeled by relations, we use 2 different types of **unlabeled** arcs:

- arc $\langle$ standing for the relation (*before* or *equal*), and
- arc $\neq$ meaning the relation (*before* or *after*).

The 8 possible relations between 2 time-points are easily expressed as 0, 1 or 2 arcs relating two nodes.

A network of $\langle$ and $\neq$ arcs corresponds to a consistent set of relations if no pair of nodes, connected by a $\neq$ arc, are involved in a loop through $\langle$ arcs. Such a loop describes a set of identical time-points that should be collapsed to a single node. Individual events corresponding to this set are kept distinct but their simultaneity is recorded by connecting all of them to the same node in the time-map. Arcs $\neq$ do not require any propagation mechanism; they are looked for only when a collapsing decision has to be taken. For that reason we can keep arcs $\neq$ implicit in the network representation.

A consistent network where all possible collapsing operations have been performed contains only $\langle$ arcs and is loop-free. It thus defines a partial order over the set of nodes. Since we can always add for convenience an origin time-point, we endup finally with a network that is a rooted DAG, *i.e.* a time-lattice. Let us denote it L=(U,A) where U= $\{t_0, u, v, w, ...\}$ is the set of time-points, t_0 is the root of L; and A is the set of $\langle$ arcs in L.

Point *u* precedes temporally (is *before* or *equal*) point *v* if there is a path in L going from *u* to *v*. Let us denote u $\langle\langle$ v this fact ($\langle\langle$ is the transitive closure of $\langle$). Thus relating 2 points requires a search of a path in L . How can we speed-up such a search?

2.3. Indexed Spanning Tree of the Time-Lattice

It is well known that ancestral information can be com-

puted in constant time for a tree correctly ordered, *e.g.* by preoder and postorder. To use such property 2 problems should be addressed: (a) how to map a time-lattice to a tree, and (b) classical orderings are not easily updated and maintained for a dynamically growing tree .

We solve the first problem by extracting from L a maximum spanning tree T defined as follow:

- T is rooted at t_0 and covers all nodes of L (it is not a free tree as is usually the case for a spanning tree);
- T contains a maximal number of arcs.

Let us denote by $r(u)$ the rank of u in L, *i.e.* the length of the longest path in L from t_0 to u:

$r(u)= 1 + \max\{r(v) \mid \forall (v,u) \in A\}$, with $r(t_0)=0$.

To compute T from L we first order the nodes in L according to their rank. This can be achieved by an O(|A|) breadth first search in L: all successors of nodes of rank k have their rank set to k+1, which may change previously computed ranks if longer pathes are found (some simple additional tests speed-up the procedure).

Let M be the maximal rank found in L. We start from any node z of rank M, put it in T, choose among its predecessors in L any node y of rank M-1 and put in T the arc (y,z) and the node y as the *parent* node of z: $p(z)=y$. This is repeated for a predecessor x of y such as $r(x)=r(y)-1$; arc (x,y) and node x are added to T. We keep on moving up along a path of maximal length until the root t_0 is put in T.

The procedure is repeated starting from a node of maximal rank among those not already in T. While processing node u, if there is a choice between several of its predecessors, all at rank $r(u)$-1, we choose one not already in T and add it to T. If none remains, we choose among such predecessors one in T which has the least number of children in T, and attach a new path to the spanning tree. The procedure is repeated until all nodes of L are put in T.

The generated tree has indeed a maximal number of arcs: in a spanning tree of L rooted at t_0 there is one and just one path from t_0 to each node of L. Each such path in T goes through a maximal number of arcs.

Let $s(u)$ be the set of *children* of u in T, and $\mathcal{S}(u)$ the set of its *descendants* in T (transitive closure of s). Our goal is to be able to compute as efficiently as possible ancestral information in T, *i.e.* whether $v \in \mathcal{S}(u)$. This is achieved through a particular indexing of T.

To each node u is attached as index a sequence: $I(u)=[i_1.\ i_2.\ \dots\ i_k]$ of one or more integers that is defined, while generating T, as follows:

- nodes of the path (t_0, ..., x, y, z) that was first put in T are indexed by their rank: $I(z)=[M]$, $I(y)=[M-1]$, $I(x)=[M-2]$, ... , $I(t_0)=[0]$;
- if $I(u)=[i_1.\ i_2.\ \dots\ i_k]$ and $v \in s(u)$ then:

  if $|s(u)|=1$ (v is the first *children* of u in T) then $I(v)=[i_1.\ i_2.\ \dots\ i_{k-1}.\ (i_k+1)]$

  if $|s(u)|=2$ then $I(v)=[i_1.\ i_2.\ \dots\ i_k.\ 1]$

  if $|s(u)|>2$ then $I(v)=[i_1.\ i_2.\ \dots\ i_k.\ (3 - |s(u)|).\ 1]$

Nodes of the first path put in T can be indexed while they are added to T. The other nodes are not indexed until their path is attached to T: indexing proceeds by moving from the attachment *parent* (already indexed) down along the path.

The addition or removal of a new node, as a leaf in the tree is straightforward; for an internal node, only some of its *descendants* will have to be reindexed. Problem (b) is thus correctly addressed.

An example will illustrate the complete procedure: figure 1.a gives a lattice L with the rank of each node shown. A corresponding maximal spanning tree is drawn in figure 1.b where the index of each node follows its label (some of the arcs of L that do not belong to T are shown in broken lines)

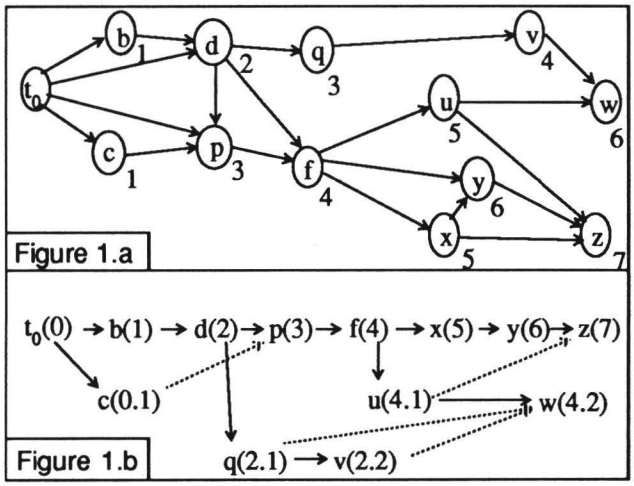

Figure 1.a

Figure 1.b

The main property of this indexing scheme is the following: for $I(u)=[i_1.\ i_2.\ \dots\ i_k]$, and $I(v)=[j_1.\ j_2.\ \dots\ j_h]$

$v \in \mathcal{S}(u)$ iff

$$k \le h \wedge [i_1.\ i_2.\ \dots\ i_{k-1}]=[j_1.\ j_2.\ \dots\ j_{k-1}] \wedge i_k \le j_k \quad (1)$$

Thus, to relate 2 nodes u and v in T we first compare their rank

- if $r(u)=r(v)$ then u and v are not related in T;
- if $r(u) < r(v)$: either condition (1) is satisfied: v is a descendant of u, or they are not related;
- if $r(u) > r(v)$: u and v are permuted before checking condition (1).

Notice that the rank of a node is given by its index :

$$r(u)= \sum_{\substack{i=1 \\ u_i > 0}}^{k} u_i \quad \text{with } I(u)=[u_1.\ u_2.\ \dots\ u_k\ ]$$

In the popular Dewey decimal notation k, the length of an index, grows while we go deeper in the tree,

whereas in the proposed indexing scheme we try as much as possible to keep this length small and make the value of i_k, the last element of an index, increases with the depth. This is because condition (1) is checked in O(k), the length of the smallest index.

In the worst case, all pathes will be attached to the root node: the maximum value of *k* can be proportional to the number of nodes at maximal rank in L. An unfavorable case is that of a balanced tree where $k \leq \log(n+1)$, *n* being the total number of nodes (if T is binary there are $n=2^{M+1}-1$ nodes, those at the maximum rank M are indexed [M], [M-1 . 1], [M-2 . 2], [M-2 .1 .1], ... [0.1. ... .1]). However the proposed procedure maximizes the depth of T while minimizing heuristically its breadth (by chosing as a *parent* a fatherless node or one with the least number of *children* in T). This keeps the average value of *k* smaller than the log(n) case.

Let us remark that the complete procedure for extracting and indexing the maximal spanning tree, while efficient in O(|A|+n), is not used by the TMM but at initialization time.

2.4. Residue of the Time-Lattice

A precedence relation in L, such as u ⟨⟨ v , can be retrieved either

- through the spanning tree T if $v \in \mathcal{S}(u)$: this is checked easily; or
- through a path using some arcs of L not belonging to T.

To take care of this last case, residue arcs are processed by 2 operations:

- Eliminating redundant arcs: if $(u,v) \in A$ is an arc of L not belonging to T such as $v \in \mathcal{S}(u)$ then this arc does not bring any useful information. It is redundant and can be eliminated. In figure 1 arcs (t_0,d), (t_0,p), (d,f), (f,y), (x,z) are redundant.
- Propagating non redundant arcs: 2 nodes may be linked by a path that mixes in any order arcs from T and non redundant residue arcs. To avoid mixing the 2 structures and simplify the retrieval of precedence relations we propagate recursively a non redundant arc (u,v) to the parent node of *u* in T, unless v is already a successor of p(u) in T, *i.e.* if $v \notin \mathcal{S}(p(u))$ then an arc from p(u) to *v* is added. This is the case of arc (q,w) in figure 1.b.

The exact procedure is described in section 3.2. It corresponds to the trade-off mentioned earlier between using a complete graph and keeping a minimal set of relations.

Let us call *residue arcs* the set obtained after elimination and propagation.

The important property here is that any *residue arc* (u,v) is such that r(u) < r(v). This is trivially true for arcs in L not belonging to T, it is also true for added arcs since the propagation goes only upward in T.

3. General algorithms

The procedures required for the 2 tasks of a TMM, retrieval of precedence relations and updating the time-map, using the proposed representation, are formally described in this section. Let us first summarize our main notations, and make clear the distinction between the part of the time-lattice L covered by the spanning tree T and the residue part.

In T we speak of the *parent* p(u) of a node u, its *children* s(u), its *descendants* $\mathcal{S}(u)$ and its *ascendants*; I(u) is its index and r(u) its rank. Only *s*, *p* and *I* are kept as data structures.

In the residue part, a(u) denotes the set of nodes linked to u by *residue arcs* . We will speak of the *followers* and *foregoers* of a node for adjacency relations defined by such arcs.

We reserve *successors* and *predecessors* for the complete lattice. Notice that $s(u) \cap a(u) = \emptyset$: the *successors* of a node are partitioned into its *children* and its *followers*.

3.1. Retrieval: relating 2 points

The proposed representation permits to characterize a precedence relation by the following conditions:

$$u \langle\langle v \Leftrightarrow (r(u) < r(v)) \wedge [(v \in \mathcal{S}(u)) \vee (v \in a(u)) \vee (\exists\, w \in a(u) \mid w \langle\langle v)]$$

The 3 first conditions come from the fact that *u* precedes *v* in L if v is a *descendant* of u in T or if it is a *follower* of *u*. The last one is due to the property of the upward propagation mechanism:

$\forall\, v \in \mathcal{S}(u), \forall\, w \in a(v) : w \in a(u) \vee w \in \mathcal{S}(u)$

all *followers* of a *descendant* of *u* are either the *followers* of *u* or its *descendants*; there cannot be a non *descendant* node *w* of *u* linked to *u* through *followers* of a node *v* in $\mathcal{S}(u)$, that is not also linked to *u* through a(u).

The comparison algorithm between two points *u* and *v* is thus:

```
Compare(u,v)
•   if r(u)=r(v) then return(nil)
    else   if r(u) > r(v) then Relate(v,u)
           else Relate(u,v)
```

```
Relate(u,v)
• if v ∈ S(u) then return(u ⟨⟨ v)
• if v ∈ a(u) then return(u ⟨⟨ v)
• else
  • for each w ∈ a(u)
    • if [r(w) < r(v) and Relate(w,v) returns (w⟨⟨v)]
      then return(u⟨⟨v)
  • return(nil)
```

Notice that the recursive calls to **Relate** are pruned

when the rank of w reaches that of v

3.2. Updating: Adding new points and relations

Updating should be done such as to keep all the properties of the representation. Let us first focus on the addition of points and relations in L (removal is considered in section 3.4). The additon of a point w and 2 relations $u \langle w \langle v$ can be decomposed into 2 steps:

- add w as a *child* of u and give it the right index ; and
- add an arc between w and v and update the tree and residue arc if necessary.

The first operation is straighforward, is is achieved by procedure **Addpoint** bellow. The second operation involves 3 steps: a test, and eventually a propagation and a reindexation.

The test determines whether $v \langle\langle w$, in this case, unless all points in every path from v to u can be collapsed, the updating is impossible: $w \langle v$ cannot be inserted, it is inconsistent with the arcs in pathes between v to w. Inconsistent arcs are returned as output for further analysis.

The reindexation takes place if $r(w) \geq r(v)$. In this case, to keep v on the longest path in the spanning tree, we give to v a new *parent* node w. Node v is removed from the *children* of $p(v)$ and put as a follower of $p(v)$; w becomes the new *parent* node of v; the *descendants* of v are reindexed. The reindexation algorithm computes the new index of each node according to the index of v, it then verifies if the *followers* of v have the right rank considering the new index of v, and, if not, it reindexes them. This is repeated recursively.

If $r(w) < r(v)$, v is put as a *follower* of w. This *residue arc* is propagated to the *parent* of w, and recursively to its *ascendants* that are not found by **Relate** linked to v. Notice that if there is a reindexation, there will be propagation of the arc between the old *parent* of v and v: all the former *ascendants* of v have to know that they are still linked to it.

```
Addrelation(w,v)     ;Add the relation w ⟨ v
• if v ⟨⟨ w then
    • if no pair of nodes in a path from v to w are
      distinct then Collapse(v,w)
      else return(Inconsistency)
  else if w ⟨⟨ v then return(already true)
      else if r(w) ≥ r(v) then
          • let x ← p(v)
          • add v to a(x) and remove v from s(x)
          • add v to s(w) and Reindex(v)
          • Propagate(x, v)
          • For each x'∈ a(v) Propagate(v,x')
          • p(v) ← w
        else
          • add v to a(w) and Propagate(w,v)
```

```
Addpoint(w,u)        ;Add a point w after the point u
•   add w to s(u)
•   calculate index of w (from that of p(w))
•   p(w) ← u
```

The **reindex** function reindexes all the *descendants* of v according to $I(v)$. It then verifies that the ranks of the *followers* of v are not affected by the new index, if they are, they will be reindexed too. The algorithm is:

```
Reindex(v)
•   calculate index of v (from that of p(v))
•   for each element x of s(v) Reindex(x)
•   for each element x' of a(v)
      • if [ r(v) > r(x') or [ r(v)=r(x')
                  and |s(v)| < |s(x)| ]] then
          • let y ← p(x')
          • add x' to a(y)
          • remove x' from s(y)
          • p(x') ← v
          • add x' to s(v)
          • Reindex(x')
          • For each x"∈ a(x') Propagate(x',x")
          • Propagate(y,x')
```

As we saw before, residue arcs have to be propagated to the *ascendants* of u to keep a link between them and v if there is none. To propagate the relation $u \langle v$, the algorithm will simply be:

until $(v \in \mathcal{S}(p(u)))$ or $(\exists\, w \in a(p(u)) \mid w \langle\langle v)$ **do**:

add v in $a(p(u))$, $u \leftarrow p(u)$.

But in a special case, this algorithm stops before the propagation is complete: in the example of figure 2,

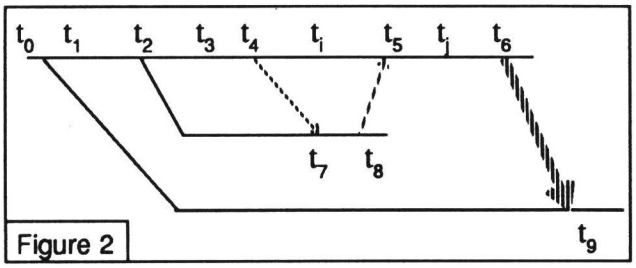

Figure 2

propagation of the relation $t_6 \langle t_9$ will go on until t_4. There we have $t_8 \in a(t_4)$, and $t_8 \langle\langle t_9$. If we stop the algorithm, the propagation will be incomplete because we will not be able to **Relate** t_2 and t_9: $t_9 \notin \mathcal{S}(t_2)$, $t_9 \notin a(t_2)$, and no element of $a(t_2)$ has a link with t_9. The propagation was stopped at t_4 because there are two pathes from it to t_9: one direct through t_6 ($t_6 \in \mathcal{S}(t_4)$) and the other one through t_7. In this case we have to continue the propagation to be sure that all *ascendants* of t_6 are really **Related** to t_9. The relation is propagated to

all the *ascendants* until t_0 while avoiding it for *ascendants* already related to the destination node. The algorithm proposed is then:

```
Propagate(u,v)                                  ; u ⟨ v
1•   while [(p(u) ≠ t0) and (r(p(u))<r(v))]
2•     while Relate(p(u),v) returns nil
         • add v to a(p(u))
         • u ← p(u)
3•     if [(p(u) ⟨⟨ v) ∧ (∃ w ∈ a(p(u)) | w ⟨⟨ v)
       ∧ (∃ w' ∈ a(p(u)) | w' ⟨⟨ v)] then
         • while [Relate(p(u),v) returns (p(u) ⟨⟨ v)
                 and p(u) ≠ t0)]
         • u ← p(u)
       else return(finished)
```

The third step in this algorithm tests if there is a path from p(*u*) to *v* through *residue arcs* that contains a descendant of p(*u*). In this case we are in the conditions described previously. This test is described only in a mathematical formalism because its efficient implementation needs special care and some modifications in the **Relate** procedure that are of less interest here.

This algorithm runs a maximum of r(*u*) comparisons in the worst case, but in fact, in many cases, it will stop sooner: the algorithm stops as soon as p(*u*) is linked to *v*, this may happen very quickly, and then, the algorithm continues only if there is more than one path that links p(*u*) to *v* and if one of those pathes contains a descendant of p(*u*). This is a very strong condition that may seldom occur.

3.3. Collapsing 2 points

When the points of a loop can be made identical (*i.e.* there is no ≠ arc between them), they are collapsed to one point. This is done by keeping the point with the highest rank and linking it through all the relations handled by the other points that are removed. This involves reindexing their *successors*, and modifying arcs and branches issued from their *predecessors*.

3.4. Removing points and relations

IxTeT enables removal of points and relations if needed. Removing a point *u* requires 2 operations: putting p(*u*) as *parent* of all the *children* of *u* and reindexing its *descendants*, and then, redirecting all the residue arcs pointing to *u* and to *successors* of *u*.

For removing a relation that is a branch of the tree from *u* to *v*, we have to find the *predecessor* of *v* with the highest rank, and put it as p(*v*). If no *predecessor* remains, t_0 is taken. Then, reindexation starts at *v*.

If the relation removed is a residue arc from *u* to *v*, *v* is just removed from a(*u*), this has to be propagated to the *ascendants* of *u* until one of them is related to *v* by another path than the one that contains *u*.

Notice that the reindexation used here is slightly different from **Reindex**: each step verifies if there is no *predecessor* with a higher rank than the new one, and in this case, changes the *parent*.

4. Experimental results

All the algorithms described in the previous section were implemented (on a Sun 4-260 in Sun Common Lisp), and tested [Mounir]. The workload for insertion of points and arcs in the lattice, and for retrieval or removal of relations between points were recorded.

Tests were made on 50 lattices, randomly generated, and growing in size up to 2000 points. Each lattice was generated by adding at each step one point and k relations (algorithms **Addpoint** and **Addrelation**). We experimented on data for k=2 to 5, without noticing a significant change in performances. Average running times increased very slightly from k=2 to 3, then decreased for k=4 and 5. Results reported are for k=2. Figure 3 plots 20 average times, and 2 standard deviations, for retrieving the relation between 2 random nodes. Each point is averaged over 2500 runs: 50 retrievals in 50 lattices.

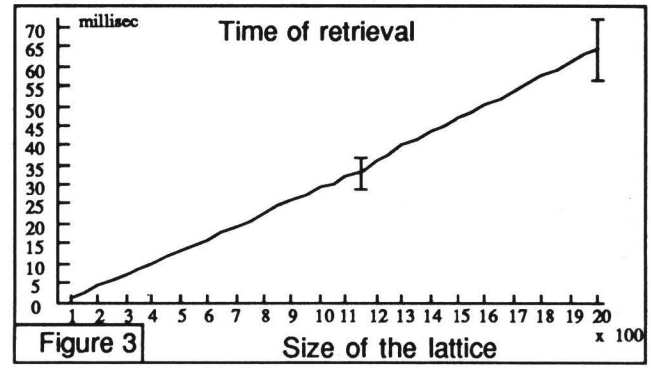

Figure 3

The same information concerning the insertion of one relation (procedure **Addrelation**) is given in figure 4.

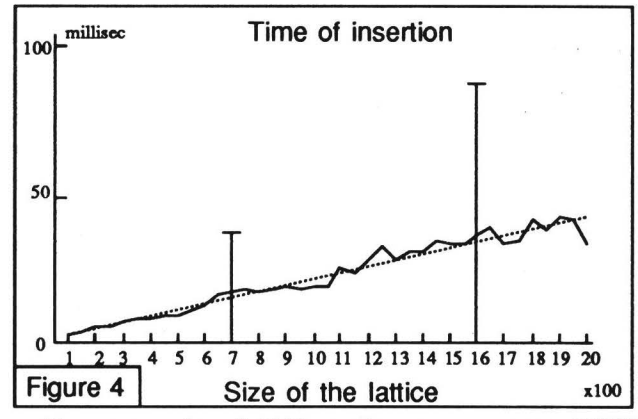

Figure 4

To summarize IxTeT performances, over 100000 runs of updatings and retrievals support the following evidences:

- the proposed TMM seems to be in average of linear complexity for both retrieval and insertion of relations; due to reindexation, the standard deviation is however higher for insertion; indeed, if a relation is added near the root of the lattice, the part of the lattice to be reindexed may remain large despite the pruning heuristics, that are efficient only in the average;
- the linearity constant is very low: our non optimized Lisp implementation required:
  - about .1 second to insert a new relation in a 2000 points lattice
  - less than 70 millisecond to compare 2 points in a 2000 points lattice
- the space complexity of IxTeT seems also to grow linearly; a 2000 points lattice required less than 20000 memory units (cons cells).

In parallel with these experimentations, the propagation algorithm of [Allen 83] was implemented. However we could not manage graphs larger than 80 intervals (less than 160 points): this algorithm is in $O(n^2)$ space. Insertion in a graph of 50 intervals required about 4 to 5 seconds. Since insertion is in $O(n^3)$ time, it would have a prohibitive cost for much larger graphs.

Experimental results of Allen's algorithms are reported in [Koomen] without giving running time. To give an idea, a graph of 101 intervals and 450 relations is processed by TimeLogic through 11×10^5 calls to the basic constraint propagation function (performs some table lookups and unions of fixed length lists). If the graph is hierarchized 4.5×10^5 calls to this function are required. To our knowledge no work reported the application of the transitive propagation algorithm for a graph of several hundred nodes.

5. Conclusion

IxTeT abilities for dealing with symbolic temporal knowledge can be extended to numerical quantitative constraints and to constant points (dates). All numeric relations and dates may be given with intervals precising earliest and latest possible occurrence or simply errors foreseen. Indeed it is easy to have in IxTeT an absolute referencing system by dates with qualitative relations. A point can be attached to a date. Arcs can be labeled by durations. The direct link that exists between durations and dates allows several deductions on the partial order of points. A lattice may have some nodes known as precise dates, others are variables linked with qualitative relations, and others have dates deduced from quantitative relations.

In conclusion this paper presented an original approach for designing a TMM that has some interesting properties:

- it relies on a Time-Point Algebra that gives the same expressive power as the restricted Interval Algebra which permits completeness in polynomial time;
- it makes use efficiently of a maximum spanning tree of the lattice together with a particular indexing scheme for checking in constant time ancestral information and for reducing and pruning the propagation and search; this data structure gives a precise and dynamic hierarchy for the temporal relations;
- it is shown experimentally to be of linear space and time complexity for both operations, retrieval and updating, thus bringing a significant improvement over known algorithm;
- it permits removal of points and relations from the time-map;
- it can deal with numerical knowledge attached to time.

References

Allen, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832-843, November 1983

Allen, J. F. Towards a general theory of action and time. *Artificial intelllience 23: 123-154*, 1984

Dean, T.L. and McDermott, D. V. Temporal database management. *Artificial intelligence 32: 1-55*, 1987

Ghallab, M., Alami, R. and Chatila, R.Dealing with time in planning and execution monitoring. *Robotics Research 4, R. Bolles, MIT Press*, 1988

Granier, T. Contribution à l'étude du temps objectif dans le raisonnement. *Rapport LIFIA RR 716-I-73, Grenoble*, February 1988

Koomen, J. A. G. M. The TIMELOGIC temporal reasoning system in common lisp. *Technical report TR 231, Rochester University*, November 1987

McDermott, D. V. A temporal logic for reasonning about processes and plans. *Cognitive Sci. 6:101-155*, 1982

Mounir Alaoui, A. IxTeT: un système de gestion de treillis d'instants. *Rapport LAAS 88.154, Laboratoire d'Automatique et d'Analyse des Systèmes / CNRS, Toulouse*, June 1988

Shoham, Y. Temporal logics in AI: semantical and ontological considerations. *Artificial intelligence 33: 89-104*, 1987

Tsang, E. Time structure for AI. in *Proceedings of the tenth IJCAI: 456-461*, 1987

Vilain, M. and Kautz, H. Constraint propagation algorithms for temporal reasoning. in *Proceedings of the fifth national conference on artificial intelligence (AAAI-86)*:377-382, August 1986