Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases

THOMAS DEAN

Brown University, Providence, Rhode Island

Abstract. Many real-world applications involve the management of large amounts of time-dependent information. Temporal database systems maintain this information in order to support various sorts of inference (e.g., answering questions involving propositions that are true over some intervals and false over others). For any given proposition, there are typically many different occasions on which that proposition becomes true and persists for some length of time. In this paper, these occasions are referred to as *time tokens*. Many routine database operations must search through the database for time tokens satisfying certain temporal constraints. To expedite these operations, this paper describes a set of techniques for organizing temporal information by exploiting the local and global structure inherent in a wide class of temporal reasoning problems. The global structure of time is exemplified in conventions for partitioning time according to the calendar and the clock. This global structure is used to partition the set of time tokens to facilitate retrieval. The local structure of time is exemplified in the causal relationships between events and the dependencies between planned activities. This local structure is used as part of a strategy for reducing the computation required during constraint propagation. The organizational techniques described in this paper are quite general, and have been used to support a variety of powerful inference mechanisms. Integrating these techniques into an existing temporal database system has increased, by an order of magnitude or more in most applications, the number of time tokens that can be efficiently handled.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*heuristic methods*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Discrimination trees, reason maintenance, temporal reasoning

1. *Introduction*

Representing and reasoning about time play a critical role in many facets of everyday problem solving. We continually have to make reference to what has happened, is happening, and might possibly happen. To make matters all the more difficult, we have to cope with the fact that the world is constantly changing around us. To plan for the future, we must be able to predict change, propose, and commit to actions on the basis of these predictions, and notice when certain predictions are no longer warranted. All of this requires handling an enormous amount of complexly interdependent information.

This work was supported in part by the National Science Foundation under grant IRI 86-12644 and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0132.

Author's address: Department of Computer Science, Brown University, Providence, RI 02912.

In the past, the problems of efficient inference in predicate-calculus databases used for temporal reasoning tasks have largely been ignored as researchers have grappled with more fundamental issues. As our representations have become more sophisticated and our ambitions to tackle more realistic domains have grown, the problems inherent in managing large temporal databases have become a major factor limiting growth [3, 41, 45]. What is needed is a computational framework in which strategies for organizing temporal facts can be exploited to expedite the search needed to support basic temporal inference procedures. As a simple example, suppose that you are planning a business trip and you are trying to remember whether the travel agent has already confirmed your airline reservation. It should not be necessary to recall (i.e., search through) all of the events past, present, and future that involve your communicating with a travel agent. Only the most recent are likely to be of interest. Restricting attention to a particular interval of time requires that facts that change over time are *indexed temporally*, that is to say, stored in such a way that facts and events common to a given interval are easily accessible from one another.

Temporal indices are subject to frequent revision. Quite often, the span of time associated with an event or fact is shifted, compressed, or expanded to suit a change of plans or reflect new information. Suppose that you decide to leave on your business trip a week earlier than previously planned. From this explicit change, it should be apparent that certain prerequisite tasks (e.g., ordering plane tickets) must also occur earlier. The database system described in this paper ensures that such implied changes are reflected directly in the data structures manipulated by the system. In updating a database of temporally dependent facts, it is our goal that the work performed by the system be proportional to the resultant changes and not to the size of the entire database.

In this paper, we are concerned with expediting a particular type of temporal inference useful in a wide variety of applications. These applications range from decision support for office automation [13, 14] to robot problem-solving [15, 18]. In the rest of this paper, wherever possible, we give concrete examples drawn from an application that involves keeping track of the status of a number of manufacturing devices (e.g., engine lathes and milling machines) in an automated factory.

For any particular application, we want to encode general knowledge about how various propositions relate to one another causally and temporally (e.g., "If a machine is repaired, then the machine will be available for use immediately following the repair.") and specific knowledge about particular situations (e.g., "Lathe #45 was repaired at 2:00 this afternoon."). We also want to be able to answer questions that bear on our general and specific knowledge (e.g., "What machines were repaired this afternoon?" or "Is there a lathe available for use at 2:15?").

The task of answering questions is complicated by the fact that we want the inference system to compensate somewhat for the inevitable gaps in its knowledge. For instance, suppose that lathe #45 was available for use at 2:01. What can we say about its availability at 2:02? We cannot say anything with certainty, but, barring information to the contrary, it is reasonable to suppose that the lathe is still available at 2:02. In general, we assume that a proposition having become true tends to persist unless something is determined to make it false. We do not require that the system be told exactly what propositions hold over what intervals of time; in fact, we would rather that the system be told only when a proposition becomes true and let it infer how long the proposition remains true. The system then is responsible for keeping track of what propositions are true over what intervals of time.

Rules like "If *event* occurs at time t, then *proposition* becomes true at $t + \epsilon$." are referred to as *causal rules*. The inference system is responsible for computing the consequences of events as directed by a set of causal rules and then using those consequences to determine the persistence of facts for processing queries. For instance, suppose I am told that lathe #45 suffered a major malfunction at 4:30 this afternoon and, further, that major malfunctions make machines unavailable for use. In this case, I would expect the query "Is there a lathe available for use at 2:15?" to succeed, but the query "Is there a lathe available for use at 4:45?" to fail.

The semantical issues concerning logical theories comprised of such rules are discussed in [40] and are beyond the scope of this article. The procedural semantics are rather simple and intuitive: Starting with some set of propositions known to hold at some earliest point, you sweep forward in time considering events known to occur and using the causal rules to determine which propositions become true, continue to persist, and cease to persist. In [16] and [17], we discuss the basic algorithms for performing this sort of causal reasoning. These algorithms ensure that the database contains accurate information concerning what propositions are true over what intervals of time.

All of the algorithms described in [17] run in time polynomial in the size of the input (i.e., the specific and general knowledge provided by the application). However, even low-order polynomial performance (e.g., $O(n^3)$) can be prohibitive in applications involving a great deal of information; ideally, we want (small) constant-time performance for routine question-answering. Fortunately, temporal reasoning problems have a great deal of structure to exploit in attempting to achieve this sort of performance. The structure that we exploit in this paper is of two types: global structure (e.g., conventions for partitioning time according to calendar boundaries) and local structure (e.g., relationships between events and their consequences). Whenever an event is added or deleted, or its duration or time of occurrence is changed, it is possible that the entire contents of the database have to be revised. We limit the requisite computations by keeping track of the day, month, and year in which events occur, and how various events are related to one another. When a transaction occurs that affects a particular event, we first determine the changes required with regard to the set of events occurring on the same day or directly related. If the changes can be limited to that set of events, then we make the necessary changes and stop; otherwise, we broaden the set of possibly affected events by considering events in adjoining days and those related less directly.

The global structure inherent in our use of calendars and clocks is hierarchical in nature, and thus forms an ideal basis for organizing our knowledge of events. We exploit this hierarchical structure in our use of data structures for storing and retrieving information about events. In addition, we allow the user to specify general relationships among events that are temporally related. These relationships are used by the system to derive additional information to reduce computational costs. The actual algorithms and data structures used to expedite various sorts of temporal inference are the primary subject of this paper. The details are at once more and less complicated than hinted at above. More complicated in that we have to deal with partially ordered events and constraints that only bound the duration and occurrence of events. Less complicated in that we simply assume that someone supplies the system with the knowledge that serves as guidance in searching among events.

This paper describes a set of techniques that have been integrated into the temporal database management system of [17] to achieve reasonable performance in applications involving a large number of events that are known, or can be predicted, to occur and a large number of propositions that might persist over intervals of time. In order to set this work in perspective, we begin by considering some related work.

2. *Related Work*

There is a large body of work on temporal representation and inference, most of which can be divided into one of three subareas: extensions to the relational database model, logics of time for reasoning about programs and hardware verification, and approaches in artificial intelligence for reasoning about planning and language comprehension. In this section, we consider related work in each of these three subareas with regard to the following requirements for a temporal inference system:

(1) It must be capable of representing and reasoning about a variety of temporal constraints both quantitative and qualitative.
(2) It must be capable of inferring the consequences of events given some specification of cause-and-effect relationships.
(3) It must be capable of reasoning with incomplete information including partially ordered events and persistence.

This section is not intended as a broad survey of the literature. For a more complete survey of research in temporal reasoning, the interested reader is encouraged to consult [4]. We begin by considering extensions to the relational database model.

Following [42], we define a hierarchy of temporal database types in terms of their expressive power. *Static databases* are conventional databases, with no temporal information. *Static rollback databases* are static databases with a log indicating what transactions were performed and at what time. *Historical databases* are static databases, extended to maintain information about the time during which the stored information is assumed to be valid [9]. Finally, there are *hybrid temporal databases* that encode both transaction time and valid time in order to support the functionality of static rollback databases and historical databases; TQUEL [42] and the work described in [30] are databases of this type.

The extensions to the relational model described above fail to satisfy all three of our requirements. First, the underlying representation of time is too weak for our purposes (e.g., all times are given with respect to a global clock, making it impossible to represent information such as "A occurs before B"). Second, there is no inference mechanism capable of reasoning about cause-and-effect relationships. Third, there is no attempt to deal with incomplete information; the user of a historical database must specify exactly over what intervals each relation is valid. The ability to reason about transaction time is useful for many purposes (e.g., restoring the database to some previous state), but we do not consider it further in this paper.

In contrast with research extending the relational database model, research on logics of time has ignored practical issues for the most part, concentrating instead on the expressiveness of the logics. For all the power these languages afford—they all contain either propositional logic [37] or first-order quantified logic [1] as a subset—they do not provide machinery for simple default reasoning about persistence. Computational issues are generally overshadowed by concerns about completeness and decidability. To provide some idea of the cost of adopting a reasonably expressive logic of time, the validity problem for almost any nontrivial model of time in an interval-based modal logic of time is at least undecidable, and many such logics have no finite axiomatization [23]. It should be noted that the inference procedures underlying the system described in this paper are not complete with respect to propositional logic; we have traded completeness for performance in an attempt to achieve a reasonable balance.

The work by McCarthy and Hayes on the *situation calculus* [33], a first-order logic in which time-variant propositions are modeled as functions called *fluents*, uncovered what is referred to as the *frame problem*: the problem of inferring what things do not change as a result of an event occurring. The solution to the frame problem that McCarthy and Hayes suggested, using what are called *frame axioms*, is still the best-known method of dealing with persistence in temporal logic. Frame axioms explicitly state for each event and for each fluent whether or not the fluent persists in the situation resulting from the occurrence of that event. In applications involving concurrent actions, frame axioms are required for all combinations of events that might possibly co-occur. The method used in this paper for dealing with persistence involves the use of default reasoning [32, 38] and derives from ideas introduced in McDermott's temporal logic [35]. The introduction of default reasoning complicates the logical issues of reasoning about time [24, 40], but actually simplifies many of the computational issues [17, 27].

The requirement that a temporal inference system be capable of reasoning about a variety of constraints implies a language for specifying constraints involving temporal entities (points or intervals) and some method of inferring additional constraints from those explicitly supplied. For almost any language that allows negation and disjunction, the decision problem of determining if a particular constraint follows from some set of constraints is at least *NP-hard* [11], and, hence, trade-offs are in order for building practical inference systems.

In [2], Allen describes a calculus for reasoning about the relationships between intervals. He represents intervals and the relations among them by means of a graph. The nodes of the graph correspond to intervals, and the arcs are labeled with the relations between them. The relations between intervals can be disjunctions of the primitive relations, (e.g., interval i_1 is before or after i_2 (written i_1 [BEFORE AFTER] i_2)). Ladkin [28] shows that Allen's interval calculus has a first-order axiomatization that is complete and decidable. Unfortunately, computing the transitive closure of the relations defined in such a network is an *NP-complete* problem [46]. Allen's approach to this problem is to use a decision procedure that is not complete, in the sense that it may not generate all the relations that can be derived from a particular graph. Vilain [46] restricts Allen's representation, and then provides a polynomial algorithm that will derive *all* the relations entailed by a particular graph. The restricted representation eliminates certain disjunctions of relations, specifically those that cannot be expressed as disjunctions involving a single endpoint. For example, "interval i_1 ends during, concurrently with, or after interval i_2" can be expressed, while "interval i_1 is either wholly before or wholly after interval i_2" cannot.

In most practical systems, only very limited use of negation and disjunction are allowed. It makes little sense to provide language constructs that cannot be efficiently supported. Representing inexact metric information does not introduce any insurmountable problems. It is straightforward to implement a polynomial-time decision procedure for determining the best bounds on the distance separating a particular pair of points in time, given a set of constraints bounding the distance between pairs of points [11]. The method for representing inexact metric information used in the system described in this paper is a generalization of Kahn's plus/minus error intervals for event dates [25].

Finally, we consider some approaches that have been developed in artificial intelligence for applications in planning, language comprehension, and decision support.

CHRONOS [5] is an example of a system designed to answer questions involving time, given information extracted from natural language input. From our perspective, the most interesting aspect of this work is its recognition of the importance of reasoning about incompletely specified knowledge. Making *appropriate* inferences in the absence of complete knowledge is perhaps *the* central problem in reasoning about time in artificial intelligence, and the frame problem has served as a focus in recent years.

Of the proposed solutions to the frame problem, perhaps the best known is built into the STRIPS representation of action [22]. STRIPS is a planning system based upon McCarthy's situation calculus [33] and Simon and Newell's theory of problem solving [21]. In STRIPS, each action (event) has a set of preconditions (fluents that must be true immediately before the action is carried out), a set of additions (fluents that must be true immediately after), and a set of deletions (fluents that must be false immediately after). STRIPS deviates from the situation calculus in not requiring any frame axioms. Persistence is handled using a particularly simple default rule of inference: If a fluent is not mentioned in the set of additions or deletions of a given event, then its status immediately following the event is the same as its status immediately preceding the event. The precise semantics of STRIPS is somewhat more complicated [29], but this simple default rule is at the heart of most computational approaches to dealing with the frame problem.

There is a class of inference systems that have been designed to reason about the consequences of events and actions whose order is not completely known. We refer to these databases as *time maps*, after [35]. The systems that maintain these databases we call *time map managers* [7]. The temporal inference system that is the subject of this paper is a time map manager, and, henceforth, we refer to it as TMM.

The earliest time map managers were based on an abstraction for representing plans in terms of partially ordered *tasks*: descriptions of actions being considered by a robot planner for execution. The time map in this case is simply a network of tasks annotated with predicted consequences of tasks [34, 39, 43]. In most cases, the underlying representation of action was based upon the situation calculus, and, hence, it is assumed that the tasks in the network will ultimately be totally ordered. Actually computing the consequences of a partially ordered set of tasks turns out to be quite complicated. For any reasonably powerful representation of action, determining what will be true in all (some) total orders consistent with a particular partially ordered task network is *NP-hard* (*NP-complete*) [6]. In [16], we describe a polynomial-time algorithm for computing the consequences of partially ordered tasks that is sound, but not complete.

SIPE [47] also maintains a network of partially ordered tasks, but does not require that all tasks eventually be totally ordered. Plans generated by SIPE may involve parallel actions, where interactions between actions occurring at the same time are mediated through a restricted form of resource modeling. SIPE's representation of action is quite general [48]. The effects of actions are modeled using a variant of the STRIPS representation supplemented with *domain rules* to deduce additional, context-dependent, effects of a particular action.

DEVISER [44] employs a partially ordered network of events, but with some added machinery for reasoning about metric constraints. Events other than the agent's actions can be represented. Events may have a finite duration, which can be computed from the context in which they occur, and they can occur simultaneously. While duration, once computed, is fixed, event starting times are represented by *windows*, similar to Kahn's plus/minus error intervals, which give ranges within which a particular event will occur. Windows are defined in terms of a global clock, and the only local relation allowed is to define two events as *consecutive*, meaning the beginning of the later event immediately follows the end of the earlier event. The action representation used by DEVISER is less general than STRIPS (e.g., the propositions deleted by an event an explicitly included in the preconditions for that event to occur).

The TMM may be viewed as a restricted implementation of McDermott's temporal logic, in the same way that STRIPS is a restricted implementation of the situation calculus. In the TMM, the classical database assertion is replaced by the notion of *time token* corresponding to a particular interval of time during which a general *type* of occurrence (a proposition or event) is said to be true. In the TMM, propositions and events are encoded as atomic formulas with no variables. The next section describes the capabilities of the TMM in greater detail.

Given that reasoning about time is critical for many applications, it is not surprising that many techniques have been suggested for simplifying the requisite computations. The basic idea of partitioning time in order to simplify updates and expedite question answering has been exploited by many researchers. Kahn and Gorry emphasize the need to organize events using *reference events*, before/after chains, and historical periods [25]. Allen discusses the use of *reference intervals* as a way of constraining both the space required for his graph of interval relations, and the time required to compute its closure [2]. The intuition behind Allen's reference intervals is that most of the propositions we want to represent will be true over intervals that can be linked together based on a hierarchy of local referents (e.g., the things I did today, the bills I paid this month, or the vacation I took last year). Koomen [26] describes a system that automatically generates reference intervals to limit the computation required during constraint propagation. Koomen's approach is similar to that taken in this paper in that both attempt to derive additional constraints in order to limit computation; however, Koomen does not deal with metric information, nor does he deal with persistence or the problems that arise in reasoning about cause-and-effect relationships. Malik and Binford suggest the use of *reference frames* to simplify deriving implicit constraints [31]. Their basic approach is similar to that of Kahn and Gorry, except that Malik and Binford assume that all temporal constraints are expressed as linear inequalities and all temporal inference is carried out using linear programming methods; they do not provide any details about how reference frames might be used to support inference in large databases. In an earlier paper [12], we consider a method similar to that of Malik and Binford that relies on keeping a *kernel* of events for which constraint propagation is optimized. We abandoned this approach when it became apparent that it could not be easily automated.

The techniques described in this paper are the culmination of trying many different methods, and finally settling on a couple of techniques that are useful across many domains and that can be integrated into all of the different algorithms that underlie our system. The emphasis in this paper is on efficiently supporting a rich set of inferential capabilities in the presence of large amounts of temporal knowledge. Before discussing our techniques, we consider the basic operations of the TMM in somewhat greater detail.

3. *Temporal Database Management*

This section provides a description of the TMM both from the user's point of view, and from the designer's point of view; the former to provide the reader with a better idea of what functionality is supported by the TMM, and the latter to provide the reader with the basic information needed to understand the indexing algorithms that are central to this paper.

A time map is a graph. Its vertices refer to *points* (or *instants*) of time. One point is related to another using *constraints* where a constraint is represented as a directed edge linking two points. Each edge is labeled with an upper and lower bound on the distance separating the two points in time. These bounds allow us to represent incomplete information concerning the duration and time of occurrence of events (e.g., unloading the truck will take between 20 and 25 minutes). Any two points can be related by finding a path from one point to the other, where a path from pt_0 to pt_n is just a sequence $pt_0 c_1 pt_1 \cdots c_n pt_n$ such that pt_0 through pt_n are points and c_i is a constraint relating pt_{i-1} to pt_i. An *interval* is just a pair of points.

An application program operates on *events* (i.e., propositions corresponding to actions and other phenomena that tend to precipitate change in the world) and *fluents* (i.e., propositions corresponding to states of the world that, once they become true, tend to persist in time). Events *cause* fluents to become true or false. The application program is expected to provide general knowledge about cause-and-effect relations (i.e., causal rules), and specific knowledge about events and fluents. The system then uses the causal rules to infer additional information about events and the intervals of time over which fluents persist.

The distinction between events and fluents is not reflected in the underlying data structures of the TMM. Instead, there is a single data structure referred to as a *time token* (or simply *token* where it should cause no confusion) used to encode information about both events and fluents. A time token consists of an interval (i.e., a pair of points) together with a *type*, where a type corresponds to a propositional formula (e.g., `(location obj73 loc14)` or `(move obj73 loc14 loc17)`). Asserting `(occurs` *type tok*`)` serves to create a new time token whose name is *tok*. The expressions `(begin` *tok*`)` and `(end` *tok*`)` denote the begin and end points of the interval associated with the time token *tok*.

New constraints are added to the time map by making assertions of the form `(elt (distance` pt_1 pt_2`)` *low high*`)` (`(elt` *q low high*`)` indicates that the quantity *q* is an "element of" the closed interval *low* to *high*, with additional syntax required for specifying open or half-open intervals), which serves to add an arc labeled with bounds *low* and *high* between the two points pt_1 and pt_2 in the time map. To illustrate, asserting `(occurs (routine-service lathe14) service1)` creates a token whose type is `(routine-service lathe14)` and whose name is `service1`, and asserting `(elt (distance (begin service1) (end service1)) 15 20)` constrains the token to last between 15 and 20 minutes.

The TMM assumes a privileged global frame of reference and a scheme for partitioning the global time line. We return to partitioning schemes and global frames of reference in Section 5; for now, it is enough to get some idea of how these ideas appear in the user interface. Specifying constraints with respect to the global frame of reference is made particularly easy using *dates*. A *date* is just an offset from the global frame of reference specified in terms of the current partitioning scheme. For instance, if the partitioning scheme is specified in terms of weeks, days, hours, and minutes, the date `(date (weeks 2)(hours 1)(minutes`

`15))` is converted into an offset from the global frame of reference with divisions not mentioned (e.g., `days`) defaulting to 0. Often, it is convenient to specify a default date (e.g., noon today) and then specify offsets, called *reldates*, relative to the default. Dates and reldates can appear anywhere in a formula that a point can. As an example, if the default date is noon today, asserting `(elt (distance (begin service1)(reldate (hours 2)(minutes 30))) -10 10)` determines that `service1` begins between 2:20 and 2:40 this afternoon.

Specific knowledge is entered in the form of time tokens and constraints. General knowledge is entered in the form of causal rules. A causal rule of the form

`(project` *antecedent-conditions trigger-event delay consequent-effects* `)`

indicates that whenever an event of type *trigger-event* occurs, and the *antecedent-conditions* are true at the outset of the interval associated with *trigger-event*, then the *consequent-effects* are true after an interval of time determined by *delay*. The trigger event is specified as a type, the antecedent conditions and consequent effects are specified as types or conjunctions of types, and the delay is specified as a pair consisting of a lower and an upper bound on the time between the end of the trigger event and the manifestation of the effects.

The TMM is primarily propositional in its operation. The types specified in time tokens do not, for instance, contain variables. There are, however, restricted forms of quantification allowed. Causal rules are implicitly quantified over all time tokens. A restricted form of quantification is also allowed in the specification of types appearing in causal rules. Universal variables are notated `?var` and their scope is the entire formula in which they appear. As an example of an explicitly quantified causal rule, consider

```
(project (and (location ?object ?location1)
              (operational-status ?machine in-service)
              (instance-of ?machine robot-forklift))
         (transfer ?machine ?object ?location1 ?location2)
         [(/ (distance ?location1 ?location2)
             (max-speed?machine))
          (/ (distance ?location1 ?location2)
             (min-speed?machine))]
         (location ?object ?location2))
```

This rule states that, whenever a transfer is made by a robot forklift, the transferred object will appear in the new location after a delay determined by the distance to be traveled and the minimum and maximum rate of travel allowed by the forklift.

Reasoning about time in the TMM consists of scanning the time map in order to determine how one event is related to another and what might be true during, before, or after an event. One of the important characteristics of the data structures used by the TMM to encode temporal information is that they can be updated incrementally. That is to say, as new events and facts are added, only those parts of the time map that are affected need be changed. This is accomplished by using *data dependencies* [20] to keep track of why various modifications were made and under what circumstances those modifications continue to be appropriate. When the user specifies changes to the time map, the TMM propagates those changes using the data dependency information to determine what additional changes have to be made. The sort of graph scanning and updating that the TMM engages in is referred to as *reasoning about time from the side* [35]. It's as though all of what you know about the past, present, and future is laid out in front of you. An example should help to strengthen this intuition.

Figure 1 shows a particularly simple time map. Tokens are represented with a vertical bar indicating when the corresponding interval begins, and either a second vertical bar providing some indication of when the interval ends, or an arrow $\longrightarrow$ indicating that the end of the interval is far enough in the future that it cannot be drawn in the diagram. The delimiters for tokens are connected by a horizontal bar (e.g., $\longmapsto$). Each token is labeled with a formula denoting its type. The tokens are laid out on the page so as to indicate their relative offset from some global reference point. In cases where a token is not completely constrained with respect to the reference point, we use the notation $| \sim\sim\sim\sim | \text{———} |$ where the first two vertical bars indicate the earliest and latest that the corresponding interval might begin and the distance separating the second and third vertical bars provides some indication of the duration of the interval. In Figure 1, there are three tokens corresponding to events: one of type `(malfunction lathe14)`, a second of type `(routine-service lathe14)`, and a third of type `(manufacture part42 job321)`. There are also a number of tokens corresponding to fluents that describe the production status (whether a machine is currently in use or not) and operational status (fitness for work) of two machines, `lathe14` and `lathe73`, in an automated factory.

Tokens corresponding to fluents (e.g., `operational-status lathe14 in-service)`) are referred to as *persistences* [35]. It is assumed that the user does not constrain the lower bound of the distance separating the begin and end of persistences; determining the duration of persistences is handled by the TMM. For any type *P*, *P* and `(not` *P*`)` are said to be *contradictory types*. The user is also allowed to specify contradictory types. For example, the expression `(contradicts (location := :not=))` indicates that any two tokens of type `(location` *arg*1 *arg*2`)` are contradictory if their first arguments are equal and their second arguments are different. If two tokens with contradictory types (e.g., `(location obj73 loc14)` and `(location obj73 loc9)`) are constrained so that one begins before the other and the earlier could persist longer than the beginning of the later, then the two are said to be *apparently contradictory*. The TMM resolves apparently contradictory tokens by forcing the end of the earlier to precede the beginning of the later. In Figure 1, the token of type `(operational-status lathe14 out-of-service)` was constrained by the TMM to end before the beginning of the token of type `(operational-status lathe14 in-service)` in order to resolve an apparent contradiction indicated by the order of the two tokens and the assertion `(contradicts (operational-status := :not=))`.

Keeping a large database free of apparent contradictions requires that it be cheap to determine the relative ordering of selected points. Furthermore, the database must be organized so that the system does not waste effort checking on pairs of tokens that cannot possibly overlap (e.g., the fact that a machine broke down for an hour last month need not enter into reasoning about whether or not it will be operational this afternoon). Section 5.2 describes how the TMM handles the problem of eliminating apparent contradictions in large time maps.

Now we turn our attention to the problem of answering questions about the information stored in the time map. For the most part, query processing in the TMM is the same as query processing in Prolog [10] with certain changes in notation: Cambridge–Polish notation is used instead of prefix notation, conjunctions are

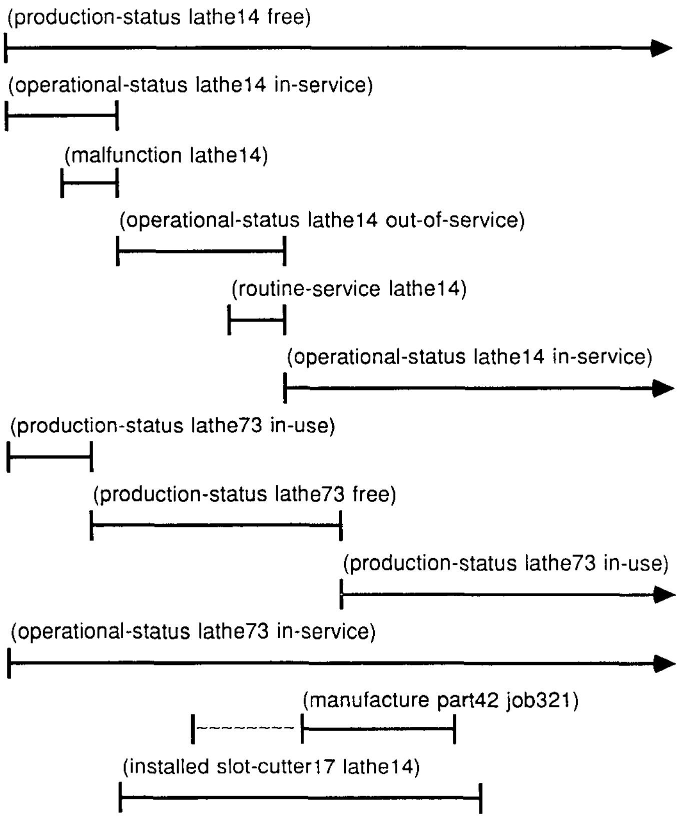

(production-status lathe14 free)
(operational-status lathe14 in-service)
(malfunction lathe14)
(operational-status lathe14 out-of-service)
(routine-service lathe14)
(operational-status lathe14 in-service)
(production-status lathe73 in-use)
(production-status lathe73 free)
(production-status lathe73 in-use)
(operational-status lathe73 in-service)
(manufacture part42 job321)
(installed slot-cutter17 lathe14)

FIG. 1. A simple time map.

notated `(and` $P_1 P_2 \cdots P_n$`)` instead of $P_1, P_2, \ldots, P_n$, and variables are notated `?var` instead of `Var`. Certain expressions, however, that are encountered in queries are treated specially by the TMM. We refer to these expressions as *temporal queries*. Temporal queries in the TMM must specify a *propositional schema* and a *fetch interval*. The fetch interval is just a pair of points. The propositional schema is either a type or a conjunction of types, and may contain variables. Variables in temporal queries are interpreted as existentially quantified with scope being the entire query. The operator `tt` (for true throughout) takes as arguments a point defining the beginning of the fetch interval, a point defining its end, and a propositional scheme. A query of the form `(tt` $pt_1\ pt_2$ `(and` $P_1 \cdots P_n$`))` is treated as a request to determine if `(and` $P_1 \cdots P_n$`)` is true throughout the interval corresponding to pt_1 and pt_2, or could be made so with additional constraints. To illustrate, consider the following database query:

```
(and (occurs (manufacture part42 job321) ?tok)
     (tt (begin ?tok) (end ?tok)
         (and (operational-status ?machine in-service)
              (production-status ?machine free)
              (instance-of ?machine lathe)
              (installed ?attachment ?machine)
              (instance-of ?attachment slot-cutter))))
```

The above query can be paraphrased as, "Is it true throughout the interval associated with the manufacture of `part42` for `job321` that an operational lathe outfitted with a slot cutting attachment is available for use?" In the time map of Figure 1, the above query would succeed with `?machine` bound to `lathe14`, `?attachment` bound to `slot-cutter17`, and the additional constraint that the manufacture of `part42` follow the task corresponding to performing routine maintenance on `lathe14`. Given a particular token, it is not all that hard to determine if it spans a particular fetch interval [17, 45]. The hard part of token retrieval is determining which tokens to consider. Section 5.1 describes a method of indexing tokens that avoids checking on tokens that could not possibly span a given fetch interval.

This section has provided an overview of the functionality supported by the TMM. One capability that has not been discussed involves keeping track of the reasons why decisions were made in order to detect when new information serves to invalidate the justification for past decisions. Using the TMM, an application program can make explicit its justifications for making decisions, and the TMM will then keep track of those justifications, detecting when they are no longer satisfied, and notifying the application program so that it can take remedial action if called for. The TMM extends the notion of *reason maintenance* in static databases [20] to handle temporally-dependent data [17]. In planning applications, the TMM's temporal reason maintenance system simplifies noticing potential conflicts between plans for achieving different tasks and assists in recovering from poor plan choices.

The indexing techniques described in the rest of this paper facilitate all of the basic inferential routines used in the TMM including those required for temporal reason maintenance. To keep the discussion to a reasonable length, we concentrate on just two inferential routines: resolving apparent contradictions and a simple form of retrieval basic to query processing. Other routines such as those employed in temporal reason maintenance and causal reasoning about partially ordered events (discussed in, respectively, [17] and [16]) employ variations on the same basic methods.

4. *Organizing Large Temporal Databases*

The most expensive operation distinguishing temporal database manipulations from those performed by static database systems (e.g., Prolog) involves finding tokens that satisfy certain temporal constraints. This operation, which we call *token retrieval*, is the temporal analog of fetching assertions in the database that match a given pattern. Token retrieval requires the system to search through the database for time tokens whose type matches a given pattern and whose associated interval spans a specified reference interval.

In this section, we consider two strategies for indexing and searching temporal information to expedite token retrieval. The first strategy involves the use of a data structure for storing tokens that allows for the efficient retrieval of tokens whose relative offset from a fixed global frame of reference can be accurately determined. In most applications, this global frame of reference corresponds to some reference with respect to the standard time and date. Specifying the occurrence of events with respect to a standard time line is so common in practice, that it has proven cost effective to build and maintain this data structure. There are situations, however, where the global frame of reference is not sufficient for relating pairs of events. For instance, in reasoning about a chemical process, you may not know exactly when a catalyst was added to a reactor vessel, but you do know that within

8 to 10 minutes following the addition of the catalyst the reaction was complete. The second strategy involves establishing local frames of reference (e.g., the beginning of the event corresponding to adding the catalyst), and keeping track of the best bounds on the relative offset of selected points (e.g., the end of the event corresponding to the reaction) from those local frames of reference. During token retrieval, these local best bounds assist in determining the order of pairs of points whose relative offset to the global frame of reference cannot be accurately determined.

We begin by considering how to extend standard techniques for discriminating on data to handle information with a temporal component. A *discrimination* corresponds to a question (or deduction) concerning the form or content of the data. On the basis of the answer to such a question, the data is usually partitioned into disjoint sets so that if a program attempts to retrieve data whose content depends on the answer to this question, then the system will know which sets to look in and which sets to ignore. This process of discriminating on data, asking questions and then partitioning according to the answer, can be thought of as caching the results of deductions that are likely to be frequently needed. As an example, by discriminating employee records on the basis of the first letter of the employee's last name, a great deal of time can be saved in looking up an individual employee's record. To be useful, discrimination should substantially reduce search with a minimum overhead (e.g., if most of your employees have last names that begin with the same first letter, then the above scheme would be a waste of effort). Not all discriminations can be depended upon to remain valid as the data changes over time (e.g., an employee might change his or her last name). Where the data is subject to change, there is an additional expense involved in keeping track of valid deductions.

The information content of a time token corresponds to the syntactic form of the token's type and the temporal extent (or *scope*) of the associated interval of time. We assume that the type of a time token does not change, and, hence, deductions corresponding to syntactic discrimination on the type of time tokens are never invalidated. All time tokens are indexed through what is called a *discrimination tree* or *dtree* [8]. Each nonterminal node in a dtree corresponds to a discrimination: A question whose answer determines that subtree various data items are stored in. Each terminal node in a dtree corresponds to a set (or *bucket*) of data items determined by the discriminations on the path leading from the root of the dtree to the terminal node. Whenever a new token is created, or the TMM detects some change in the temporal extent of an existing token, the system makes sure that the token is stored in an appropriate bucket of the existing dtree.

Discrimination in the TMM is based on demand. If the size of a bucket exceeds some fixed threshold, then the TMM will attempt to subdivide the bucket by adding an additional discrimination node and some number of terminal nodes as dictated by the chosen partitioning scheme. Generally speaking, if it is possible, partitioning a bucket of tokens is based upon a syntactic discrimination according to the types of the tokens stored in the bucket. There are situations in which further syntactic discrimination is undesirable (e.g., for certain queries it is useful to lump all tokens of the form `(location` *arg*1 *arg*2`)`, where *arg*1 is fixed and *arg*2 is allowed to vary, in the same buckets). If further syntactic discrimination is either impossible or undesirable, the system attempts to discriminate on the basis of the temporal scope of tokens.

Temporal discrimination in the TMM involves choosing a temporal partitioning scheme and subdividing the overly large bucket of tokens according to this scheme.

In the TMM, the application program is required to supply a set of hierarchically arranged temporal partitions of a timeline (i.e., the real numbers) such that all of the partitions can be related via a single global frame of reference (i.e., 0). Attempts to *derive* an adequate partitioning scheme solely on the basis of the current contents of a bucket have proven difficult [36]. The partitioning scheme chosen must essentially anticipate the sort of questions that will frequently be asked during token retrieval. In the factory domain, the partitions supplied by the FORBIN planner [19] correspond to weeks, days, eight-hour work shifts, and one-hour intervals. The system discriminates as demand dictates, starting with the coarsest partitions and refining only as required.

The scope of time tokens corresponding to tasks and their effects are liable to change frequently in the course of planning. For this reason, it is useful to prevent discriminations on certain tokens below a specified level of precision. For example, in the factory domain you may be committed to performing a particularly expensive overhaul on a machine sometime in the coming year. You may also tentatively agree to performing the overhaul in June, but this is likely to be overridden as new information becomes available. The TMM makes it possible to say that, with regard to a given token, make no temporal discriminations finer than a given partition size. If further information induces the planner to make stronger commitments (e.g., the machine starts showing signs of imminent breakdown forcing you hasten the overhaul), then it is expected that the planner will allow finer discriminations. Each node in the dtree corresponding to a temporal discrimination consists of a list of partition/subdtree pairs and a list of tokens that cannot be further discriminated upon. Since the partitions ar totally ordered it is easy to keep the list of partition/subdtree pairs sorted. Noticing that a token can be further discriminated upon or is inappropriately discriminated upon is done using a data dependency mechanism similar to that described in [17].

The partitioning scheme described above corresponds to a set of successively finer partitions of time with respect to a single clock. This is enormously useful for precisely reasoning about many kinds of events (i.e., we often know the exact time of events with respect to a standard date and time), and serves as a coarse-grained filter for most others (i.e., even if we don't known the exact minute that an event occurred, we can often pin down the day, week, or month). There are, however, events whose exact times of occurrence are not known with respect to the global frame of reference, but whose relative offsets can be determined with precision.

The TMM provides a mechanism for specifying hierarchies of event relations that can serve to guide search in determining temporal orderings among events that are not precisely known with respect to the global frame of reference. The most common strategy for exploiting the TMM's search machinery involves the use of the *event/subevent hierarchy* (also referred to as a *task/subtask hierarchy* when the events correspond to tasks). In many applications, it is useful to reason about events at multiple levels of detail. For instance, in specifying a complex machining task, an application program might start by creating a token, `task1`, of type `(manufacture part42 job321)`, and then roughly scheduling the token to occur in a time slot during which a critical resource appears to be in ample supply. Later, the application program will want to specify further details about how to carry out the machining task. In particular, the program might create a token, `task2`, of type `(setup machine45 operation132)` corresponding to a task performed as part of machining `part42` for `job321`. The event corresponding to `task2` may have its own subevents providing yet more detail and serving to establish a hierarchy of events. By specifying that `task2` is a subevent of `task1`

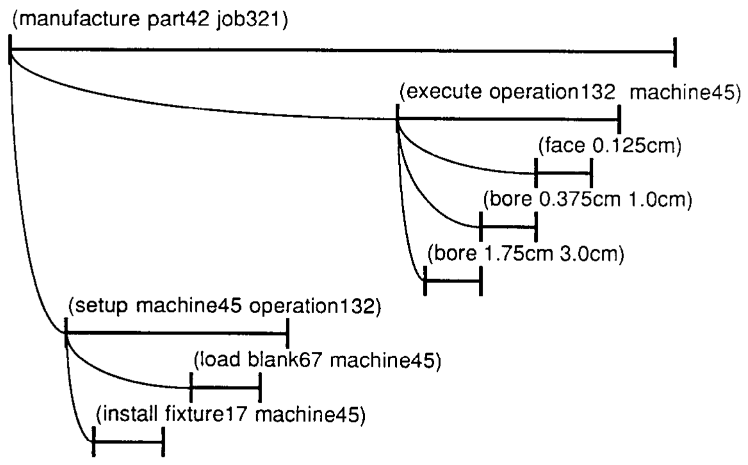

(manufacture part42 job321)
(execute operation132 machine45)
(face 0.125cm)
(bore 0.375cm 1.0cm)
(bore 1.75cm 3.0cm)
(setup machine45 operation132)
(load blank67 machine45)
(install fixture17 machine45)

FIG. 2. Hierarchically organized events.

(i.e., asserting `(subevent task2 task1)`), the application program instructs the TMM to keep careful track of certain point-to-point distance estimates. If e_1 is specified as a subevent of e_2, then the TMM can guarantee (within certain limitations) that there exists an edge in the time map connecting the beginning of e_1 and the beginning of e_2 labeled with the best bounds on the distance in time separating the two points.

The labels on these edges are maintained by the same algorithm used in [17] to ensure correct behavior with regard to the addition and removal of information. Figure 2 shows a simple hierarchy of depth 3, indicating the edges whose labels are maintained by the TMM. The token retrieval machinery takes advantage of these special edges to speed search in determining the relative ordering of tokens that are not distinguished in the dtree by temporal discriminations. These combined searching and caching techniques guarantee that under certain conditions (see Section 6) the machinery for determining the bounds on the distance between two points will always return the best bounds and will do so in time proportional to the depth of the hierarchy. For most problems, the depth of the hierarchy is seldom greater than 10 and the alternative exhaustive search would cost on the order of n^3 where n is the total number of tokens in the partition (often on the order of several hundred). Section 5.3 describes how the TMM performs the necessary caching and search required to implement the above scheme.

Token retrieval routines use the dtree to provide a set of candidate tokens and then determine the relative ordering of the beginning of these tokens using the search methods described in the previous paragraph. Determining the duration of a fact token relative to a reference interval is also accomplished using search methods that exploit the hierarchical partitions and cached distance estimates. All the search routines return the information requisite for setting up appropriate data dependencies. Searches corresponding to different token retrieval requests can be coroutined to support efficient backtracking during backward chaining.

5. *Algorithmic Details*

The TMM employs a heuristic graph traversal routine to compute bounds on the distance separating pairs of points in the time map. These bounds are used to determine relations between pairs of points and intervals. Estimates of the distance between pairs of points are computed by finding paths through the network of constraints. Recall that a path in the time map is a sequence of points and directed edges corresponding to constraints. Each edge c is labeled with an upper and lower bound, denoted low(c) and high(c) respectively. For each path $p = pt_0 c_1 pt_1 \cdots c_n pt_n$, we have bounds($p$) = $\langle low, high \rangle$ where $low = \sum_{i=1}^{n}$ low(c_i), and $high = \sum_{i=1}^{n}$ high(c_i). In computing the best bounds, the heuristic graph traverser tries to find the paths with the greatest lower and least upper bounds. The heuristic graph traverser is guaranteed to search all paths of length less than a specified constant, *max_path_length*, and is guaranteed to compute the best bounds if they can be determined from paths of length less than *max_path_length*. The details of the TMM's graph traverser are described in [17], and will not be repeated here. Additional complications that arise as a result of ideas introduced in this paper are treated in Appendix A. For a discussion and analysis of existing constraint propagation techniques for applications in artificial intelligence, see [11].

In order to expedite token retrieval, the TMM precomputes and caches certain point-to-point distance estimates. Whenever a constraint is added or removed, the system has to determine how the change affects the current set of point-to-point distance estimates. The process of updating point-to-point distance estimates is performed by propagating constraints through the time map using the heuristic graph traverser. The TMM relies upon the assumption that the best bounds on the distance separating any two points in the time map can be computed from a path of length less than *max_path_length*. For the planning applications we have worked with, setting *max_path_length* to 20 is sufficient to satisfy the assumption. Bounding the search depth of the heuristic path traverser is a concession to complexity; the general problem of computing the best bounds can be performed in $O(n^3)$ time where n is the number of tokens in the time map (on the order of 10^3). If we wish to add or delete a constraint between two points pt_1 and pt_2, the TMM's propagation routines run in $O(m^3 \log m)$, where m is the number of points reachable from pt_1 or pt_2 by paths of length less than *max_path_length*. With regard to our experience in planning, m is generally on the order of 10^2.

It would be wasteful to cache distance estimates for all pairs of points. Selective caching, on the other hand, can provide real benefits. In Section 5.1, we see how caching estimates of the distance between each point and the global frame of reference forms the basis for an effective temporal discrimination scheme. In Section 5.3, we see how a strategy for caching distance estimates between points corresponding to related tokens can expedite search within portions of the time map that are not highly constrained with respect to the global frame of reference.

5.1 INDEXING TIME TOKENS IN A TEMPORAL DISCRIMINATION TREE. In the following, let R denote the set of real numbers, and $[a, b)$ the interval defined by $\{x \mid x \in R, a \leq x < b\}$. For our purposes, a *partition* $\mathscr{P}$ of an interval I (subset of R) is just a set $\{[x, y) \mid x, y \in I\}$ such that $I_1 \cap I_2 = \varnothing$ for all distinct I_1 and I_2 in $\mathscr{P}$, and for each $x \in I$ there exists an I' in $\mathscr{P}$ such that $x \in I'$. A *hierarchical partitioning scheme* consists of a sequence of partitions $\mathscr{P}_0 \mathscr{P}_1 \cdots \mathscr{P}_n$ of R such that for each $i < n$ if $I \in \mathscr{P}_i$ then there is a set intervals in $\mathscr{P}_{i+1}$ that partitions I. In addition, we insist that $\mathscr{P}_0$ is always the set consisting of just R. $\mathscr{P}_i$ is said to be more *restrictive* than $\mathscr{P}_j$ with respect to I just in case $I \in \mathscr{P}_j$ and there is a subset $\mathscr{S}$ of $\mathscr{P}_i$ such that $|\mathscr{S}| > 1$ and $\mathscr{S}$ partitions I. A hierarchical partitioning scheme is *strict* if it is the case that for any i such that $0 \leq i < n$, if $I \in \mathscr{P}_i$, then $\mathscr{P}_{i+1}$ is more restrictive than $\mathscr{P}_i$ with respect to I. All the partitioning schemes we will be looking at in this paper are strict.

A *time-line partition* is just a partition of R such that 0 is identified with a particular frame of reference (e.g., midnight on April 30, 1777, the birthdate of Karl Friedrich Gauss) and each real number x corresponds to an offset in time from this global frame of reference as measured by a particular clock. The global frame of reference simplifies internal bookkeeping and provides a basis for using dates in specifying constraints. Figure 3 shows part of a strict hierarchical timeline partition in which the partitions consist of days, shifts (eight-hour periods), half-shifts (four-hour periods), and one-hour periods offset from a fixed zero point. In Figure 3, C is contained in A and B, and is partitioned by $\{D, E, F, G\}$.

Every point is identified with a tuple $\langle low, high \rangle$, called its relative offset, indicating the best (lower and upper) bounds over all paths through the network of constraints on the distance in time separating the point from the global frame of reference. Relative offsets are updated during constraint propagation. The constraint propagation routines ensure that, if additions or deletions to the set of constraints require that the relative offset of a point be updated (i.e., the bounds made either more or less restrictive), then the system can easily detect this and respond appropriately. Tokens are indexed in the discrimination tree using the relative offsets of their begin points. To simplify the discussion of temporal indexing, we assume that all tokens are syntactically indexed down to atoms according to type, and consider only those nodes in the discrimination tree corresponding to temporal indices. A temporal index is implemented as a data structure called a TBUCKET consisting of a set (possibly empty) of tokens that cannot be further discriminated upon, a partition interval $[a, b)$, and a set of subindices (i.e., TBUCKETs) sorted by their associated partition intervals. For efficiency reasons, subindexing is usually postponed until the set of tokens stored at an index exceeds some fixed threshold. Figure 4 shows a simple hierarchical partitioning scheme and a table indicating the relative offsets for five tokens of the same type. Each token is shown in the hierarchical partition indicating the interval of the most restrictive partition that necessarily contains the beginning of the token. Figure 5 shows a portion of a discrimination tree. (Recall that $\mathscr{P}_0$ is the singleton set corresponding to the entire timeline.) Note that T_5 could be further discriminated, but is not in this case since the TBUCKET containing it would otherwise be empty. If the relative offset of the beginning of T_2 was changed from $\langle 1.2, 2.7 \rangle$ to $\langle 1.2, 1.8 \rangle$, then T_2 would be further discriminated ending up in the same bucket with T_3. If, on the other hand, the relative offset was changed to $\langle 1.2, 3.5 \rangle$ or $\langle 1.2, +\infty \rangle$, then T_2 would end up (respectively) in the same bucket with T_1 or in the top-most index corresponding to $\mathscr{P}_0$.

Now we can describe the algorithm used in the TMM for token retrieval. Recall that token retrieval is invoked by specifying a type P and a fetch interval pt_1 to pt_2. The task is to identify all tokens of type P that might possibly persist throughout the fetch interval (i.e., all tokens of type P that could begin before pt_1 and persist at least as long as pt_2). We assume that P contains no variables. The basic intuition underlying the algorithm is simple. First, using the heuristic graph traverser, consider all those tokens whose relative ordering with respect to pt_1 and pt_2 cannot be determined using offsets from the global frame of reference. Second, using the temporal discrimination tree, search backward in time from pt_1 considering all tokens until you find one that has been clipped by a contradictory token. We refer to Figure 6 in explaining how the token retrieval algorithm searches the tokens

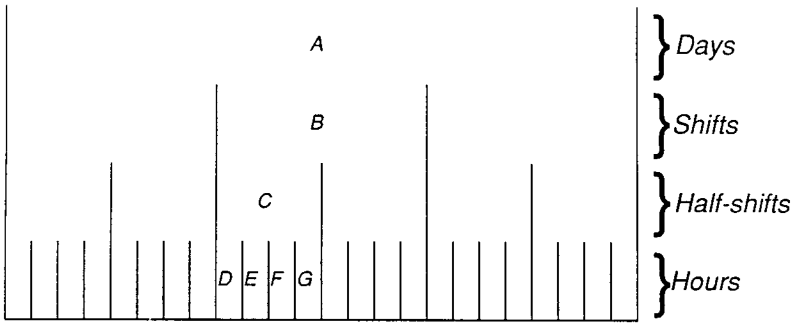
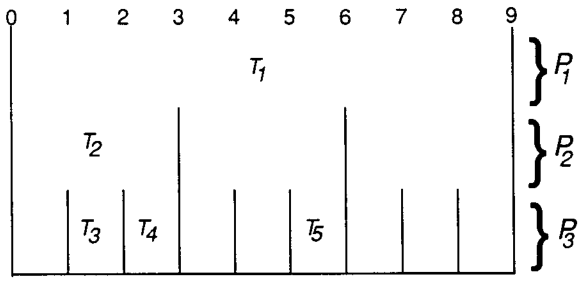


FIG. 3. Portion of a strict hierarchical time-line partition.



	T_1	T_2	T_3	T_4	T_5
low	2.0	1.2	1.5	2.4	5.0
high	7.5	2.7	1.5	2.7	5.5

FIG. 4. Data for demonstrating temporal indexing.

stored in the time map; think of the areas laid off by the partitioning scheme as sets of tokens to type *P*. The steps in the algorithm are:

(1) Determine a partition $\mathscr{P}$ and an interval *I* belonging to $\mathscr{P}$ such that the fetch interval is constrained to (necessarily) begin during *I* and cannot be shown to (necessarily) begin during any interval belonging to a partition more restrictive than $\mathscr{P}$.

(2) Using the heuristic graph searching routines, consider each token whose associated partition interval either is contained in *I* or contains *I*. In Figure 6, the shaded areas correspond to the partition intervals searched during this step. For each token that either begins before or can be constrained to begin before the fetch interval, determine whether or not that token persists throughout the fetch interval. Relative offsets and the heuristic search routines are used to determine the relative ordering of the end-point of the found token and the end-point of the fetch interval.

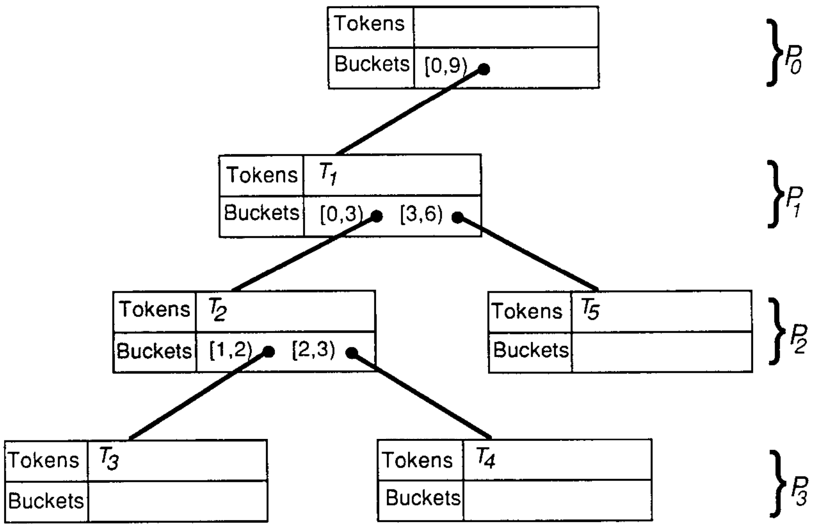


FIG. 5. Portion of a temporal discrimination tree.

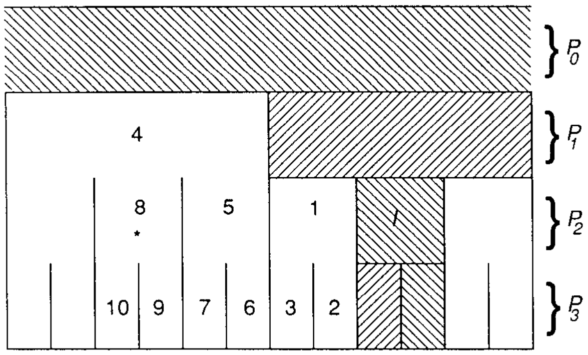


FIG. 6. Search strategy using a hierarchical partitioning scheme.

(3) We search through the remaining tokens as follows:

(a) Set the variable *early_termination* to false.
(b) Determine a partition interval, call it $\mathcal{N}$, preceding I such that there is no other unexamined partition interval preceding I, which is either later than $\mathcal{N}$ or in a less restrictive partition.
(c) For each token in $\mathcal{N}$, use the relative offset from the global frame of reference to determine if the token persists long enough.
(d) Mark $\mathcal{N}$ as examined.
(e) If a token is found that fails to persist throughout the fetch interval because it has been clipped by a contradictory token, then set *early_termination* to true.
(f) If *early_termination* is true and $\mathcal{N}$ is an element of the most restrictive partition, then stop, otherwise return to Step 3(b).

The numbers 1 through 10 in Figure 6 indicate the order in which partition intervals are examined during Step 3 of the algorithm. The starred partition interval is meant to indicate where the variable *early_termination* was set to true.

If P contains variables, then the termination criteria is more complicated. If we assume that all tokens of type P are mixed together in the same buckets of the temporal discrimination tree, then, having found a token of type P' unifying with P and clipped by a contradictory token of type $\neg P'$, you can set *early_termination* to true just in case $\neg P'$ clips all tokens of type P.

The above indexing scheme relies heavily upon their being a single global frame of reference and a rather simple and restrictive hierarchy of partitions. The fact that the partitions are strictly nested (i.e., for each $i < n$, if $I \in \mathscr{P}$, then there is a set of intervals in $\mathscr{P}_{i+1}$ that partitions I) can result in certain inefficiencies at the boundaries of partition intervals. Points that are unordered with respect to the boundary of a coarse partition interval may be assigned a fairly unrestrictive partition interval even though their relative offset is known with considerable precision. For instance, an event constrained to begin at midnight January 1 give or take a minute will end up in the bucket corresponding to the decade partition, assuming a partition according to decades, years, months, weeks, days, and hours. Such events cause the system to do a bit more work, but since they are relatively rare, the overall effect is negligible. Tokens whose begin points are known with some precision (i.e., the difference between the lower and upper bound of the relative offset is small), but which, nevertheless, are unordered with respect to major time breaks might be handled by considering pairs of partition intervals adjacent to the time break and indexing the token in more than one interval, but no attempt has been made to implement such a strategy in the current system.

5.2 Resolving Apparent Contradictions in Large Time Maps. Dean and McDermott [17] describe a technique for detecting and resolving apparent contradictions that involves setting up data dependencies (i.e., special data structures that help in incrementally updating databases [20]) for each pair of contradictory time tokens. This technique is designed to detect and resolve all and only those apparent contradictions produced by the current set of constraints (i.e., the TMM behaves correctly under the addition and deletion of constraints). The problem is that there are generally a great many pairs of contradictory tokens few of which will ever be implicated in an apparent contradiction. Since the cost of setting up the necessary data dependencies is significant, we would like to avoid doing so for as many pairs of tokens as is possible while still guaranteeing that the system detects and resolves all contradictions that actually occur. Not only would we like to avoid setting up the dependencies, if at all possible we would like to avoid even considering pairs of tokens that could not possibly cause apparent contradictions. The indexing scheme described in the previous section provides the means to efficiently handle the detection and resolution of apparent contradictions.

Intuitively, whenever a token T is placed in a new bucket, data dependencies are set up to keep track of contradictory tokens that are unordered with respect to T, and the system looks backward in time for tokens that should be clipped by T and forward in time for tokens that should clip T. More precisely, whenever the relative offset for a token T of type P changes so that T is constrained to begin during a partition interval I different than it was constrained to begin during previously, the system performs the following update procedure:

(1) Determine the set $\mathscr{T}_{\neg P}$ of all tokens of type contradicting P whose begin points are constrained to a partition interval either containing or contained in I.

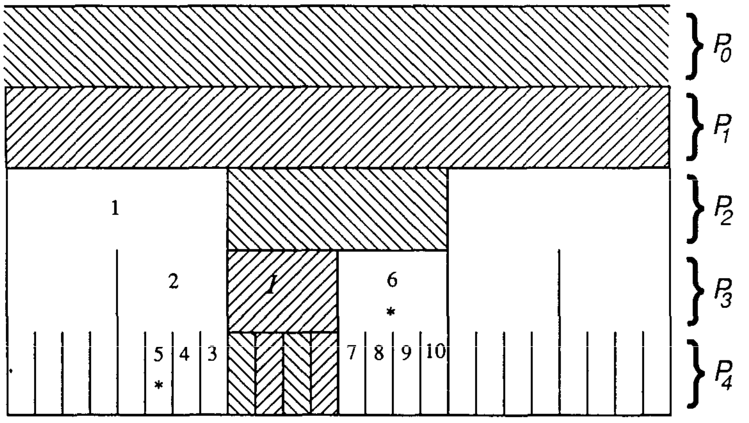


FIG. 7. Apparent contradiction-handling strategy.

Figure 7 depicts a partitioning scheme indicating I and showing the partition intervals either containing or contained in I as shaded.

(2) For each token in $\mathcal{T}_{\neg P}$, set up the data dependency structures required to detect and resolve apparent contradictions.

(3) Using the search strategy described in the token fetching scheme, work backward from I searching for tokens of type $\neg P$, clipping where required. As soon as you find a token T_{clipped} of type $\neg P$ that has already been clipped by a token of type P occurring prior to T, you can stop after searching through all partition intervals contained in the partition interval containing T_{clipped}.

(4) Similarly, work forward in time searching for tokens of type $\neg P$ that should clip T. As soon as you find a token T_{clips} of type $\neg P$ you can stop after searching through all partition intervals contained in the partition interval containing T_{clips}.

Figure 7 shows the order in which partition intervals are visited. Starred partition intervals indicate where the search encountered tokens corresponding to T_{clipped} and T_{clips}.

There are some additional complications that we have to deal with. If I corresponds to the whole time line (i.e., the single member of $\mathcal{P}_0$), then $\mathcal{T}_{\neg P}$ corresponds to *all* tokens of type $\neg P$. Since all tokens are initially unconstrained, unless steps are taken to avoid it, the system would have to set up data dependency structures for all pairs of contradictory tokens. To deal with this, the current TMM simply squelches indexing with respect to a given token until its relative offset meets some threshold level of precision.

A second complication occurs when a token T that has clipped or has been clipped is reindexed to a less restrictive partition interval. In order for the database to reestablish itself correctly, some clipping constraints may have to be deleted and other ones added; T may no longer clip or be clipped by the tokens it was previously, and those tokens clipped by T may now require clipping by tokens occurring after T's previous relative offset. With a bit of additional bookkeeping, however, it is fairly straightforward to handle reindexing.

5.3 EXPLOITING THE STRUCTURE OF TIME MAPS TO SPEED SEARCH. Searches to determine the best-known estimate on the distance separating two points are performed frequently in the time map, and it is worth the effort to (a) avoid them whenever possible and (b) optimize their operation if they are absolutely necessary. One way in which the TMM attempts to avoid unnecessary search is by ordering conjuncts in temporal queries to take advantage of special purpose routines for combining, suspending, and extending searches [17]. A second way of avoiding search involves the use of the temporal discrimination tree during token retrieval to ignore tokens that could not possibly satisfy the requirements of the fetch interval. There is still a significant amount of search for which the cached offsets from the global frame of reference will not provide adequate estimates of the distance between pairs of points. Just because two points are unordered with respect to the global frame of reference, does not mean that there is no path through the constraint network that would not serve to order them. To speed up point-to-point searches, the TMM caches derived estimates of the distance between selected pairs of points. These pairs of points are selected on the basis of some strategy for organizing the database. The performance of the search routines depends upon the same sort of factors that influence the performance of the constraint-propagation routines (e.g., the number of points and constraints in the time map and degree of redundancy of the information encoded in the constraints).

Each constraint introduced into the database, `(elt (distance` pt_1 pt_2`)` *low high*`)`, corresponds to a pair of directed edges in the time map $pt_1 \rightarrow pt_2$, labeled $\langle low, high \rangle$, and $pt_2 \rightarrow pt_1$, labeled $\langle -high, -low \rangle$. Each link is weighted with an integer indicating how useful that link is judged to be in estimating point-to-point distances. In searching for the best estimate of the distance separating two points, paths are extended from both points in best-first order, with best being determined by the sum of the weights on the links traversed in a given path. When a path starting from one point collides with a path starting from the other, the system checks the bounds of the composite path and updates its estimate if necessary. Usually the search routine is only allowed to explore paths of less than a given length, and is given a fixed amount of CPU time to carry out its search. If all links in the time map were given exactly the same weight, the search would be conducted in breadth-first manner. In applications involving sets of intricately interrelated events that cannot easily be organized (e.g., into distinct episodes), breadth-first search is probably the best strategy. In many applications, however, we can direct search with far greater precision.

It is possible to speed up the computation of point-to-point distance estimates by keeping track of (caching) the best-known estimates of distances separating selected pairs of points in the time map. If you cache distance estimates for all pairs of points, then determination of the best estimate for a given pair is trivial. The entire computational burden rests upon the routines for updating the table used to store point-to-point distance estimates. An earlier version of the TMM actually did compute the best estimates for all pairs of points in a restricted portion of the time map called the *kernel*. Updating this kernel requires $O(n^3 \log n)$ integer additions and array references where n is the number of tokens in the kernel. In typical applications, however, the kernel grew to several hundred tokens, and updating (resulting from adding one or more constraints) required several minutes even with all the important procedures optimized and coded in a low-level language. It became apparent that much of the work expended in updating all of these point-to-point distances was wasted. The version of the TMM used in this paper performs selective caching; only certain pairs of points are chosen for maintaining an accurate estimate of their separation. This means that the burden of computing point-to-point distance estimates rests partially with the routines for updating the selected point-to-point distances, and partially with routines for determining a given estimate on demand.

The point-to-point distance estimates chosen to be cached are selected on the basis of a general strategy for exploiting the hierarchical structure of tokens in the time map. The heuristic search routines employed in determining distance estimates take advantage of the cached distance estimates in order to quickly converge upon a "good" estimate. The expectation is that for pairs of points for which one might reasonably need to know an accurate distance estimate, the system would perform as well as exhaustive search, and for pairs of points that are not normally related to one another, performance would degrade reasonably.[1] It is possible to "tune" the system (adjust the way constraints are weighted and the search routines compute the best path to follow) for a given application in order to get significantly faster response time with no decrement in performance compared with a system without caching. In this subsection, we explore the issues involved in efficiently computing point-to-point distances.

In the TMM, without caching and making no assumptions about the constraints in the time map, determining the best bounds on the distance separating two points will require on the order of n^3 arithmetic operations where n is the number of points in the time map. If we assume that the length of a path used in establishing the best bounds never exceeds some constant (*max_path_length*), then we can limit the search by simply limiting the length of paths explored. The assumption is that temporal connectivity is largely determined locally, and rarely does it require consideration of all the points in the time map. Although this assumption is reasonable in most applications, we can still do a lot better. Tokens in a time map are constrained relative to one another in fairly regular ways: tasks are related to their subtasks and supertasks, persistences are constrained by some offset from the event that caused them, and prerequisite tasks are constrained to precede the tasks they serve. If we could rely upon certain point-to-point distances being known with precision (i.e., the values cached in some way), then the search paths for most other point-to-point distances would be comprised primarily of these cached values. Good examples of candidate pairs of points for caching point-to-point distances are the beginning and end of a token and, for a token corresponding to a task, the beginning of the token and the beginning of the token corresponding to its immediate supertask in the task/subtask hierarchy. We present a simple example illustrating some of the issues involved in caching and then proceed to demonstrate how selective caching pays off in certain circumstances.

Consider the diagram in Figure 8. The vertical lines labeled pt_1 through pt_6 are points. The bracketed numbers indicate low and high bounds on the estimated distance between the two points that they separate. T_1, T_2, and T_3 are tokens corresponding to the intervals pt_1 to pt_6, pt_2 to pt_3, and pt_4 to pt_5, respectively.

Given the information supplied in the diagram, one should be able to determine that pt_3 is coincident with pt_4. Determining that the two points are coincident using blind search will require as many as 6^3 operations depending upon the number of additional constraints in the network. If the system selectively caches certain point-to-point distance estimates, then it can make such determinations with significantly less effort. Suppose that the diagram in Figure 8 represents a small part of a task network in a time map. Suppose that the three tokens T_1, T_2, and T_3 correspond to tasks in this network, and that T_2 and T_3 are subtasks of T_1. Now, suppose

[1] I am relying upon the reader's intuitions here, but a simple criterion of reasonableness might be that the scale of the separation between a pair of points determines the accuracy of an estimate of their separation. For events widely spaced in the time map, no precise estimation of their separation is necessary even though given enough time an exhaustive search could supply such a precise estimation. For example, if I worked hard enough, I could determine to within a day how long it's been since my last visit to the dentist. For most purposes, however, "approximately three months" will work just fine.

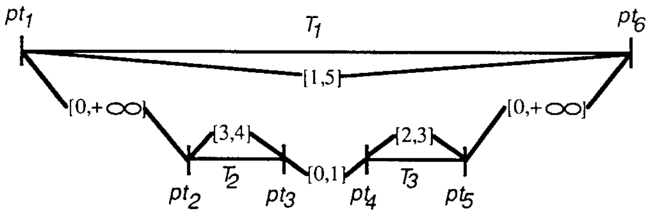


FIG. 8. Tokens in a simple task/subtask hierarchy.

further that the system keeps track of the best estimates on point-to-point distances separating the following pairs of points:

—the beginning and end of each time token
—the beginning of a task and the beginning of its supertask

Figure 9 shows the network of Figure 8 indicating the cached point-to-point distance estimates. Notice that in this network, finding the best estimate between any two points will require at most a path of length 4. The search strategy can be stated as follows. If you are trying to extend a path from a point, choose an edge corresponding to a cached value that takes you up (toward supertasks) in the task/subtask hierarchy. If the point has no such edge, then take whatever edge corresponding to a cached value you can get. If in the course of extending a path you find a second path struggling up the task hierarchy, see if you can combine the two paths to get the required distance estimate.

This strategy is not guaranteed to provide best estimates. If you need convincing, consider the network of Figure 8 with the constraint linking pt_1 and pt_6 labeled $\langle 1, 6 \rangle$ instead of $\langle 1, 5 \rangle$. In this network, augmented by using the caching scheme outlined above, our simple search strategy would return the value $\langle -2, 1 \rangle$ in response to a request for the best estimate of the distance between pt_3 and pt_4. The correct answer should be $\langle 0, 1 \rangle$. We can correct for this in most cases by combining a search strategy similar to the one outlined above with a bounded breadth-first search.

There are three main issues that have to be addressed with respect to our caching scheme:

(1) *Improving Cached Values.* Determining when a new constraint can assist in providing a better estimate on the distance separating a pair of points selected for caching.
(2) *Removing Invalid Estimates.* Determining when an old cached value is no longer valid due to the removal of constraints employed in its original derivation.
(3) *Expediting Heuristic Search.* Using the cached values in a search strategy for speeding up the computation of point-to-point distance estimates.

The first two of these issues are handled by the same techniques used for keeping track of relative offsets from the global frame of reference. The details are not all that interesting. During constraint propagation, if a path is found between two points that provides a better estimate of the distance separating those two points than the current best estimate, then the system updates a special constraint called a *caching constraint* to reflect this new estimate. The bounds of a caching constraint are expected to reflect the current best estimate of the distance separating the two points that the caching constraint links. By giving the caching constraint an

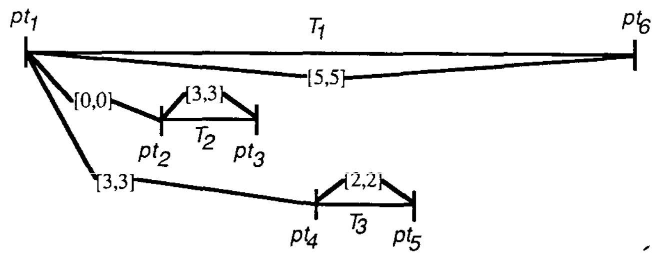


FIG. 9. A simple task network with cached values.

appropriate justification in the data dependency network, the system ensures that it will notice when an old cached value is no longer valid. Maintaining the invariant that all caching constraints reflect the best bounds is handled using standard data dependency programming techniques and a special priority queue mechanism (see [17] for details).

Search strategies for taking advantage of cached values are implemented by simply weighting the edges in the time map associated with caching constraints. If you want search to be biased in favor of moving up the task/subtask hierarchy, then you associate little weight (or cost) with the edge connecting the beginning of a subtask to the beginning of its immediate supertask, but some more significant cost with moving the other way. Bidirectional caching constraints (such as those used for caching the duration of time tokens) have the same cost no matter which direction they are traversed.

The cached values make many searches extremely fast. A time map corresponding to a *pure hierarchy* is one in which each task is constrained only with respect to its immediately superior supertask in the task/subtask hierarchy or with respect to one of its sibling tasks (both tasks share the same immediate supertask). Top-level tasks are constrained only with respect to one another. If time maps were pure hierarchies and all constraints were precise (i.e., $\forall c$ low(c) = high(c)), then all point-to-point distance estimates could be performed in time proportional to the depth of the hierarchy.

In experiments involving half a dozen top-level tasks expanded into over a hundred subtasks through several levels of refinement, the results are fairly clear. The additional cost involved in handling caching during constraint propagation increased the time to actually construct such time maps by approximately ten percent over the time required without caching. The time required to determine point-to-point distances accurately in the time map with caching decreased significantly over the scheme using breadth-first search with a depth cut-off carefully selected to take into account the sort of connectivity expected in the time map. The scheme with caching broke even with the breadth-first scheme after determining approximately 20 point-to-point distance estimates. In robot problem-solving tasks involving a time map containing several hundred tokens, the system typically computes hundreds of point-to-point distance estimates. In such situations, it is expected caching will result in significantly faster processing times, though no detailed comparisons have been made as yet.

One obstacle to getting the caching scheme described above to perform well is the fact that, in most applications, time maps are simply not pure hierarchies and most constraints are not precise. For example, networks of tokens corresponding to tasks typically become extremely tangled in the process of planning. This is due to the interleaving of tasks and the action of persistence clipping to resolve apparent contradictions. The average time to determine best possible distance estimates using the caching scheme is still much better than undirected (breadth first) search. We are currently experimenting with various techniques for weighting edges in the time map that depend upon specific properties of particular domains and problem-solving applications. Preliminary results appear promising, but we still need a lot more experience before we are able to tell to what extent savings made in speeding up searches are offset by the increased effort expended during the propagation of constraints. The evidence in favor of the caching techniques described in this section is not nearly as conclusive as the evidence in support of the caching and discrimination techniques described in Section 5.1.

6. *Performance*

The primary claim of this paper is that, in many common situations, temporal databases can be organized so that the cost of token retrieval is comparable with assertion retrieval in static databases. Restricting attention to the problem of retrieving a single token of type *P* (no variables) spanning the interval *I*:

(1) In situations in which the set of tokens can be partitioned into temporally distinct periods (e.g., months, years, factory work shifts), the cost of token retrieval is proportional to the sum of:

  (a) the number of periods separating the period containing the beginning of *I* and the first period, *p*, preceding the beginning of *I* containing a token of type *P*, and
  (b) the cost of determining if any token of type *P* in *p* spans *I*.

(2) In situations in which the database can be organized according to an event/subevent hierarchy and the constraints between events and subevents are exact (i.e., the upper bound is identical with the lower bound), then the cost of determining if any token of type *P* in a given period spans *I* is proportional to the product of the number of tokens of type *P* within that period and the depth of the event/subevent hierarchy.

As with any scheme for speeding up retrieval, there is a cost associated with organizing the data to support these fast retrieval routines. Fortunately, much of the work required for organizing time tokens is already being handled by other routines in the time map, specifically, the routines used for supporting temporal reason maintenance.

It is difficult to compare the system described in these pages with other systems for the simple reason that most existing systems are not as powerful as the TMM and are not designed to handle databases of the size being considered here. Earlier versions of the TMM employed techniques for searching the time map that made them at least the equal in token retrieval of systems such as those discussed in [3] and [45]. If an earlier version of the TMM was required to process a query that involved fetching tokens of a type of which there were hundreds, then it would potentially have to examine all of them with a significant reduction in performance. Vere [45] and Bell and Tate [3] would be forced into expending equivalent energies in similar circumstances. In the TMM with temporal indexing, such a query would require examining at most a few tokens. Caching offsets from a global reference makes determining point-to-point distances at query time trivial in many cases, and makes resolving apparent contradictions efficient where otherwise it would be prohibitively expensive.

Since there is a trade-off being made, we have to ask the question, "How much does it cost to perform the extra caching and indexing operations necessary to support this 'improved' performance?" The major additional cost is incurred at assertion and retraction time while propagating constraints and updating selected point-to-point distances. To handle temporal discrimination, there is at least one point-to-point distance to be updated for each point corresponding to the begin or end of a token. Of course, not every point-to-point distance is updated or even examined in the course of propagating a given constraint. Quite the contrary, in propagating a single constraint, generally only a very small portion of the time map is investigated. Given reasonable assumptions concerning the density of events over intervals of fixed duration, it is fairly easy to establish bounds on the length of paths explored during the propagation of constraints. For all intents and purposes, constraint propagation takes constant time, where the constant is measured in terms of a few CPU seconds. In the factory domain of the FORBIN planner, setting *max_path_length* to 20 is more than sufficient for handling most planning situations. In some cases, constraint propagation in the current implementation is faster than in previous implementations without temporal discrimination. This speed-up is due to the fact that, in the older version of the TMM, constraints made with respect to the global frame of reference tended to connect pairs of points distant in time with short paths, which increased (by quite a bit) the number of points reachable from a given point by paths of less than *max_path_length.* In the current system, since all offsets from the global frame of reference are cached, it is not necessary to consider paths that include the global reference as other than a begin or end. It is not really surprising that the global timeline eases the cost of search incurred in propagating constraints; clocks and calendars provide us with the structure required to organize large amounts of information and the TMM is simply making use of that structure to guide search.

Because the TMM attempts to optimize for many different sorts of inference (e.g., token retrieval, inferring the consequences of causal rules, temporal reason maintenance), it is difficult to design a test that accurately reflects the TMM's performance. Nevertheless, a concrete example that reflects the expected applications of the TMM is in order. Suppose that we are interested in keeping track of the operational status (i.e., `in-service` or `out-of-service`) of a single machine, `lathe14`, given information about observed and anticipated malfunctions and service calls (i.e., events of type `(malfunction lathe14)` and `(routine-service lathe14)`). Malfunctions and service calls have consequences corresponding to tokens of contradictory types (i.e., respectively, `(operational-status lathe14 out-of-service)` and `(operational-status lathe14 in-service)`). Now, suppose that the information concerning the malfunctions and service calls comes in a piece at a time, and that we expect the database to be updated after each piece of information to accurately reflect those intervals during which `lathe14` is in or out of service. At one moment, we might be told that `lathe14` broke down yesterday at noon, and, later, we might be told that it was serviced at 3:00 in the afternoon. For the test, we assert tokens corresponding to malfunctions and service calls and constrain these tokens to occur within a fixed interval of time (of duration 10^4 minutes). The events have fixed duration with their consequences true immediately following, and the time of occurrence is specified as a random offset from the global frame of reference with a fixed amount of uncertainty (+ or − 10 minutes). Since the events are randomly placed, a given persistence may clip and be clipped by a number of tokens of contradictory type. In the ideal situation, each persistence clips and is clipped by at most one token of a contradictory type; such situations are trivial for the TMM to handle and do not reflect the sort of situations encountered in real applications. Given the way that data is generally made available, and the way that plans are changed in the course of routine rescheduling and reevaluation, this ideal situation hardly ever occurs. Table I shows the data gathered from several runs of the TMM using the test described above. We are interested in the time required to build a time map containing *n* tokens half of which are of type `(operational-status lathe14 in-service)` and half of which are of type `(operational-status lathe14 out-of-service)`. Table I shows the total elapsed time spent and also the portion of that time spent in paging, as this becomes a significant factor as the number of tokens increases. The tests were performed on a Texas Instruments Explorer II Lisp machine configured with 8 megabytes of memory and garbage collection turned off. Paging and garbage collection affect one another, and we chose to eliminate the latter in order to focus on the former. The column labeled "Assertion Time" refers to the average time required to add one more token (of either type) minus the time spent in paging, given that the time map already contains *n* tokens. Once the time map is up to about 100 tokens, the average number of tokens clipped by a given token is about 10. We discount paging in order to show that the time required to add a single token is close to a constant function of the number of tokens in the database.

There are a couple issues concerning efficiency and reliability that are handled in traditional database systems, but that have been ignored in the current implementation of the TMM. In most database systems, relations are stored together in tables on disk. The fact that they are on disk makes them relatively impervious to system failures. The fact that the relations are stored in tables that (once read from disk) reside in a compact area of memory makes certain queries and data modifications quite efficient. In the current TMM, the tokens of a particular type and the discrimination tree through which they are accessed are spread all over the LISP system heap. It is possible to arrange things so that processing a single query requires the system to write into memory nearly every page of the LISP system's virtual memory. Fixing the problem is not as simple as forcing the system to be a little more careful in allocating storage. Part of the problem is due to the fact that the underlying data dependency network, upon which the caching and discrimination routines are so dependent, is also spread all over memory. Paging costs become significant for an 8-megabyte Explorer II handling time maps containing over a thousand tokens. If the need arises, we expect some combination of heuristic methods and traditional database technology will likely provide us with the means to handle still larger time maps.

There is an implementation of the TMM written in Common Lisp available to anyone interested in temporal reasoning. We encourage researchers and engineers to experiment with our system. Temporal reasoning of the sort that the TMM supports is complicated. It is quite possible that the trade-offs that we made for our applications are unsuitable for other applications, but we need feedback in order to explore the issues more deeply.

7. *Conclusion*

In conclusion, the TMM provides a wide range of functionality (backward and forward temporal inference, dependency-directed default reasoning, temporal reason maintenance) in a simple-to-use system (predicate-calculus syntax and Prolog compatibility) in which routine temporal reasoning is optimized using caching and search techniques to speed inference. Straightforward partitioning schemes supplied

TABLE I. TMM STATISTICS

Tokens	Total Time	Paging Time	Assertion Time
20	4.0	0.3	0.35
40	8.3	0.3	0.35
60	12.6	0.4	0.32
80	20.6	0.6	0.39
100	28.4	0.6	0.50
120	36.5	0.8	0.41
140	47.7	0.9	0.45
160	65.3	7.3	0.70
180	111.9	37.0	0.70
200	239.2	147.6	0.45

by an application program are used to fragment a temporal database into non-overlapping periods. Under situations that arise frequently in everyday reasoning, token retrieval, the basic operation common to most forms of temporal inference, can be performed in time proportional to the sum of (a) the number of periods separating the period containing the beginning of the fetch interval and the first previous period that contains a token of the desired type, and (b) the number of tokens of that type beginning in that previous period. The result is that the performance of the temporal inference engine corresponds roughly to our expectations given the distribution of tokens of the underlying fact types being manipulated.

The organizational schemes described in this paper are integral with the basic functionality of the TMM; they instigate reorganization correctly in response to the addition of new information or the deletion of old, and they serve to expedite the basic operations used in temporal reason maintenance. The techniques involve methods for partitioning the set of time tokens temporally and for caching estimates of the distance separating selected points and noticing when certain distance estimates become licensed by the current set of constraints or cease to be so. If we accept that the basic operation of temporal reason maintenance is essential, then the additional overhead in time and space necessary to implement these organizational techniques is just a constant factor times the number of tokens stored in the database.

Appendix A. Constraint Propagation

Relative offsets make the current constraint propagation algorithm slightly more complicated than the one described in [17]. This short appendix is provided to show how the algorithm presented in [17] can be extended to handle relative offsets. Knowledge of the notation and techniques described in [17] is assumed. The actual mechanics of the search are as before, the only thing that has changed is that all points now have a caching constraint used to update the point's relative offset from the global frame of reference. The following describes how the system updates TCONDITs[2] relating various pairs of points in the time map. Consider the simple constraint network shown in Figure 10. If you find a path $p_{3\rightarrow 4}$ from pt_3 to

[2] TCONDITs are annotations attached to points that tell the constraint propagation machinery to check if newly found paths between pairs of points can be used to derive facts of interest to the system. For instance, if the constraint propagation machinery found a path p from pt_1 to pt_2 with $\text{bounds}(p) = \langle 2, 3\rangle$ and a TCONDIT $tc_{1\rightarrow 2}$ concerned with the fact of whether or not pt_1 precedes pt_2, then it would note that the fact was justified by p and call a function associated with $tc_{1\rightarrow 2}$ to perform whatever response was required by the system. TCONDITs play an important role in temporal reason maintenance.

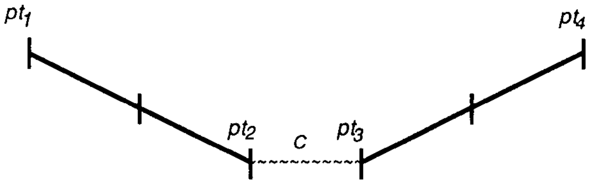


FIG. 10. Simple constraint network.

pt_4 and a TCONDIT $tc_{1 \to 4}$ relating pt_1 and pt_4, then

(i) Check for a path $p_{1 \to 2}$ from pt_1 to pt_2. If such a path exists, construct the composite path formed from $p_{1 \to 2}$, C, and $p_{3 \to 4}$ and use the computed bounds on the path to update $tc_{1 \to 4}$.
(ii) Check to see if you can get a better relative offset for pt_4. If so, update pt_4 and then use the new relative offset for pt_4 and the relative offset for pt_1 to update $tc_{1 \to 4}$.

Note that the best offset from pt_2, not employing C, is already known and hence if the relative offset of pt_4 is to be updated, it will be computed from $p_{3 \to 4}$, C, and the relative offset for pt_2. In the current TMM, constraints can refer to a single point and a date providing an indirect reference to the global frame of reference of the hierarchical partitioning scheme. Updating such constraints is handled by a simple variant of the above.

REFERENCES

1. ABADI, M., AND MANNA, Z. A timely resolution. In *Proceedings of the Conference on Logic in Computer Science* (Cambridge, Mass.). IEEE Computer Society Press, Washington, D.C., 1986, pp. 176–186.
2. ALLEN, J. F. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11 (Nov. 1983), 832–843.
3. BELL, C. E., AND TATE, A. Using temporal constraints to restrict search in a planner. Tech. Rep. AIAI-TR-5. Artificial Intelligence Applications Institute, Univ. Edinburgh, Edinburgh, Scotland, 1985.
4. BOLOUR, A., ANDERSON, T. L., DEKEYSER, L. J., AND WONG, H. K. T. The role of time in information processing: A survey. *SIGART Newsletter* (1982), 28–48.
5. BRUCE, B. A model for temporal reference and its application in a question answering program. *Artif. Int. 3* (1972), 1–25.
6. CHAPMAN, D. Planning for conjunctive goals. *Artif. Int. 32* (1987), 333–377.
7. CHARNIAK, E., AND MCDERMOTT, D. V. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1985.
8. CHARNIAK, E., RIESBECK, C. K., AND MCDERMOTT, D. V. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, N.J., 1980.
9. CLIFFORD, J., AND WARREN, D. S. Formal semantics for time in databases. *ACM Trans. Datab. Syst. 8*, 2 (June 1983), 214–254.
10. CLOCKSIN, W. F., AND MELLISH, C. S. *Programming in Prolog.* Springer-Verlag, New York, 1984.
11. DAVIS, E. Constraint propagation with interval labels. *Artif. Int. 32* (1987), 281–331.
12. DEAN, T. Planning and temporal reasoning under uncertainty. In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems.* IEEE, New York, 1984, pp. 131–138.
13. DEAN, T. Decision support for coordinated multi-agent planning. In *Proceedings of the 3rd International ACM Conference on Office Information Systems.* ACM, New York, 1986.
14. DEAN, T. Handling shared resources in a temporal data base management system. *Decision Supp. Syst. 2* (1986), 135–143.
15. DEAN, T. An approach to reasoning about the effects of actions for automated planning systems. *Ann. Oper. Res. 12* (1988), 147–167.

16. DEAN, T., AND BODDY, M. Reasoning about partially ordered events. *Artif. Int. 36* (1988), 375–399.
17. DEAN, T., AND MCDERMOTT, D. V. Temporal data base management. *Artif. Int. 32* (1987), 1–55.
18. DEAN, T. L., FIRBY, R. J., AND MILLER, D. P. The FORBIN paper. Tech. Rep. 576. Dept. Comput. Sci., Yale Univ., 1987.
19. DEAN, T. L., FIRBY, R. J., AND MILLER, D. P. Hierarchical planning involving deadlines, travel time and resources. *Comput. Int. 4* (1988).
20. DOYLE, J. A truth maintenance system. *Artif. Int. 12* (1979), 231–272.
21. ERNST, G., AND NEWELL, A. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, Orlando, Fla., 1969.
22. FIKES, R., AND NILSSON, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Int. 2* (1971), 189–208.
23. HALPERN, J., AND SHOHAM, Y. A propositional modal logic of time intervals. In *Proceedings of the Conference on Logic in Computer Science* (Cambridge, Mass.). IEEE Computer Society Press, Washington, D.C., 1986, pp. 279–292.
24. HANKS, S., AND MCDERMOTT, D. V. Nonmonotonic logic and temporal projection. *Artif. Int. 33* (1987), 379–412.
25. KAHN, K., AND GORRY, G. A. Mechanizing temporal knowledge, *Artif. Int. 9* (1977), 87–108.
26. KOOMEN, J. A. The TIMELOGIC temporal reasoning system. Tech. Rep. 231, Dept. Comput. Sci., Univ. Rochester, Rochester, N.Y., 1988.
27. KOWALSKI, R., AND SERGOT, M. J. A logic-based calculus of events. *New Gen. Comput. 4* (1986), 67–95.
28. LADKIN, P. The completeness of a natural system for reasoning with time intervals. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence* (Milan, Italy). Morgan-Kaufman, San Mateo, Calif., 1987, pp. 462–467.
29. LIFSCHITZ, V. On the semantics of STRIPS. In *Reasoning about Actions and Plans.* Georgeff, M. P. and Lansky, A. L., Eds. Morgan-Kaufman, San Mateo, Calif., 1987, pp. 1–10.
30. LUM, V., DADAM, P., ERBE, R., GUENAUER, J., PISTOR, P., WALCH, G., WERNER, H., AND WOODFILL, J. Designing DBMS support for the temporal dimension. In *Proceedings of SIGMOD* (Boston, Mass., June 18–24). ACM, New York, 1984, pp. 115–130.
31. MALIK, J., AND BINFORD, T. O. Reasoning in time and space. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany). Morgan-Kaufman, San Mateo, Calif., 1983, pp. 343–345.
32. MCCARTHY, J. Circumscription—A form of nonmonotonic reasoning. *Artif. Int. 13* (1980), 295–323.
33. MCCARTHY, J., AND HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence. *Mach. Int. 4* (1969), 463–502.
34. MCDERMOTT, D. V. Flexibility and efficiency in a computer program for designing circuits. Tech. Rep. 402, AI Laboratory. MIT, Cambridge, Mass., 1977.
35. MCDERMOTT, D. V. A temporal logic for reasoning about processes and plans, *Cognit. Sci. 6* (1982), 101–155.
36. MCDERMOTT, D. V., AND DAVIS, E. Planning routes through uncertain territory. *Artif. Int. 22* (1982), 107–156.
37. PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* IEEE, New York, 1977, pp. 46–57.
38. REITER, R. A logic for default reasoning. *Artif. Int. 13* (1980), 81–132.
39. SACERDOTI, E. *A Structure for Plans and Behavior.* American Elsevier, New York, 1977.
40. SHOHAM, Y. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence.* MIT Press, Cambridge, Mass., 1988.
41. SMITH, S. F. Exploiting temporal knowledge to organize constraints. Tech. Rep. CMU-RI-TR-83-12. Intelligent Systems Laboratory. Carnegie-Mellon Univ., Pittsburgh, Pa., 1983.
42. SNODGRASS, R. A temporal query language. Tech. Rep. TR-85-013. Dept. of Comput. Sci., Univ. of North Carolina, Chapel Hill, N.C., 1985.
43. TATE, A. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence* (Cambridge, Mass.). Morgan-Kaufman, San Mateo, Calif., 1977, pp. 888–893.
44. VERE, S. Planning in time: Windows and durations for activities and goals. *IEEE Trans. Pattern Anal. and Machine Intelligence, 5* (1983), 246–267.
45. VERE, S. Temporal scope of assertons and window cutoff. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (Los Angeles, Calif.). IJCAI, 1985, pp. 1055–1059.

46. VILAIN, M., AND KAUTZ, H. Constraint propagation algorithms for temporal reasoning. In *Proceedings AAAI-86* (Philadelphia, Pa.). AAAI, 1986, 377–382.
47. WILKINS, D. Domain independent planning: Representation and plan generation. *Artif. Int. 22* (1984), 269–302.
48. WILKINS, D. *Practical Planning.* Morgan-Kaufman, San Mateo, Calif., 1988.

RECEIVED AUGUST 1986; REVISED JUNE 1988 AND MARCH 1989; ACCEPTED MARCH 1989