Efficient Temporal Reasoning through Timegraphs *

Alfonso Gerevini
IRST - Istituto per la Ricerca
Scientifica e Tecnologica
38050 Povo TN, Italy
gerevini@irst.it

Lenhart Schubert
Computer Science Department
University of Rochester
Rochester, NY 14627, USA
schubert@cs.rochester.edu

Abstract

In this paper we address the problem of scalability in temporal reasoning. In particular, new algorithms for efficiently managing large sets of relations in the Point Algebra are provided. Our representation of time is based on *timegraphs,* graphs partitioned into a set of chains on which the search is supported by a rnetagraph data structure. The approach is an extension of the time representation proposed by Schubert, Taugher and Miller in the context of story comprehension. The algorithms presented in this work concern the construction of a timegraph from a given set of relations and are implemented in a temporal reasoning system called *TG-II.* Experimental results show that our approach is very efficient, especially when the given relations admit representation as a collection of chains connected by relatively few cross-chain links.

1 Introduction

Temporal reasoning is an important task in many areas of AI, including planning, natural language processing and expert systems. In several applications, temporal knowledge may be expressed in terms of collections of binary relations between intervals or points. Temporal reasoning tasks include determining the consistency (satisfiability) of such collections, finding a consistent scenario (an interpretation for all the temporal variables involved) and deducing new relations from those that are known (or computing their closure). Since Allen's work on binary interval relations [Allen, 1983], numerous researchers have addressed temporal reasoning in terms of constraint propagation techniques [Ladkin and Maddux, 1988; Vilain *et aL,* 1990; van Beek and Cohen, 1990; Dechter *ei aL,* 1991]. The main problem with these techniques is their scalability. The problems of determining the satisfiability and of computing the closure of a set of assertions in Allen's interval algebra (IA) are NP-complete [Vilain *et aL,* 1990]. Restricting IA to the "pointizable" IA (P1A), i.e. to the set of relations in IA that can be translated into conjunctions of point relations between the endpoints of the intervals, makes these problems tractable. Algorithms have been developed for PIA running in $0(n^2)$ space and $0(n^2)$ time for determining consistency [van Beek, 1990] and $0(n^4)$ time for computing the closure [van Beek and Cohen, 1990] (for n points). Unfortunately these bounds are still unacceptable for domains in which a large data-base of relations needs to be managed [Allen and Schubert, 1991]. Recently, other approaches based on graph algorithms have been proposed whose main characteristic is that of providing better performance in practice compared to the more traditional constraint-based approaches [Miller and Schubert, 1990; Ghallab and Mounir Alaoui, 1989; Dorn, 1992; Golumbic and Shamir, 1992].

We are interested in the problem of efficiently managing large data sets of qualitative temporal relations in the Point Algebra (PA) [Vilain *el aL,* 1990]. The elements of PA are the set of relations {<, >, =, > ,>, ≠, <>, 0}, through which all the relations of the PIA can be represented [Ladkin and Maddux, 1988]. Our approach is based on ideas derived from a temporal reasoning system developed in the context of natural language comprehension [Schubert *et aL,* 1987; Miller and Schubert, 1990]. In this system temporal relations are represented through graphs, called timegraphs, whose vertices represent points and whose edges represent temporal relations. The main characteristics of timegraphs are their partitioning into a set of chains, which are sets of linearly ordered points, and their use of a *rnetagraph* structure to guide the search processes across the chains. One advantage of this approach is that the space complexity can be linear in the number of stipulated relations. The other advantage is the efficiency in terms of computation time in domains such as planning [Allen *et aL,* 1991] and story understanding in which the temporal data tend to fall naturally into chain-like aggregates [Miller and Schubert, 1990]. Essentially this is because the "worlds" described in stories and plans consist of individuals (or sets of related individuals) each moving through a course of actions and events, creating a trajectory in time. Building timegraphs in practice takes much less time than computing closure (the minima/ *network* in constraint propagation terminology), and the amount of time spent in querying relations is nearly constant.

In this paper we present new algorithms for building

*The authors thank James Allen for drawing their attention to the problem of scalability and for many fruitful discussions. The work of the first author was carried out in part in the context of the MAIA project at IRST, with the support of Oliviero Stock, and of CNR PF Sistemi Informatici e Calcolo Parallelo. The second author was supported by Rome Lab contract F30602-91-C-0010.

timegraphs from sets of relations in the Point Algebra. Among them there is an algorithm for dealing efficiently with "not equal" relations ($\neq$) which were not considered in previous timegraph algorithms. To our knowledge, the only authors who have proposed algorithms for reasoning efficiently with $\neq$ relations are Ghallab and Mounir Alaoui [1989] and van Beek and Cohen [1990]. Neither has treated the problem quite satisfactorily. In fact, the correctness of the algorithm proposed by van Beek [1990] is based on a lemma whose proof given in [van Beek and Cohen, 1990], we show to be incorrect. However, we provide a new proof for the lemma which shows that van Beek's algorithm is indeed correct. In [Ghallab and Mounir Alaoui, 1989] the $\neq$ relations are only partially treated as the proposed algorithms cannot derive some strict orderings induced by $\neq$ relations.

Section 2 introduces our framework formally; Section 3 shows the algorithms for building a timegraph and for querying relations; Section 4 reports experimental results we have from *TG-II*, a system in which the algorithms described in the previous sections have been implemented; Section 5 gives conclusions and future work.

2 Representing temporal relations through graphs

In this section we introduce the definitions and theorems on which the proposed approach is based. A path of length n is a sequence of n triples $(v_0, l_1, v_1), ..., (v_{n-1}, l_n, v_n)$ where v_i $(0 \leq i \leq n)$ are vertices and l_j $(1 \leq j \leq n)$ are labels (relations) on edges.

Definition 2.1 A *temporally labeled graph* (*TL-graph*) is a graph with at least one vertex and a set of labeled edges, where each edge (v, l, w) connects a pair of distinct vertices v, w. The edges are either directed and labeled $\leq$ or $<$, or undirected and labeled $\neq$.

We assume that every vertex of a TL-graph has at least one name attached to it. If a vertex has more than one name, then they are alternative names for the same time point. The name sets of any two vertices are required to be disjoint.

Definition 2.2 In a TL-graph we call a path a *$\leq$-path* if each label l_i is $\leq$ or $<$. A $\leq$-path is a *$<$-path* if at least one of these labels is $<$.

Definition 2.3 A $\leq$-path ($<$-path) of length n, $n \geq 1$, is a *$\leq$-cycle* (*$<$-cycle*) if $v_0 = v_n$. A TL-graph is *acyclic* if it does not contain any $\leq$-cycle.

Definition 2.4 A TL-graph is *consistent* if its vertices can be interpreted as elements of a totally ordered set (with ordering $<$) satisfying the constraints imposed by the edges.

Theorem 2.1 *A TL-graph is consistent iff it does not contain any $<$-cycle, or any $\leq$-cycle that has two vertices connected by an edge with label $\neq$.*

Proof (sketch): The "only if" direction is obvious from irreflexivity of $<$, and from equality of vertices on a $\leq$-cycle. The "if" direction is proved by induction on the number of $\neq$-edges. When there are none, the absence of $<$-cycles guarantees consistency. In the induction step, we note that we can consistently add $(v, \neq, w)$ just in case we can consistently add $(v, <, w)$ or consistently add $(w, <, v)$. We use the former if there is a $\leq$-path from v to w, and the latter otherwise, obtaining a situation covered by the induction assumption. □

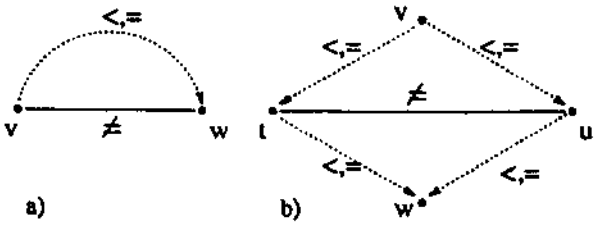

Figure 1: The two kinds of implicit $<$ relation. Dotted arrows indicate paths, solid lines $\neq$ edges. In both the graphs there is an implicit $<$ relation between v and w.

Definition 2.5 A TL-graph contains an *implicit* $<$ relation $v < w$ if there is no $<$-path from v to w and either there is an edge between v and w with label $\neq$ and a $\leq$-path (but no $<$-path) from v to w (see Figure 1.a)); or there exist two vertices s and t such that there is an edge between s and t with label $\neq$ and $\leq$-paths (but no $<$-path) from v to s, s to w, v to t, and t to w (see Figure 1.b)). The graphs isomorphic to the one given in Figure 1.b) are called $\neq$ *diamonds*.

Definition 2.6 An *explicit* graph for a given TL-graph G is an acyclic TL-graph logically equivalent to G and with no implicit $<$ relations.

Theorem 2.2 *An explicit TL-graph entails $v \leq w$ iff there is a $\leq$-path from v to w; it entails $v < w$ iff there is a $<$-path from v to w, and it entails $v \neq w$ iff there is a $<$-path from v to w or from w to v, or there is an edge $(v, \neq, w)$.*

Proof (sketch): This is easy to establish for the case where there are no $\neq$-edges. When there are $\neq$-edges, the idea is to show that if there is no $\leq$-path ($<$-path) from v to w, we can "decide" any $\neq$-edge (making it a $<$-edge in one direction or the other) without creating cycles, without creating a $\leq$-path ($<$-path) from v to w, and without forfeiting the "explicit graph" property. Thus all the $\neq$-edges can be decided, and we end up with a consistent graph that entails the original one, yet doesn't entail $v \leq w$ ($v < w$). The reason we can avoid creating a $\leq$-path ($<$-path) from v to w is that if both ways of deciding a $\neq$-edge gave such a path, then a $\neq$-diamond would have to be present (formed by v, w, and the vertices connected by the $\neq$-edge), contrary to the "explicit graph" property. And the reason we don't forfeit the "explicit graph" property is that a $<$-edge replacing a $\neq$-edge cannot create paths that support an *implicit* $<$ relation (they can only support an explicit $<$ relation). □

Remark. This theorem has two important consequences. One is that it enables us to write linear time procedures which obtain the strongest relation between two points entailed by an explicit graph (called the *minimal labelby* van Beek and Cohen [1990]).Secondly, it provides the basis for a new proof of their Lemma 1 in [1990] stating that any path-consistent nonmininmal network

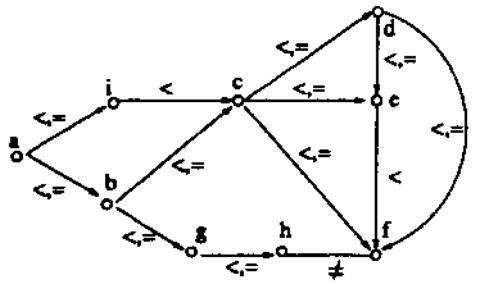

Figure 2: an example of TL-graph.

of relations [1] taken from PA must include a four-vertex subgraph isomorphic to the graph in Figure 1.b). The value of this new proof lies in the fact that the one given by van Beek and Cohen is not correct. This becomes evident if one observes that the four-vertex constraint network obtained by replacing the $\leq$-paths in Figure 1.b) with $\leq$ edges and adding the edge $(v, \{<, =\}, w)$ is not the only four-vertex *path consistent network* that, up to isomorphism, can be derived in the last step of the proof of van Beek and Cohen. In fact, the network obtained by replacing $(v, \{<, =\}, u)$ with $(v, \{<, =, >\}, u)$ and $(u, \{<, =\}, w)$ with $(u, \{<, =, >\}, w)$ is another possible network. This contradicts what van Beek and Cohen assert at the end of their proof.

We will show how linear time procedures for querying relations can be designed in section 3.5, while a new proof of van Beek's lemma is available to the interested reader in [Gerevini and Schubert, to appear] an extended version of this paper.

3 The construction of a timegraph

Given a set S of binary relations in the Point Algebra we first build a TL-graph whose vertices correspond to the variables (time points) in S, and whose edges correspond to the relations (note that $=$ relations are translated by creating only one vertex for each set of explicitly equated variables, and labelling that vertex with that set of variables). From the TL-graph we then form a *timegraph* whose definition is:

Definition 3.1 A *timegraph* is an acyclic TL-graph partitioned into a set of *time chains*, such that each vertex is on one and only one time chain. A time chain is a $\leq$-path, plus possibly *transitive edges* connecting pairs of vertices on the $\leq$-path.

Distinct chains of a timegraph can be connected by *cross-edges* (these, and certain auxiliary edges, will also be called *metaedges*). Vertices connected by cross-edges are called *cross-connected vertices* (or *metavertices*[2]). The construction of a timegraph from a TL-graph consists of four main steps: consistency checking, ranking of the graph, formation of the chains and making explicit implicit $<$ relations.

3.1 Checking consistency

Determining the consistency of a TL-graph G=(V,E), where V is the set of vertices and E is the set of edges, can be accomplished by two steps. The first step consists of the identification of all the strongly connected components (SCC) of the TL-graph derived from G ignoring the $\neq$ relations. The second step consists of checking if any of the SCCs contains a pair of vertices connected by an edge with label $<$, or $\neq$. It can be shown from Theorem 1 that a TL-graph is consistent if and only if such a SCC does not exist. When the graph is consistent, each SCC can be collapsed into any arbitrary vertex v within that component. All the cross-component edges entering or leaving the component are transferred to v. The edges within the component can be eliminated and a supplementary set of alternative names for v is generated. It is clear from the definition of a SCC that the resulting graph is acyclic and that this resultant graph (along with the sets of alternative names) is logically equivalent to the original one. It is well known that the computation of the SCCs can be accomplished in time $O(n + e)$ (where n is the number of vertices and e the number of edges) using Tarjan's algorithm [Cormen *et al.*, 1990]. Checking the existence of $<, \neq$ edges between vertices in a SCC and collapsing each SCC into a single vertex are linear tasks in the number of the edges. It follows that the global time complexity is $O(e)$. This provides a proof of the next theorem (a similar theorem is also given by van Beek [1990]):

Theorem 3.1 *A TL-graph can be recognized as being inconsistent, or if it is consistent, collapsed to a logically equivalent acyclic TL-graph (supplemented with disjoint sets of alternative names for the vertices) in $O(e)$ time, where e is the number of edges.*

3.2 Ranking the graph

Given a TL-graph G, we define the *universal start time* of G as a special vertex with no predecessors and with successors the set of vertices in G which have no other predecessors. Each edge leaving from the universal start time has label $\leq$.

In accordance with the definition given by Ghallab and Mounir Alaoui in [1989], the *rank* of a vertex v is defined as the length of the longest $\leq$-paths from the universal start time to v, times the distance increment k. (I.e., rank increases in steps of size k along maximal-length paths.) In our terminology the rank of a vertex is also called the *pseudotime* of that vertex. The main purpose of the ranks is bounding the search at query time. In fact, using the ranks, it is often possible to obtain the strongest relation between two vertices that is entailed by the graph in constant time, i.e. without performing any search. We will show how this is possible in the next Section, while in Section 3.5 we discuss the query algorithms further. Figure 3 shows an algorithm for computing the ranks for an acyclic TL-graph based on the DAG-longest-paths algorithm reported in [Cormen *et al.*, 1990] and taking $O(n + e)$ time.

[1] A network is minimal if there is exactly one labelled edge for each pair of distinct vertices and all the labels are minimal. A *path-consistent* network is one in which for any three vertices u, v, w, any of the possible relations $\{<, >, =\}$ allowed by the label of the edge joining u and v is consistent with the labels joining u and w, and v and w.

[2] This is a slight departure from the terminology of [Schubert *et al.*, 1987; Miller and Schubert, 1990].

INPUT: A consistent TL-graph G=(V,E), an integer k.
OUTPUT: The ranks of the vertices in G.

1. topologically sort the vertices in G
   (using only $<,\leq$ edges);
2. *for each* vertex $v \in V$ *do* rank(v) := $-\infty$;
3. rank(*universal-start-time*) := 0;
4. *for each* vertex u taken in topologically sorted order
5. *do for each* vertex v such that $(u, \{<, \leq\}, v) \in E$ *do*
6. *if* rank(v) < rank(u) + k *then*
   rank(v) := rank(u) + k.

Figure 3: An algorithm for computing the ranks.

INPUT: two vertices v_1 and v_2 on the same chain.
OUTPUT: the strongest relation between v_1 and v_2.

if pseudotime(v_1) < pseudotime(v_2) *then*
  if pseudotime(nextgreater(v_1)) $\leq$ pseudotime(v_2) *then*
    return $v_1 < v_2$ *else return* $v_1 \leq v_2$
else if pseudotime(nextgreater(v_2)) $\leq$ pseudotime(v_1)
    then return $v_2 < v_1$ *else return* $v_2 \leq v_1$.

Figure 4: Algorithm for determining the relation between vertices on the same chain of a timegraph.

3.3 Forming the time chains and the metagraph

The main computational importance of time chains is that, given a pair of vertices belonging to the same chain, it is possible to compute the strongest relation entailed by the graph in constant time, while, in general, this task requires a graph search linear in the number of edges (unless of course we precompute all $0(n^2)$ minimal labels). The constant time algorithm is given in Figure 4. This result is achieved by using the pseudotimes of four vertices and an additional, possibly null, link for each vertex. Following [Miller and Schubert, 1990], we call this link the *nextgreater* link, defined in the following way:

Definition 3.2 Given a vertex v, the *nextgreater of v* (written nextgreater(v)) is the nearest successor v' of v that is on the chain of v such that $v < v'$ is entailed by the graph. If v' does not exist, then nextgreater(v) is null.

We describe how the nextgreaters can be efficiently computed in the next Section. Since vertices on the same chain support constant time queries, it is desirable to keep the number of time chains to a minimum in building the timegraph. Another desirable constraint is that the number of cross-chain edges be minimal. Figure 5 shows an algorithm for creating a set of time chains from a given TL-graph that attempts to minimize the number of cross-chain edges and that works well in practice. It can be shown that the time complexity of the algorithm is $0(n + e)$. Figure 7 shows three time chains formed by using the algorithm on the graph of Figure 2.

3.3.1 Computing the nextgreater links

The algorithm for computing the nextgreater links consists of two main steps. In the first step the nextgreater links for each vertex v are computed considering only edges connecting vertices on the same chain as v. In the second step these nextgreater links are refined by looking for cross-chain <-paths.

INPUT: a consistent TL-graph G=(V,E).
OUTPUT: G with vertices assigned to chains.

1. X := list of vertices in V;
2. *for each* vertex v $\in$ V chain(v):= nil; y := nil; c := 1;
3. x_{max} := a vertex in X with the highest rank;
4. y := v | v $\in$ V, chain(v)=nil and $(v, \{<, \leq\}, x_{max}) \in E$
   {preferring a vertex connected by a < edge};
5. *if* y is not nil *then begin*
6. chain(y) := c; remove y from X; x_{max} := y;
7. *goto* 4 *end*
8. *else if* X is not empty *then begin*
   c := c + 1; goto 3 *end*.

Figure 5: Algorithm for creating chains.

INPUT: a time chain C.
OUTPUT: the local nextgreater links for C.

1. i := first vertex on C ; n := last vertex on C;
2. *while* rank(i) <rank(n) *do*
3. k := successor of i on C; strict := false
4. *while* (strict = false and rank(k) $\leq$ rank(n)) *do*
5. $S = \{(j, \{<, \neq\}, k) \} \mid j \in C$ and
   rank(i) $\leq$ rank(j) < rank(k);
6. *if* S is not empty *then begin*
7. r := the vertex with the highest rank in
   $\{start(e) \mid e \in S\}$;
8. strict := true;
9. *for each* vertex t on C between i and r *do*
   nextgreater(t) := k;
10. i := successor of r on C *end*
11. *else* k:= successor of k on C
   end
end.

Figure 6: Algorithm for computing local nextgreaters.

Figure 6 shows an algorithm for performing the first step for a single chain C that takes $O(n_C + e_C)$ time, where n_C is the number of vertices on C and e_C is the number of edges between those vertices. The function $start((v_1, l, v_2))$ indicates the vertex between v_1 and v_2 that has the minimum rank.

Since two vertices on the same chain may also be connected via cross chain paths, it is clear that the first step is not sufficient for computing nextgreaters. The second step completes this task by performing a search for the <-paths that go from one cross-connected vertex of a chain to another on the same chain, starting with an out-going cross-edge and ending with an incoming cross-edge. For example, in Figure 7 the nextgreater of a is c, but before the refinement it was f. Edges with no label have been assumed to be labeled $\leq$. The dotted edges represent the nextgreater links. The number labeling each vertex is the pseudotime obtained using increments of 1000.

The second step is in general the most time-consuming in the computation of nextgreaters; in fact since each search can cost $O(e)$ time, the time complexity is $O(n \cdot e)$. However, for timegraphs it is possible to improve this bound significantly with a negligible cost in space complexity (a constant factor). In order to do that, we need

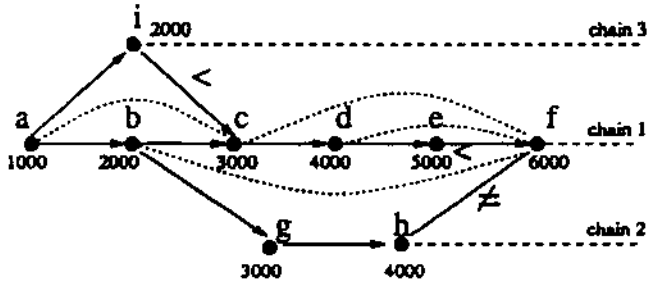

Figure 7: Timegraph of the TL-graph of Figure 2.

two new links for each cross-connected vertex $\hat{v}$. The first points to the first successor of $\hat{v}$ on the same chain which has outgoing cross-edges, and the second points to the first successor of $\hat{v}$ on the same chain which has incoming cross-edges. We indicate the vertices pointed to by these two links with *nextout($\hat{v}$)* and *nextin($\hat{v}$)*. Based on the metavertices and the metaedges of the timegraph, together with these new links, we define the *metagraph* of a given timegraph T:

Definition 3.3 Given a timegraph T with underlying TL-graph G, the *metagraph* of T is the graph $\hat{G} = (\hat{V}, \hat{E})$ in which $\hat{V}$={ v | v is a metavertex in T } and $\hat{E}$={e | e is a cross-edge in T}$\cup\{nextout(v_i), nextin(v_i) \mid v_i \in \hat{V}\}$.

It can be shown that $\hat{G}$ can be computed from T in $O(n + e)$ time. The algorithm for refining the nextgreaters on a chain C, not reported for lack of space, takes $O(n_C + \hat{n}_C \hat{e})$ time, where n_C is the number of vertices on C and $\hat{n}_C$ is the number of metavertices on C. The global complexity of the algorithm for computing the nextgreaters is then: $O((e - \hat{e}) + n + \hat{n} \cdot \hat{e})$. But, in practice, this bound can be reduced since the search required for refining the nextgreater of a vertex v can be pruned whenever a vertex with a rank greater or equal to the rank of the current nextgreater of v is reached (by the definition of rank and by the way the chains are formed, it is not possible to have a path from a vertex v to a vertex with a rank equal or greater than the rank of v).

3.4 Dealing with "not equal" relations

Reducing a TL-graph (i.e. making explicit all the implicit $<$ relation by the addition of new edges with label $<$) can be the most expensive task in the construction of a timegraph. This is the same as in van Beek's approach whose algorithm [van Beek, 1990] takes $O(n^2 \cdot e_{\neq} + n^3)$ time, with n the number of time points, and $e_{\neq}$ number of $\neq$ relations. Fortunately, the data structures provided by a timegraph allow this task to be accomplished more efficiently. Figure 1 shows the two cases of implicit $<$ relations. An algorithm for the first case (Figure 1.a)) is given in Figure 8. The time complexity of this algorithm is $O(e_{\neq} \cdot (\hat{e} + \hat{n} + k))$, where $e_{\neq}$ is the number of $\neq$ edges in the timegraph, $\hat{e}$ is the number of metaedges and k is the constant time required for determining the relation between points on the same chain. In fact, using the metagraph, the tests in step 2 and 3 can be performed in $O(\hat{e} + \hat{n} + k)$ time (see Section 3.5).

INPUT: a consistent TL-graph G=(V,E).
OUTPUT: G without the implicit $<$ of Figure 1.a).

1. *for each* edge $(v, \neq, w)$ in G
2. *if* there exists a $<$-path from v to w *then* remove $(v, \neq, w)$ from G
3. *else if* each path from v to w is a $\leq$-path but not a $<$-path *then* remove $(v, \neq, w)$ from G and add $(v, <, w)$ to G.

Figure 8: Algorithm for reducing $\neq$ relations.

In order to make implicit $<$ relations of the second kind (Figure 1.b) explicit, a number of $\neq$ diamonds of the order of $e_{\neq} \cdot n^2$ may need to be identified in the worst case. However, for timegraphs only a subset of these needs to be considered. In fact it is possible to limit the search to the *smallest* $\neq$ diamonds, i.e., the set of diamonds obtained by considering for each edge $(v, \neq, w)$ only the *nearest common descendants* of v and w (NCD(v,w)) and their *nearest common ancestors* (NCA(v,w)). This is a consequence of the fact that, once we have inserted a $<$ edge from a vertex in NCA(v,w) to a vertex in NCD(v,w), we will have explicit $<$-paths for all pairs of "diamond-connected" vertices. From the previous observation we can derive a criterion for pruning the search that in practice can give significant time savings. In fact the total number of diamonds that has to be considered is bounded by: $\sum_{(v,\neq,w)} | NCA(v,w) | \cdot | NCD(v,w) |$.

Figure 9 shows an efficient algorithm, based on the computation of the nearest common descendants and ancestors, for making the implicit $<$ relation of $\neq$ diamonds explicit. The algorithm uses the list OPEN as a supplemental data structure. Each item in OPEN is a pair (v, q) where v is a vertex to be visited and q is an integer between 1 and 4 called the code of v. The codes of two vertices are combined using the COMBINE table. The time complexity is $O(e)$, but exploiting the timegraph's data structures, we can obtain better performance by limiting the search to the metagraph. This is because if $(v, \neq, w)$ is a transitive edge on a chain we have an implicit $<$ relation of the first kind, and hence it is already considered by the algorithm of Figure 8. If it is a cross-chain edge, NCD(v, w) and NCA(v, w) must all be metavertices. So, indicating with $\overline{NCA}(v, w)$ and with $\overline{NCD}(v, w)$ respectively the subset of NCA(v, w) and the subset of NCD(v, w) consisting only of metavertices, the number of $\neq$ diamonds to be considered is reduced to $\sum_{(v,\neq,w) \in \hat{E}} | \overline{NCA}(v, w) | \cdot | \overline{NCD}(v, w) |$. The algorithm for computing NCA(v, w) ia analogous to the NCD algorithm and is not reported for lack of space.

3.5 Query algorithms

Given an explicit TL-graph, Theorem 2.2 permits us to derive the strongest relation between two time points that is entailed by the graph, just by looking for all the paths connecting the two corresponding vertices. Furthermore, in timegraphs there are four special cases in which those paths can be obtained in constant time. Given two points p_1,p_2 the first case is the one where P_1 and P_2 are alternative names of the same point (see

COMBINE

$c_1 \backslash c_2$	1	2	3	4
1	1	3	3	4
2	3	2	3	4
3	3	3	4	4
4	4	4	4	4

NCD-NCA algorithm
INPUT: a ranked TL-graph G
OUTPUT: the explicit graph of G

1. *for each* edge $(u, \neq, v)$ *do*
2. Y := NCD(u,v);
3. X := NCA(u,v);
4. *for each* pair x,y such that $x \in X$ and $y \in Y$ *do*
   add the edge $(x, <, y)$ to G.

NCD Algorithm
INPUT: a ranked TL-graph G and a pair of vertices u, v
OUTPUT: the nearest common descendants of u and v.

1. *for each* vertex $v \in G$ code(v) := 0; NCD := nil;
2. *if* rank(u) $\leq$ rank(v) *then* OPEN:= *((u 1) (v 2))*
   else OPEN:= *((v 2) (u 1))*
3. *if* there is no item in OPEN with code in {1,2,3}
   then return NCD;
5. w := pop(OPEN);
6. *if* code(w)=3 *then* add w to NCD and let code(w):=4;
7. X := $\{x \mid (w, \leq, x)$ is an edge of G $\}$;
8. *for each* $x \in X$ *do*
9. *if* $x \in$ OPEN *then* change code(x) on OPEN to
10. code(x) := COMBINE(code(w),code(x)))
11. *else* add the item *(x code(w))* to OPEN in
    rank order {maintaining vertices with
    the highest ranks in initial position };
12. *if* OPEN=nil *then return* NCD *else goto* 3

Figure 9: Algorithm for reducing $\neq$ diamonds. The meanings of code(x) are: 1 descendant of u; 2 descendant of v; 3 common descendant of u and v; 4 vertex from which the search can be stopped.

Section 3). The second case is the one where the vertices v_1 and v_2 corresponding to p_1, p_2 are on the same chain (see Section 3.3). The third case is the one where v_1 and v_2 are not on the same chain and have the same rank, and there is no $\neq$ edge between them (the entailed relation is $\{<, =, >\}$). The fourth case concerns $\neq$ relations for which the search is localized to the edges between the two input vertices. In fact, as a consequence of making implicit $<$ relations explicit, a $\neq$ relation can be the strongest relation entailed by the graph if and only if there exists a cross-edge with label $\neq$ connecting them. In the remaining cases an explicit search of the gra-ph needs to be performed. If there exists at least one $<$-path from v_1 to v_2, then the answer is $v_1 < v_2$. If there are only $\leq$-paths (but no $<$-paths) from v_1 to v_2, then the answer is $v_1 \leq v_2$. Analogously for paths from v_2 to v_1. An algorithm for accomplishing this task can be derived by a slight adaptation of the single-source-longest-paths algorithm reported in [Cormen *et al.*, 1990]. This algorithm has a time complexity of $O(n + e)$ but, exploiting again the timegraph's data structures, and in particular the ranks and the metagraph, we obtain a complexity $O(k + \hat{e} + \hat{n})$, where k is the constant corresponding to the time required by the four special cases.

points	CPU-time	chains	length-chain	max-chain
100	0.34	20.3	6.10	48.9
200	0.80	40.5	8.18	97.5
300	1.46	54.6	12.21	154.1
400	2.22	73.3	12.28	206.2
500	2.90	96.5	11.47	244.7
1000	9.01	182.8	14.36	508.8

Table 1: Statistics about the construction of a timegraph

4 Experimental Results

The algorithms described in the previous Sections have been implemented in a system called *TG-II* (Timegraph II). In this Section we report some preliminary results from large scale tests we are currently conducting on a SUN Sparcstation 2. Table 1 reports statistics about building timegraphs for randomly generated consistent sets of relations. For each number n of time points considered, the test procedure generates n sets of relations, each containing a number of relations equal to $n \cdot INT\{\log_2 n\}$. The table reports the average CPU-time for building the timegraph, the average number of chains, the average length of the chains and the average maximum length of the chains. Given a set S of n time points with indices 1, ...,n, the first rn points (5') - with m a random number between 1 and n - are chosen to represent distinct elements whose time order is the same as the order of their indices; the remaining $n - m$ points (S") are randomly assigned to coincide with points in S'. Relations are generated randomly by choosing a pair i,j (i < j) of indices. Specifically, if both i and j are among the m distinct points, then the relation iRj is generated, where R is randomly taken from $\{<, <, \neq\}$, with the percentage of $\neq$ relations kept low (below 2%). If i or j (or both) is one of the $n - m$ points in S", the corresponding point in 5' is considered. Since we were mostly interested in measuring *TG-II's* performance with data sets likely to allow chain formation, the pairs of points were generated using a geometric distribution with expected value 3. More general experiments that use the uniform distribution are in progress.

Figure 10 compares the CPU-times for reducing $\neq$ diamonds in a timegraph and in the corresponding network of relations (van Beek's algorithm [1990]). For each value marked on the curves 500 randomly generated data sets of $<$ relations were considered. Five *not reducible* $\neq$ relations (i.e. that do not induce any implicit $<$ relation of Figure Lb)) were then added to each data set. The high efficiency of NCD-NCA compared to van Beek's algorithm derives mainly from the fact that the number of $\neq$ diamonds examined by NCD-NCA was nearly constant for all the data sets generated. Other experiments show that querying a timegraph is on average a fast operation. For example, the average CPU-time over 100,000 queries on 100 randomly generated timegraphs with 500 points was 0.004 seconds. Finally, we should mention some recent experiments conducted by E. Yampratoom and J. Allen [to appear] comparing the performance of Timegraph I and II with several temporal reasoning systems; the timegraph approach proved by far the most efficient for large data sets generated for the TRAINS

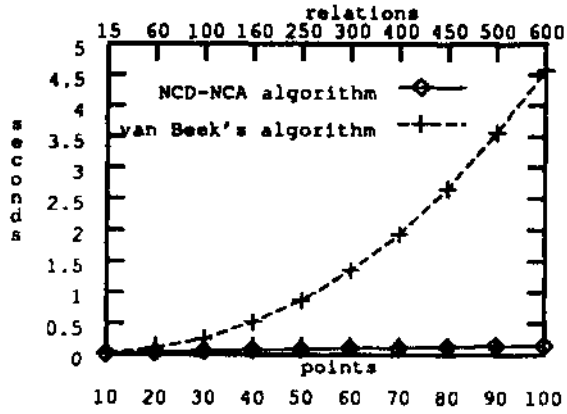

Figure 10: CPU-time for reducing ≠ diamonds.

world [Allen and Schubert, 1991].

5 Conclusions

In this paper we addressed the problem of scalability in temporal reasoning, proposing a collection of new algorithms that are implemented in a temporal reasoning system called TG-II. With respect to the constraint propagation approach, in our system both space and time complexity are reduced significantly for many practical applications. In fact, the space complexity depends on the size of the set of the stipulated relations and on a limited number of ≠ relations (a subset of those forming ≠ diamonds), and not on the number of time points (as in the constraint propagation approach). Instead of computing the closure of the set of relations, we build a timegraph providing a collection of data structures that allow efficient deduction of relations at query time. Experimental results show that building a timegraph is much faster than computing the minimal network and that, on average, querying relations is very efficient. The larger lesson here is that exploitation of the inherent temporal structure (in our case, the chain-like structure) of the information in certain important classes of domains can yield dramatic improvements in practical performance, vis a vts algorithms aimed at the worst case.

Current and future work concerns the design of efficient algorithms for adding new relations to the timegraph dynamically and the integration of metric relations [3]. We are also investigating an extension of the Point Algebra in order to include in a timegraph particular 3-point and 4-point relations which allow the representation of "disjoint" relations. Unfortunately, it appears that to obtain any significant extensions, while retaining computational tractability, we need to sacrifice completeness, since we recently established that even consistency for graphs containing only relations of form "x < y" and "x strictly outside the interval formed by y, z" (where y < z) is NP-complete [Gerevini and Schubert, 1993]. Thus we are exploring efficient, incomplete methods for such graphs.

[3]These were handled in the original implementation of timegraphs [Schubert et al., 1987; Miller and Schubert, 1990], but as noted ≠ was not handled and also < and < relations entailed via metric relations were not extracted in a deductively complete way.

References

[Allen and Schubert, 1991] J. Allen and L. Schubert. The trains project. Tech. Rep. 91-1, Department of Computer Science, University of Rochester, Rochester, NY, 1991.

[Allen et al., 1991] J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg. Reasoning about Plans. Morgan-Kaufman, 1991.

[Allen, 1983] J. F. Allen. Mantaining knowledge about temporal intervals. Communication of the ACM, 26(1):832-843, 1983.

[Cormen et al., 1990] T. Gormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. The MIT Press, 1990.

[Dechter tt al., 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. Artificial Intelligence, 49:61-95, 1991.

[Dorn, 1992] J. Dorn. Temporal reasoning in sequence graphs. In Proc. of the tenth AAAI, pages 735,740, 1992.

[Gerevini and Schubert, 1993] A. Gerevini and L. Schubert. Complexity of temporal reasoning with disjunctions of inequalities. Tech. Rep. 9303-01, IRST, 1993.

[Gerevini and Schubert, to appear] A. Gerevini and L. Schubert. Efficient temporal reasoning through timegraphs (extended report), to appear.

[Ghallab and Mounir Alaoui, 1989] M. Ghallab and A.Mounir Alaoui. Managing efficiently temporal rela as through indexed spanning trees. In Proc. of the Eleventh IJCAI, pages 1297-1303, 1989.

[Golumbic and Shamir, 1992] M. Goiumbic and R. Shamir. Algorithms and complexity for reasoning about time. In Proc. of the tenth AAAI, pages 741-746, 1992.

[Ladkin and Maddux, 1988) P. Ladkin and R. Maddux. On binary constraint networks. Tech. Rep. KES.U.88.8, Kestrel Institute, Palo Alto Calif., 1988.

[Miller and Schubert, 1990] S A. Miller and L. K. Schubert. Time revisited. Computational Intelligence, 6:108-118, 1990.

[Schubert et ai, 1987] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Accellerating deductive inference: Special methods for taxonomies, colours, and times. In N. Cercone and G. McCalla, editors, The Knowledge Frontier: Essays in the Representation of Knowledge, pages 187-220. Springer-Verlag, 1987.

[van Beek and Cohen, 1990] P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. Computational Intelligence, 6:132-144, 1990.

[van Beek, 1990] P. van Beek. Reasoning about qualitative temporal information. In Proc. of the eighth AAAI, pages 728-734, 1990.

[Vilain et al., 1990] M. Vilain, H. Kautz, and P. van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. In Readings in Qualitative Reasoning about Physical Systems, pages 373-381. Morgan Kaufman, 1990.

[Yampratoom and Allen, to appear]
E. Yampratoom and J. Allen. Performance of temporal reasoning systems, to appear.