Point-Based Approaches to Qualitative Temporal Reasoning

J. Delgrande, A. Gupta,
School of Computing Science,
Simon Fraser University,
Burnaby, B.C., Canada, V5A 1S6.
E-mail: {jim, arvind}@cs.sfu.ca

T. Van Allen
Department of Computing Science,
University of Alberta,
Edmonton, Alberta, Canada T6G 2H1
E-mail: vanallen@cs.ualberta.ca

Abstract

We address the general problem of finding algorithms for efficient, qualitative, point-based temporal reasoning over a set of operations. We consider general reasoners tailored for temporal domains that exhibit a particular structure and introduce such a reasoner based on the series-parallel graph reasoner of Delgrande and Gupta; this reasoner is also an extension of the TimeGraph reasoner of Gerevini and Schubert. Test results indicate that for data with underlying structure, our reasoner performs better than other approaches. When there is no underlying structure in the data, our reasoner still performs better for query answering.

Introduction

Reasoning about temporal events is a central problem in the design of intelligent systems for such diverse areas as planning, reasoning about action and causality, and natural language understanding. Allen (All83) proposed the *interval algebra* (IA) of temporal relations wherein time intervals are taken as primitive; reasoning within this algebra is NP-complete (VK86). The *point algebra* (PA), introduced in (VK86; VKvB90), is based on time points as primitives. Many important temporal problems are expressible in the point algebra, and the existence of significantly more efficient algorithms than in the interval algebra has lead to its extensive study. Here, we are interested in the general problem of efficient, qualitative, point-based temporal reasoning.

A major problem in temporal reasoning is *scalability*. An $O(n^2)$ algorithm, while seemingly efficient, can be unacceptable for very large database. Matrix based deductive closure techniques for qualitative point based reasoning (VKvB90) require $O(n^2)$ space and $O(n^4)$ time and are only useful when data sets are small and "dense" ($O(n^2)$ assertions) and the number of queries is large. Research ((GS95; GM89)) has focused on methods which, by sacrificing query speed, achieve faster compilation and require less storage. Such approaches are appropriate for large, sparse data sets and are complementary to the matrix based approach.

Another consideration is the nature of the application domain. The underlying structure in a restricted domain may admit significantly faster algorithms. Gerevini and Schubert in their seminal work (GS95) study reasoners where the domain is dominated by *chains* of events. Building on this, (DG96) consider a reasoner for domains that may be modelled by *series-parallel graphs*. These restricted structures may be exploited for general point-based reasoning by first decomposing a general graph into components of the restricted graphs, computing the closure of each component, and then using lookup and search to answer queries (see (GS95; GM89)).

Here we explore and compare different approaches to point-based reasoning. We first describe *point relations* and the point algebra, define entailment with respect to this algebra, and discuss operations on the language. We develop two basic reasoners, the first using standard graph operations, and a second that also uses rankings of nodes to improve query-answering times. We also consider two general reasoners tailored to temporal domains that likely exhibit a particular structure. We develop and describe an implementation of a general reasoner based on series parallel graphs, subsuming and improving on (DG96). We also reimplement the chain-based approach of Gerevini and Schubert, describing how our implementation differs from the original.

The four reasoners are tested on random data sets from different domains. The series-parallel reasoner is consistently the fastest for query answering and is faster than the Gerevini and Schubert approach for compilation even for those domains specifically tailored for this latter approach. Consequently, the series-parallel reasoner gives the best expected performance.

Preliminaries

Our results rely substantially on graph theoretic concepts (see (BM76) for terms not defined here). All graphs G are simple, finite and directed. The node and edge sets of G are denoted by $V(G)$ and $E(G)$ respectively. For $v \in V(G)$, $incoming(G, v)$ ($outgoing(G, v)$) are the edges terminating (starting) at v. We use n to denote $|V(G)|$ as well as the number of points in a database of point relational constraints (these are normally the same). Similarly, e denotes the number of edges in a graph or the number of constraints in the database. For $u, v \in V(G)$, an unlabeled edge (or an edge where the label is implicit) is denoted by (u, v)

and a labeled edge by (u, m, v), where m is the label. We assume that edge labels support the standard operations of composition and summation, which we denote $\odot$ and $\oplus$ respectively (see (CLR90) for more details). Via these operations we extend the notion of edge labels to path labels. We use $u \rightsquigarrow v$ to indicate that there is a path from u to v, where a path may have length 0. A subscript will denote a path label: $u \rightsquigarrow_r v$ indicates that some edge in the path is labeled by r.

We assume that basic operations on $\log n$ bit integers are performed in constant time and the numbers require unit space. This is a standard complexity-theoretic assumption consistent with other work in the area.

Point Relations

Our primitive objects are *points* which can be viewed as "events in time" or points on the real line (but we make no such assumption). We denote the infinite set of points by P. The *point relations* are a set $R = \{\emptyset, <, >, =, <>, <=, >=, <=>\}$ where each element can be viewed as a subset of $\{<, =, >\}$, the *primitive point relations*. A point relation in R is a disjunction of primitive relations; thus $\emptyset$ is the relation that never holds and $<=>$ is the relation that always holds. The standard set operations $\cup, \cap, \subseteq, \ldots$ are defined over the point relations, with obvious interpretations. Relation r_1 is *stronger* than r_2 iff $r_1 \subseteq r_2$.

We also have two functions, *sequence*: $R \times R \rightarrow R$ and *inverse*: $R \rightarrow R$. *Sequence* is the transitive relation entailed by a sequence of two point relations. It is the composition operation for edge labels over R. *Inverse* maps a single relation onto the relation that holds in the opposite direction.

Constraints The language C is the set of all sentences $S ::= P\ R\ P$; a *constraint* is an element of C. At times we use *assertion* to mean *constraint*. For $A \subset C$, $P(A)$ is the set of all points "mentioned" in A. Entailment in C can be axiomatized as follows:

1. $\{\} \models x = x$, for all $x \in P$
2. $\{\} \models x <=> y$, for all $x, y \in P$
3. $\{x\ r\ y\} \models y\ inverse(r)\ x$
4. $\{x\ r_1\ y\} \models x\ r_1 \cup r_2\ y$, for all $r_2 \in R$
5. $\{x\ r_1\ y,\ y\ r_2\ z\} \models x\ sequence(r_1, r_2)\ z$
6. $\{x\ r_1\ y,\ x\ r_2\ y\} \models x\ r_1 \cap r_2 y$
7. $\{x <= v, x <= w, v <> w, v <= y, w <= y\} \models x < y$

Note that in Axiom 7, if $a <> b$ then either $a < b$ or $b < a$. But then, if $x <= a$ and $x <= b$, then either $x < a$ or $x < b$. This is a valid entailment in propositional logic but one which we cannot express in C. However, since $a <= y$ and $b <= y$, it follows that $x < y$. (GS95) prove completeness for this formulation. We will call *inconsistent* any set of assertions which entails $x\ \emptyset\ y$ for any $x, y \in P$.

Operations on Constraint Sets We are interested in the general problem of computing the entailments of a given set of point relational constraints. There are three basic subproblems:

1. Compilation: compile a set of constraints in C into a representation that allows efficient reasoning.
2. Querying: given such a representation, compute the strongest relation between two points.
3. Updating: change the representation to reflect the addition of a new assertion.

Generally, there is a tradeoff between compilation and query answering. In some applications it might be more efficient to precompute all strongest relations and explicitly store these; since there are $O(n^2)$ strongest relations which can be stored in a table, query answering can be performed in $O(1)$ time. However, computing deductive closure on a set of point relation constraints is not a particularly efficient process because of axiom 7 (deductive closure is not transitive closure for temporally labeled graphs). We can compute the transitive closure of the $<, <=$ labels with an $O(ne)$ algorithm, but to handle implicit $<$ relations entailed by axiom 7 we must resort to an $O(|E_{<>}| \cdot |E_{<=}|) = O(e^2)$ algorithm. The approach of (GS95) and (VKvB90) yields a $O(n^2)$ algorithm, as opposed to $O(en^2)$ by using simple search to compute all strongest relations.

For applications with static unstructured data and a large number of queries, our results suggest a "lazy" scheme which adds strongest relations to a hash table as they are computed. To maintain linear space, the table would be an $O(n)$ array with collisions resolved by overwriting previous information. With structured data however, our query answering algorithms are significantly faster than any other to date.

Temporally Labeled Graphs

Temporally labeled graphs (GS95) are used to represent "compiled" consistent sets of point relations.

Definition 1 For $A \subset C$, the *temporally labeled graph representing* A is a graph G with $V(G)$ the set of all $=$ points mentioned in A and $E(G)$ the constraints of A labeled by one of $<$, $<=$, $<>$.

Note that edges labeled $<$ or $<=$ are directed but those labeled $<>$ are undirected. $E(G)$ can be partitioned into the sets $E_<$, $E_{<=}$,and $E_{<>}$ based on edge labels. The graph composed of only edges from $E_< \cup E_{<=}$ is the $(<, <=)$-subgraph of G. Algorithms for compiling a set of assertions into such a graph and testing it for consistency are given in (GS95) and (vB92). They show that any set of constraints from C can be translated into constraints using only the relations $\{<, <=, <>\}$. The set of assertions is inconsistent iff a $<>$ or $<$ edge spans a cycle. We make G into a directed acyclic graph (DAB) by collapsing all directed cycles into single vertices.

Compilation takes $O(e)$ time using Tarjan's *strongly connected components* algorithm to isolate maximal cycles (see (CLR90)). The strongest relation between two points is found in $O(e)$ time using depth-first search to find the strongest path between the points.

For updates, suppose the assertion $x r_1 y$ is added. We compute the strongest relation, r_0, between x, and y. If r_0 is consistent with r_1 the update proceeds but any

new cycle is "collapsed" to a single node of the graph; this algorithm takes $O(e)$ time.

Ranking We can speed up multiple searches in a DAG by bounding search depth. We define the *rank* of a node as the length of the longest path from the source to that node. To search all paths between two nodes we can confine our search to those nodes with intermediate ranks. We have developed both ranked and non-ranked base reasoners for comparison with other approaches.

Series Parallel Graphs

The point algebra provides a very general framework for reasoning which can be used in any domain modeled as points on a line. However, certain restricted domains lend themselves to more efficient reasoning strategies.

Consider a domain in which sets of events are related to others via some simple operations. For example, two events may occur sequentially (in *series*) or they may occur during some common time frame (in *parallel*). If the structure of our events is defined recursively by these series and parallel operations, we obtain *series parallel graph* (sp-graph) structures. We will see that more efficient, general, point-based reasoners can be constructed from this structure.

Definition 2 A *sp-graph* G with properties $source(G)$, $sink(G)$, $label(G)$, is given by:

1. (Base case) $G = edge(v, r, w)$, where:
   (a) $V(G) = \{v, w\}$, $E(G) = \{(v, r, w)\}$
   (b) $source(G) = v$, $sink(G) = w$, $label(G) = r \in \{<, \leq\}$
2. (Inductive case) $G = series(G_1, G_2)$ where:
   (a) G_1 and G_2 are sp-graphs
   (b) $V(G) = V(G_1) \cup V(G_2)$
   (c) $V(G_1) \cap V(G_2) = \{sink(G_1)\} = \{source(G_2)\}$
   (d) $E(G) = E(G_1) \cup E(G_2)$, $E(G_1) \cap E(G_2) = \{\}$
   (e) $source(G) = source(G_1)$, $sink(G) = sink(G_2)$
   (f) $label(G) = label(G_1) \odot label(G_2)$
3. (Inductive case) $G = parallel(G_1, G_2)$ where:
   (a) G_1 and G_2 are sp-graphs
   (b) $V(G) = V(G_1) \cup V(G_2)$
   (c) $V(G_1) \cap V(G_2) = \{source(G_1), sink(G_1)\}$
   (d) $E(G) = E(G_1) \cup E(G_2)$, $E(G_1) \cap E(G_2) = \{\}$
   (e) $source(G) = source(G_1) = source(G_2)$
   (f) $sink(G) = sink(G_1) = sink(G_2)$
   (g) $label(G) = label(G_1) \oplus label(G_2)$

Note that sp-graphs have a single source and sink. The label summarizes all paths from the source to the sink.

Transitive Closure

It is straightforward to compute the transitive closure of sp-graphs in $O(n^2)$ time and space when the edge labels support $O(1)$ composition and intersection. For G an edge (u, r, w), the closure is r. If G is either $series(G_1, G_2)$ or $parallel(G_1, G_2)$ we inductively compute the closure of G_1 and G_2. Additionally in the series case, for u in G_1 and v in G_2 we must compose the path from u to $sink(G_1)$ and the path from $source(G_2)$ to v.

The structure of sp-graphs allows more efficient algorithms for special cases. (DG96) show that the complexity of determining path closure is $O(n)$. This involves associating each node v with a point (x_v, y_v) in the $n \times n$ integer lattice. For $v, w \in V(G)$ there is a path from v to w iff $x_v < x_w$ and $y_v < y_w$.

Given path closure, we will compute the $<, <=$ closure in an sp-graph G in $O(n)$ time.

Definition 3 For $v \in V(G)$, let $s(v)$ be the maximum number of $<$ edges on any path from $source(G)$ to v.

Definition 4 For $v \in V(G)$, if $v = sink(G)$ or, for some w, $(v, <, w) \in E(G)$ then define $a(v) = s(v)$ otherwise define $a(v) = \min\{a(w) : (v, <=, w) \in E(G)\}$.

Lemma 1 For any $v \in V(G)$, $s(v) \leq a(v)$.

Proof: The proof is by induction. Suppose that for every child w of v, $s(w) \leq a(w)$. We show $s(v) \leq a(v)$. If $v = sink(G)$ or v has an outgoing $<$ edge, then by definition $s(v) = a(v)$. If v has no outgoing $<$ edge, then $a(v) = \min\{a(w) : (v, <=, w)\}$ and $a(v) = a(w)$ for some child w of v. Since $s(v) \leq s(w)$ and $s(w) \leq a(w) = a(v)$, the lemma follows. **QED**

Theorem 1 For $v, w \in V(G)$ such that $v \rightsquigarrow w$, $a(v) < s(w)$ iff $v \rightsquigarrow_< w$.

Proof: Suppose $v, w \in V(G)$ such that $v \rightsquigarrow w$. If $v \rightsquigarrow_< w$ there there is an edge $(x, <, y)$ such that $v \rightsquigarrow x$ and $y \rightsquigarrow w$. Since a and s are non-decreasing, $a(v) \leq a(x) = s(x) < s(y) \leq s(w)$.

Conversely assume $a(v) < s(w)$ but $v \not\rightsquigarrow_< w$. Since $s(source(G)) \leq s(v) \leq a(v) < s(w)$, $s(source(G)) < s(w)$. Then there is an edge $(x, <, y)$ such that $source(G) \rightsquigarrow x$, $y \rightsquigarrow w$ and $s(w) = s(y)$. Since $a(v) < s(w) \leq a(w) \leq a(sink(G))$, there is an edge $(u, <, z)$ such that $v \rightsquigarrow u$, $z \rightsquigarrow sink(G)$, and $a(v) = a(u)$. Since $v \not\rightsquigarrow_< w$, $z \not\rightsquigarrow w$ and $v \not\rightsquigarrow x$. Because G is a sp-graph, the three paths

1. $v \rightsquigarrow u \rightsquigarrow_< z \rightsquigarrow sink(G)$
2. $source(G) \rightsquigarrow x \rightsquigarrow_< y \rightsquigarrow w$
3. $source(G) \rightsquigarrow v \rightsquigarrow w \rightsquigarrow sink(G)$

have a common node (say j). Since $v \not\rightsquigarrow_< w$:

$$v \rightsquigarrow j \rightsquigarrow w \text{ and } y \rightsquigarrow j \rightsquigarrow u.$$

Then $a(v) \leq a(j) \leq a(u) = a(v)$ so $a(j) = a(v)$ and $s(y) \leq s(j) \leq s(w) = s(y)$ so $s(j) = s(w)$. Thus $s(j) \leq a(j) = a(v) < s(w) = s(j)$, a contradiction. **QED**

We can compute $\{s(v)\}$ using depth first search on G and compute $\{a(v)\}$ from $\{s(v)\}$ using depth first search on the transpose of G. Both these computations require $O(n)$ time. This yields the following:

Theorem 2 For G a temporally labeled sp-graph, the $<$ and $<=$ closures can be computed in $O(n)$ time with $O(n)$ bits of storage.

Notice that our technique strictly improves on that in (DG96) who also claim $O(n)$ time and space but require arbitrary real number precision ($O(n^2)$ bits of storage).

Metagraphs

We slightly generalize the terms *metagraph*, *metaedge* and *metanode* introduced in (GS95).

Definition 5 A graph G' is a *metagraph* of a DAG G iff $V(G') \subseteq V(G)$ and there is an onto function $m : E(G) \rightarrow E(G')$ such that, for $m((x,y)) = (u,v)$:

1. Either $x = u$ or for all $(w,x) \in E(G)$, $m((w,x)) = (u,v)$ (and at least one such (w,x) exists); and
2. Either $y = v$ or for all $(y,w) \in E(G)$, $m((y,w)) = (u,v)$ (and at least one such (w,x) exists).

The nodes and edges of G' are called *metanodes* and *metaedges* respectively. Metaedges correspond to (edge disjoint) single source, single sink components of G. The edge label of $(u,v) \in E(G')$ is the intersection of the labels of all paths from u to v in G.

Metagraphs are a convenient way to encapsulate subgraphs. Metaedges correspond to components. Relations between nodes inside a metaedge are determined by the subgraph corresponding to that metaedge; relations between metanodes are determined by the metagraph; while relations between nodes inside different metaedges are determined by their relationship to the sources and sinks of their metaedges, and by the relations between the source/sink metanodes.

Metagraphs are useful for representing domains where relational data is composed of "self-contained" units connected only through common sources and sinks. Using a metagraph may improve efficiency when graphs are largely composed of substructures that lend themselves to more efficient implementation of basic operations (search, closure, updates, etc.).

Series Parallel Metaedges

We present a method for partitioning the edge set of a graph into maximal series parallel metaedges. We start with a very high level algorithm that collapses all series parallel components into single edges. Let G be a DAG.

Rule A If $v \in V(G)$ has only one incoming edge (u,m,v) and only one outgoing edge (v,n,w) then remove these edges and add the edge $(u, m \odot n, v)$.

Rule B If $(u,m,v),(u,n,v) \in E(G)$ then remove these edges and add the edge $(u, m \oplus n, v)$.

These rules are iterated so that in the resulting metagraph, each metaedge represents an edge disjoint, maximal, series parallel component of the original graph. We can label metaedges with sp-graphs and define $\odot$ as a series step and $\oplus$ as a parallel step. In (VDG98) we show that this algorithm correctly reduces a graph to its maximal series parallel components.

Our Temporal Reasoner

So far we have presented a base point algebra reasoner, closure algorithms for sp-graphs, and an algorithm for transforming a graph into series parallel metaedges. Our hybrid scheme is unique in constructing a temporally labeled graph, transforming it into maximal series parallel metaedges and then computing the internal closure of the metaedges. Queries for two points in the same metaedge take constant time; for points not in the same metaedge we only search the metagraph. Updating the structure follows a similar pattern. This approach will be useful in domains where temporal information is hierarchically structured, and when queries tend to reflect that structure.

Implementation and Testing

For comparison, we implemented four temporal reasoning algorithms:

1. The simple graph-based approach.
2. The graph-based approach using ranking.
3. The TimeGraph approach of Gerevini and Schubert (GS95), updated as described below.
4. Our metagraph algorithm based on decomposing a DAG into series parallel components.

Our implementations are in Common Lisp with data structures kept consistent between the four approaches.

Reimplementing TimeGraph

The TimeGraph approach, of (GS95), is arguably the first metagraph-based approach. We implemented a revised version of TimeGraph that is significantly faster than the original but did not implement those portions of TimeGraph, dealing with disjunctive relations since they are not directly relevant to us (see (VDG98) for details.) We briefly describe our version, noting where we diverge from the original.

TimeGraph differs from the series parallel metagraph approach in several significant ways:

1. The underlying components are chains instead of series parallel graphs.
2. Entailed $<$ relations involving $<>$ are compiled out.
3. Chains may be connected via *cross-edges* (edges that originate and terminate in different chains). Computing closure within a chain involves following paths outside the chain, which takes $O(ne)$ time.

Making the Graph Explicit We must add $<$ edges so that all $<>$ relations between nodes involve a $<$ edge. For each $(u,v) \in E_{<>}$ we find the least common ancestors and descendants of u and v and add the cross product of these two sets to $E_<$. The total complexity is $O(|E_{<>}| \cdot |E_{<=}|)$. Notice that this algorithm adds at most $O(e)$ edges to the graph. Our algorithms for computing nearest common ancestors and descendants are based on an algorithm in (GS95).

Computing Closure for the Chains A chain G is a DAG such that for $v, w \in V(G)$, either $v \rightsquigarrow w$ or $w \rightsquigarrow v$. A *timechain* is an sequence of nodes, $[v_1, v_2 \ldots v_n]$, with $(v_i, <=, v_{i+1}) \in E(G)$ for $i < n$. A timechain may also contain $<$ links between any two nodes (in the consistent direction). To determine $<$ relations, each node is labeled with the index of the next $>$ node on the chain and $n+1$ if no such node exists. These labels can be determined with a single depth first search.

Since chains can be connected by cross-edges between arbitrary nodes, computing the internal closure of a

chain requires searching the entire region of the graph reachable from the nodes of the chain (bounded by the rank of the last node on the chain). For c chains, this step requires $O(c(n+e))$ time; since c is $O(n)$, the complexity is $O(n(n+e))$. To compute a *nextGreater* value (ie rank of the next greater node) for each node, (GS95) use the metagraph to speed up the search. This necessitates breaking the algorithm into two parts – one to compute the *nextGreater* value based only on edges internal to the chain, and one to "refine" these *nextGreater* values by searching the graph. We use an algorithm that computes the *nextGreater* values with one depth first search for each chain.

Constructing the Metagraph The metagraph must reflect all relationships between metanodes so that any relation between two metanodes can be determined by search. (GS95) define metavertices as "cross-connected vertices" (the sources and termini of cross-edges) and the metagraph of a timegraph T as:

> ... the graph $G' = (V', E')$ where $V' = \{v : v$ is a metavertex in $T\}$ and $E' = \{(v, l, w) : (v, l, w)$ is a cross-edge in $T\} \cup \{(v, nextout(v)), (v, nextin(v))$ for all $v \in V'\}$.

where $nextout(v)$ gives the next node on v's chain with an outgoing cross-edge, and $nextin(v)$ gives the next node on v's chain with an incoming cross-edge.

However, this graph does not contain sufficient information to deduce all relations between metanodes. Consider the chains $[a, b, c, d, e]$ and $[f, g, h]$, cross-edges $(a, <=, f), (c, <=, g), (e, <=, h)$ and the transitive edge $(b, <, d)$. The metagraph in the definition above would be: $V' = \{a, c, e, f, g, h\}$ and $E' = \{(a, <=, c), (a, <=, f), (c, <=, e), (c, <=, g), (f, <=, g), (e, <=, h), (g, <=, h)\}$ In the original graph, $a < e$ and $a < h$, but the metagraph does not allow us to deduce this. To fix this, we must include b and d in V', and $(b, <, d)$ in E'. Therefore, we slightly modify the above definition to:

Definition 6 (v, r, w) is a metaedge, and v, w are metanodes, iff one of the following holds:

1. $chain(v) \neq chain(w)$.
2. w is the *nextGreater* value for v and r is $<$.
3. All of:
   (a) v, w are metanodes as defined above.
   (b) $chain(v) = chain(w)$.
   (c) No metanode lies between v and w on the chain.
   (d) r is the internal relation between v and w.

Our definition allows us to compute all relations between metanodes. We construct the metagraph by finding all edges of the first two kinds, labeling their sources and sinks as metanodes, and then adding the edges of the third kind by going through the chains and adding links between nearest metanodes.

Queries Queries in the revised TimeGraph are basically the same as in our series parallel metagraph: if the two nodes are on the same chain then their strongest relation is computed with an $O(1)$ test; otherwise a search is conducted from the next metanode of the lowest ranked node to the previous metanode of the highest ranked node. The relation so obtained is composed with the relation each holds to the appropriate metanode to yield the strongest relation between the two nodes.

Testing

To compare approaches, we generated random sets of constraints (DAG's) and compared compile and query times. To generate a DAG over the nodes $v_1, v_2, \ldots v_n$, we let the ordering on nodes define a topological sort of the intended graph, and defined the graph by enumerating the forward edges, adding an edge with some probability. For the purpose of testing the approaches presented in this paper, we were interested only in sparse graphs, that is, when e is $O(n)$. To produce sparse graphs we add an edge with a probability of approximately k/n, for some fixed k. Then we randomly assigned labels from $\{<, <=, <>\}$ to the edges. This set of edges corresponds to a set of assertions.

Test Domains We experimented with various modifications to the basic graph generation scheme. We investigated data sets with no <> assertions, data sets based on series parallel graphs with a certain amount of "noise" (random edges) introduced, and data sets based on chains where, again, a certain amount of noise was added. Here we describe three test domains:

1. Sparse graphs with all edge labels equiprobable.
2. Sparse graphs based on chains (no <> edges).
3. Sparse graphs based on sp-graphs (no <> edges).

Test Results Thirty data sets were generated with the average time to compile and the average time to answer 100 queries given below. Times are in milliseconds; the tests were run on a Sun UltraSparc 1 workstation.

Domain 1

	Approach	Nodes (Edges = ~ 5× nodes)				
		100	200	300	400	500
Compile Time	Base	5	9	13	17	22
	Ranked	4	10	16	21	27
	TG	304	1150	2477	4501	6857
	SP	69	138	207	286	354
Query Time	Base	38	56	69	78	91
	Ranked	19	25	28	31	35
	TG	21	25	31	35	38
	SP	18	25	30	33	35

Domain 2

	Approach	Nodes (Edges = ~ 1.2× nodes)				
		100	200	300	400	500
Compile Time	Base	3	6	10	13	15
	Ranked	3	6	10	16	20
	TG	16	41	82	133	200
	SP	17	34	52	69	86
Query Time	Base	42	61	70	77	84
	Ranked	20	26	26	27	28
	TG	10	13	14	15	17
	SP	11	12	13	14	16

Domain 3

	Approach	Nodes (Edges = $\sim 1.6\times$ nodes)				
		100	200	300	400	500
Compile Time	Base	5	6	9	12	15
	Ranked	5	7	11	16	19
	TG	31	103	214	375	557
	SP	21	42	62	83	107
Query Time	Base	45	89	133	172	225
	Ranked	21	35	44	58	65
	TG	20	34	44	55	63
	SP	13	16	21	28	30

Discussion of Test Results

The test results indicate that, as expected, the simple approaches have much lower compile times. In all domains, adding ranking ("Ranked") to brute force searching ("Base") yields significantly faster query times with little increase in compile times. When the data has no underlying structure (Domain 1), ranking and the series parallel approach perform optimally for query but because of its reduced compile time, ranking is the better approach.

The situation changes dramatically when the data has some underlying structure. When there are many chains in the database (Domain 2) both the TimeGraph approach ("TG") and the series parallel approach ("SP") yield query times that are half those of ranking. While the query times for both approaches is similar, the compile time for SP is about half that for TG. This may seem surprising but notice that when the number of edges is linear, sp-graphs effectively encompass chains. Thus the SP approach will likely have metanodes that contain many chains thus speeding up the query answering. When there are many sp-graphs in the database (Domain 3), the SP approach answers queries about twice as fast as either ranking or TG.

We conclude that, when the data exhibits no structure, ranking is a feasible approach. It is easy to implement, allows for quick compilation, and is the fastest (or near-fastest) in almost all cases. When the data exhibits chains or series parallel structures, SP is the best choice especially when a large number of queries must be performed. When query time significantly dominates compile time, the SP approach seems the best all-round approach, since it performs no worse in answering queries than other approaches, yet is able to exploit any structure in the time constraints.

Conclusion

We have addressed the general problem of qualitative, point-based temporal reasoning. To this end, we have developed two basic reasoners, one a straightforward implementation of standard graph operations, and a second that also incorporates a ranking of nodes to improve query-answering times. We also developed general reasoners tailored for temporal domains that are expected to exhibit a particular structure. This leads naturally to the notion of a *metagraph*, and metagraph reasoners. We implemented the chain-based approach of Gerevini and Schubert and developed and implemented a *series-parallel graph* approach, roughly analogous to the Gerevini and Schubert approach but where series-parallel graphs replace chains.

Our test results indicate that when there is some underlying structure in the data, our reasoner performs better than the Gerevini and Schubert reasoner or simple search algorithms. When there is no underlying structure in the data, the series parallel reasoner still performs better for query answering than these other approaches. Hence, when there is no *known* structure in the data, we argue that our reasoner will provide the best expected performance: if the domain is indeed unstructured, our approach performs generally better than the others for the (presumably) dominant operation of query answering; if there is some structure in the date, this structure will be exploited by our reasoner for overall superior results.

References

James Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(1):832–843, 1983.

J. Bondy and U.S.R. Murty. *Graph Theory with Applications*. North-Holland, 1976.

T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, 1990.

J.P. Delgrande and A. Gupta. A representation for efficient temporal reasoning. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 381–388, Portland, Oregon, August 1996.

Malik Ghallab and Amine Mounir Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1297–1303, Detroit, 1989.

Alfonso Gerevini and Lenhart Schubert. Efficient algorithms for qualitative reasoning about time. *Artificial Intelligence*, 74(2):207–248, April 1995.

Peter van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58(1-3):297–326, 1992.

T. Van Allen, J. Delgrande, and A. Gupta. Point-based approaches to qualitative temporal reasoning. Technical Report CMPT TR 98-16, Simon Fraser University, 1998.

Marc Vilain and Henry Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 377–382, Philadelphia, PA, 1986.

Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1990.