Report Number: WUCS-90-09

1990-04-01

Determine Interior Vertices of Graph Intervals

Victor Jon Griswold

The problem of determining which events occur "between" two bounding events A and B in partially-ordered logical time is equivalent to being able to list, for a directed acyclic graph, the vertices on all paths with origin a and terminus b. We present four approaches to this problem, each progressively less memory-intensive. The two most promising of these approaches are examined in depth.

DETERMINING INTERIOR VERTICES OF GRAPH INTERVALS

Victor Jon Griswold

WUCS-90-09

April 1990

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

The problem of determining which events occur "between" two bounding events A and B in partially-ordered logical time is equivalent to being able to list, for a directed acyclic graph, the vertices on all paths with origin a and terminus b. We present four approaches to this problem, each progressively less memory-intensive. The two most promising of these approaches are examined in depth.

TABLE OF CONTENTS

LIST OF FIGURES

Determining Interior Vertices of Graph Intervals

Victor Jon Griswold

1. Introduction

1.1 Background

Our work towards the monitoring of distributed systems by means of observing "events" generated by those systems has made apparent the need to determine which events V_i occur "between" two bounding events **A** and **B** in quasi-ordered logical time.* By between events **A** and **B**, we mean **A**, **B**, and all events temporally after **A** and before **B**. The basis of temporal ordering is determined by the binary relation **precedes**; **C precedes D** means that, in a given system, **C** can be shown to have occurred before **D** (this often results from some form of causality).

Transitivity of **precedes** allows the ordering of many events which might have no obvious temporal relationship. Since the basic data used to record a system's event history consists of the events generated by the system and the "obvious" **precedes** orderings** between those events, we must use transitivity to infer the system's complete **precedes** information.

* We view the temporal progress of a distributed system in terms of quasi-ordered logical time[6], not real time. A quasi order is an "irreflexive partial" order, meaning that $A \prec A$ is false. Though quasi order is the proper description of distributed time, few people regularly use this term. Throughout the remainder of this paper, partial order will be used for quasi order except when ambiguity may otherwise result.

** An "obvious" precedes ordering means that, for two events **C** and **D**, it can be shown that **C** comes before **D** in all possible systems.

The nature of event-based logical time allows a directed acyclic graph to be constructed such that its vertices and edges are in one-to-one correspondence with events and the "basic" temporal **precedes** orderings mentioned above, respectively. Complete **precedes** information would correspond one-to-one with the edges in the transitive closure of such a graph. The above "list all events between **A** and **B**" problem is therefore equivalent to being able to list, for a directed acyclic graph, the vertices v_i on all paths with origin a and terminus b. Analogous to events, such vertices v_i are called <u>between</u> a and b (a and b inclusive). For a directed acyclic graph H, we define the vertices a and b and all other vertices between them as the **interval** from a to b, written $[a \Rightarrow b]$, in H. <u>Interval</u> is similarly defined for the logical-time event history corresponding to H.

In order to make this report less verbose, we shall, from this point on, make use of the one-to-one correspondence between a logical-time event history and a directed acyclic graph, H. **Precedes** relationships between events map directly to paths (or edges in the transitive closure) between vertices with no loss or gain of information; we shall therefore consider them identical except where ambiguity may result. Events map to vertices, but with a significant loss of information. So that we may refer to events and vertices as identical with less chance of ambiguity, we consider it possible for any algorithm working with H to reference the event $\mathbf{V}_i$ corresponding to any vertex v_i of H, and vice-versa. Thus, if we refer to "adding an event $\mathbf{V}_i$ to H," we mean adding a vertex v_i to H and establishing references between that vertex and $\mathbf{V}_i$.

Though the speed of responding to the above "list interval" query is not unimportant, the nature of our application makes the space complexity required for that response of paramount importance. A distributed system might generate thousands of events; any algorithm requiring just $O(\nu^2)$ space, ν the number of events (vertices in the corresponding history graph), is therefore considered of no practical use. Instead, we maintain that $O(\varepsilon)$, ε the number of temporal **precedes** edges in the basic history graph (not in its transitive closure), to be our target space complexity. Since our application must perform an operation on each event in the list returned by the query, there exists a limiting factor of $O(\nu_I)$ with respect to overall speed of the application, ν_I the number of events in the requested interval.

We currently consider only a centrally-controlled <u>monitor</u> of a distributed system, or <u>subject</u>. Distribution of the monitor itself, though of interest, is not a current area of research.

1.2 Problem Definition

Partially-ordered logical time is viewed as an incrementally-constructed history graph H, with events as vertices and temporal **precedes** relationships between events as paths. For instance, if event **C** occurred before event **D**, there would be a path in H with origin c and terminus d. It is known whether or not an event may potentially be a start or end bound in a "list interval" query, and it is known whether or not an event may potentially be at the head or tail of a **precedes** edge with an event later to be added to H. Events are associated with specific timelines, representing individual processors or shared data objects; events from the same timeline are added to H "in order" with respect to each other. Given this background, we now define the problem formally:

1.2.1 Terms

A history graph $H = \langle V, E \rangle$ is a directed acyclic graph. A vertex $v_i \in V$ represents a single event V_i, event defined below. A directed edge $e_k = (v_i, v_j) \in E$ represents a temporal relationship between two events V_i and V_j. An event (vertex) V_i (v_i) is said to precede an event (vertex) V_j (v_j), written $\mathrm{V}_i \prec \mathrm{V}_j$ ($v_i \prec v_j$), if there exists a directed path in H with origin v_i and terminus v_j. Initially, $H = H_0$, which contains a single distinguished vertex v_0 and no edges. Event V_0 represents the start of monitoring and is therefore the first vertex on each timeline, timeline defined below. Let $\nu \equiv |V|$, and $\varepsilon \equiv |E|$.

A timeline $\mathrm{T}_u \in T$ is a directed path originating at v_0 and containing only those vertices whose corresponding events are associated in such a way that they are known, before being added to H, to form a total order. A timeline generally represents the sequential event history of a single, deterministic component of a distributed system. T is the set containing all timelines, and $\tau \equiv |T|$.

The graph interval, or just interval, from vertex a to vertex b is the set of vertices on all directed paths in H with origin a and terminus b. This is written $[a \Rightarrow b]$; a is the start bound and b is the end bound of the interval.

An event $\mathrm{V}_i = \langle A_i, C_i \rangle$ consists of two sets. The elements of A_i are tuples called attributes with the structure $\langle type, value \rangle$, where *type* identifies the form of attribute and *value* is information associated with the attribute (an event may have more than one attribute of the same *type*). An event may possess characteristics, listed in C_i, which enable future operations to be performed with that event. These characteristics specify that the event is a candidate for incidence with a **precedes** edge tail or head or for being the start or end bound of a queried graph interval. Designated by **t**, **h**, **s**, or **e**, respectively, the meaning of these terms will be further clarified below. Every A_i contains at least one attribute of the form ⟨**timeline**, ⟨*timeline_id*, *version*⟩⟩, where **timeline** is the attribute *type*, *timeline_id* specifies a timeline on which the event is ordered, and *version* is a non-negative integer which specifies V_i's position on that timeline. V_0 has characteristic **t** and perhaps s, and is the only event with *version* equal to 0.

1.2.2 Operations

Initially:

$H = H_0$ (contains only v_0)

Operations: (subject to restrictions listed below)

add_event: Add a vertex v_i to H and establish references between v_i and its corresponding event V_i. Record the attributes and characteristics of V_i.

add_edge: Add an edge e_k to H.

rmchar: Remove a characteristic $\in$ {**t**, **h**, **s**, **e**} from an event V_i.

Queries: (subject to restrictions listed below)

list_interval: List the interval $[V_i \Rightarrow V_j]$ (i.e. list all events temporally between and including V_i and V_j).

Restrictions:

1) The above operations and query are issued in three phases, called "new_event," "update," and "query." During the *new_event* phase, a single event V_i is added to H. If edges will ever be added between this new event and any pre-existing events (any $V_j \ni j < i$), they are added during V_i's *update* phase. Furthermore, characteristics of any events may be removed to reflect new information during this *update* phase. Only **add_edge** and **rmchar** operations are legal during *update*.

The state of the history graph is stable with respect to its existing vertices after completion of V_i's *update* phase; the graph in this state is referred to as H_i. The *query* phase consists of any number of **list_interval** queries performed on H_i.

No **add_edge** or **rmchar** operations may take place during the *query* phase, which ends with the next **add_event**. The system starts with H_0 in the *query* phase, after the implied addition of V_0 with no edges.

2) An edge e_k added through **add_edge** must be from a vertex v_i to a vertex v_j such that V_i has characteristic **t** and V_j has characteristic **h**.

3) An event V_i recorded through **add_event** must have at least the **t** and **h** characteristics so that subsequent edges can associate it with a timeline.

4) The object of **rmchar** (an event V_i) must possess the characteristic to be removed (note that one can only "add" characteristics to an event at the time of its recording with **add_event**).

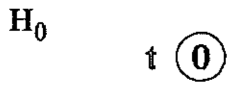
5) The interval listed through **list_interval** must be from an event V_i with characteristic s to an event V_j with characteristic **e**.

6) Given an event V_i recorded through **add_event**, there must be at least as many **precedes** edges with head v_i as V_i has **timeline** attributes. For each **timeline** attribute of V_i with value ⟨*timeline_id, version*⟩, *version* must be equal to the pre-existing maximum *version* of an event on timeline *timeline_id*, plus 1. Call this pre-existing event with maximum *version* V_j; the required **precedes** edge is (v_j, v_i). Note that this restriction enforces both that events on a particular timeline form a total order and that they are added to H "in order."

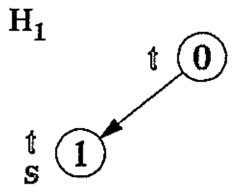
1.2.3 Diagrams

With Figure 1, we show the type of diagram used for a history graph and present an example of a sequence of valid operations on such a graph. In such a diagram, vertices/events are represented by circles with the event number inside the circle and the event characteristics to the side of the circle. **Precedes** edges are represented by arrows from tail to head. Vertices within the same timeline are arranged vertically with the least version number at the top (i.e. temporal order "flows down" the timeline). We say that an event **V** is ordered *on* a timeline **T** if it possesses a **timeline** attribute with value ⟨**T**, *version*⟩; an event **V** is ordered *with* a timeline **T** if there exists a path from any event **W** on **T** to **V**. Similarly, two events **V** and **W** are ordered with respect each other if one **precedes** the other.

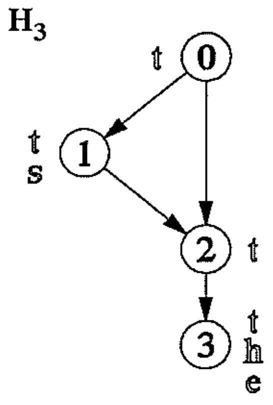
Figure 1 shows the construction of a history graph H from five events besides V_0, three timelines, and the potential for one interval query. In the following discussion, we will refer to



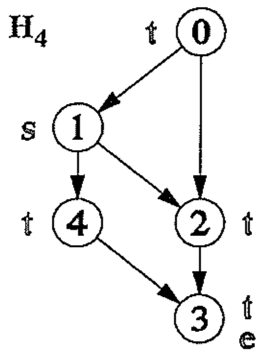
— Initial Conditions —



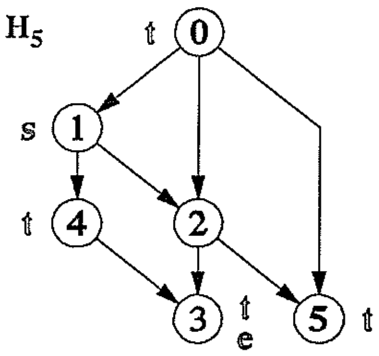
add $\mathbf{V}_1 = \langle\ \{\langle \mathbf{tmln}, \langle \mathrm{T}_1{,}1\rangle\rangle\},$
$\{\mathbf{t,h,s}\}\ \rangle$
add $e_0 = (v_0, v_1)$
rmchar V_1, $\{\mathbf{h}\}$



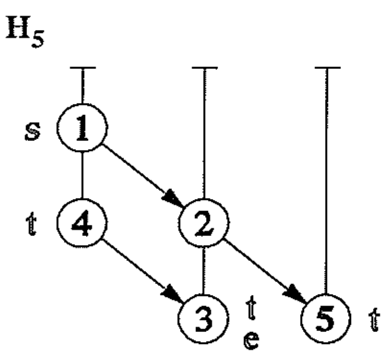
add $\mathbf{V}_2 = \langle\ \{\langle \mathbf{tmln}, \langle \mathrm{T}_2{,}1\rangle\rangle\},$
$\{\mathbf{t,h}\}\ \rangle$
add $e_1 = (v_0, v_2)$
add $e_2 = (v_1, v_2)$
rmchar V_2, $\{\mathbf{h}\}$
add $\mathbf{V}_3 = \langle\ \{\langle \mathbf{tmln}, \langle \mathrm{T}_2{,}2\rangle\rangle\},$
$\{\mathbf{t,h,e}\}\ \rangle$
add $e_3 = (v_2, v_3)$



add $\mathbf{V}_4 = \langle\ \{\langle \mathbf{tmln}, \langle \mathrm{T}_1{,}2\rangle\rangle\},$
$\{\mathbf{t,h}\}\ \rangle$
add $e_4 = (v_1, v_4)$
rmchar V_1, $\{\mathbf{t}\}$
rmchar V_4, $\{\mathbf{h}\}$
add $e_5 = (v_4, v_3)$
rmchar V_3, $\{\mathbf{h}\}$



add $\mathbf{V}_5 = \langle\ \{\langle \mathbf{tmln}, \langle \mathrm{T}_3{,}1\rangle\rangle\},$
$\{\mathbf{t,h}\}\ \rangle$
add $e_6 = (v_0, v_5)$
add $e_7 = (v_2, v_5)$
rmchar V_2, $\{\mathbf{t}\}$
rmchar V_5, $\{\mathbf{h}\}$



As to the left, but:
— V_0 implied
— time within a timeline implied

Figure 1. History Graph Structure and Operations

the operations and queries performed on H as being supplied by "the user," though in reality this "user" will be a program.

As shown in Figure 1, H_0 contains only v_0. V_0 has the **t** characteristic, so **precedes** edges may originate at it, but, in this example, V_0 has no s characteristic, so no interval query may designate V_0 as its start bound. Event V_1 is then added to H_0. V_1 belongs to timeline $\mathbf{T}_1$ and is version 1 of that timeline (V_0 is version 0 of $\mathbf{T}_1$ and all other timelines). V_1 has the **t**, **h**, and **s** characteristics, so **precedes** edges may originate and terminate at it and the user may make an interval query with V_1 as the start bound (but not an interval query with V_1 as the end bound). **Precedes** edge e_0 is then added to H from V_0 to V_1, as shown by the arrow. The user, in this example, realizes that e_0 is the only edge which may terminate at V_1, and therefore removes its **h** characteristic. If the user did not realize this fact and remove the **h** characteristic, proper query results would not be affected but certain data structure optimizations might not be possible. This completes V_1's *update* phase and thus the construction of H_1.

The constructions of H_2 and, afterwards, H_3 are similar to that of H_1, and involve the addition of two events and three edges. Of note is that V_3 has the **e** characteristic; after V_3 and all incident edges are added to H (i.e. after H_3 is completed in V_3's *update* phase), the user may issue an interval query for $[V_1 \Rightarrow V_3]$ (and would receive $\{V_1, V_2, V_3\}$ in response).

H_4 consists of H_3 with one more event and two more edges. Also, the user realizes that no further edges may originate at V_1 and removes its **t** characteristic, providing an avenue for further data structure optimizations. H_5 adds the final event and edges to our example. In H_5, neither V_1 nor V_2 may be incident to any new edges to be added to H. With this graph, the response for an interval query of $[V_1 \Rightarrow V_3]$ would be $\{V_1, V_2, V_3, V_4\}$. The response would not include V_5 because, though V_5 is after V_1, its temporal relationship with V_3 is indeterminate.

The last graph in Figure 1 shows a somewhat abbreviated representation of H_5; this is the style of representation we shall use throughout the remainder of this report. In this style of representation, V_0 and the edges incident to it are implied since they are present in all history graphs. Additionally, the **precedes** edges which show the progression of order along a timeline are represented simply by segments instead of by arrows, since arrows within a timeline always point down in our graph representations.

2. Transitive Closure Method

2.1 Approach

A rather robust means to respond to an interval query is to maintain complete transitive closure information about the history graph, making no assumptions about its structure other than it is directed and acyclic. When a query is issued for [A $\Rightarrow$ B], the answer is simply the intersection of all events after A and before B, A and B inclusive.

To our knowledge, the fastest published algorithm for incrementally maintaining the transitive closure of a directed acyclic graph was developed by Giuseppe F. Italiano.[4] This algorithm adds edges to a graph in O(v) amortized time per edge and reports the ordering between two events in O(1) (constant) time. Unfortunately, Italiano's algorithm requires pre-knowledge of the number of vertices in the graph, due to storage allocation considerations, and has a space complexity of $\Theta(v^2)$.

2.2 Algorithm

Though the algorithm's space complexity is prohibitive and **add_edge** time undesirable, it is still of some interest to examine how the algorithm might be employed. Of particular use for the **list_interval** query is the v by v lookup table maintained by the algorithm in order to directly check for the existence of a path from any vertex v_i to any other vertex v_j. We find the algorithm's ability to list a single path from v_i to v_j of little use for our purposes of listing <u>all</u> such paths.*

We take this opportunity to introduce the pseudocode representation employed for the expression of algorithms in this report. Our pseudocode employs a Pascal-like syntax, explained in detail in Appendix 7.1. The three operations and query we require are declared in Figure 2, along with the data types used in the declarations and data structures which might support the operations.

The operations and data structures provided by Italiano's algorithm are presented in Figure 3 and detailed in Appendix 7.2. As shown, one may add an edge, check if a path exists between two vertices, or find a path between two vertices. The data structures maintained include

* It is not feasible to modify Italiano's algorithm in order to report all paths between a pair of vertices. The very optimization which allowed him to achieve O(v) (instead of O(vlogv)) "add edge" time was the removal of all such "redundant" multiple-path information from the algorithm's data structures.

```
constants
    V_limit, e_limit : integer := some large positive number    //  greatest # of elements
    id_null : integer := -1;                                    //  "no such object"

types
    natural = range [0..] of integer;
    event_id, edge_id, timeline_id = range [id_null..] of integer;
    version_index = natural;

    attribute_type = (timeline, other);          // we are unconcerned about other attribute types

    ordering = record                            //  to order an event w.r.t. a specific timeline
        tid :  timeline_id;
        ver : version_index;
    end ordering;

    attribute = record
        case atype : attribute_type of
            timeline : (value_tl : ordering;);
            other :    (value : tuple;);         // arbitrary structure
        endcase;
    end attribute;

    characteristic = (t, h, s, e);               // edge tail or head, interval start or end

    event = record
        attrib : list of attribute;
        chr :   set of characteristic;
    end event;

    edge = record
        tail, head : event_id;
    end edge;

globals
    V : array [0..V_limit] of event;             //  any O(1) access time structure
    V_total : natural := 0;                      //  current number of events
    e :  array [0..e_limit] of edge;             //  any O(1) access time structure
    e_total :  natural := 0;                     //  current number of edges

procedure add_event (new_V : event; out vid : event_id);
procedure add_edge (t_vid, h_vid : event_id; out eid : edge_id);
procedure rmchar (rm_vid : event_id; chr : set of characteristic);
function list_interval (s_vid, e_vid : event_id) : list of event_id;
```

Figure 2. Declaration of Required Operations

an array with which to make rapid path existence checks and a set of trees to record the actual paths.

Unless reorganization of the path existence lookup table is permitted, the number of vertices is fixed for Italiano's algorithm* and our **add_event** procedure is a thus no-op with respect to the Italiano data structures, simply making a record of the event attributes. Additionally, since Italiano's algorithm makes no optimizations based on knowledge of future **add_edge** or **list_interval** operations, the **rmchar** operation is effectively a no-op with respect to the algorithm. The **add_edge** operation is not a no-op, though is trivial:

```
procedure add_edge (t_vid, h_vid : event_id; out eid : edge_id);
begin
    Ital_add_edge(t_vid, h_vid);
    eid := e_total;

    return;
end add_edge;
```

```
types
    Ital_node = record
        key :     event_id;
        parent : ^Ital_node;
        child :   ^Ital_node;
        sibling : ^Ital_node;
    end Ital_node;

globals
    //  index[i, j] ≠ null → a path exists from v_i to v_j
    //
    index : array [0..V_limit, 0..V_limit] of ^Ital_node := null;
    desc :  array [0..V_limit] of ^Ital_node;

procedure Ital_add_edge (tail, head : event_id);
function Ital_check_path (org, term : event_id) : Boolean;
function Ital_get_path (org, term : event_id) : list of event_id;
```

Figure 3. Operations Provided by Italiano's Algorithm

* It is possible to dynamically increase the expected number of vertices in the lookup table, but this operation requires a significant reorganization of that data structure (it does not increase the O(v) running time, just the constant factor). Such restructuring would place a bursty, unpredictable performance impact on the monitor which might well prove unacceptable.

We perform the **list_interval** query by reference to Italiano's **index** array, effectively finding the intersection of events after the interval's start bound with those before its end bound. This query implementation, though rather straightforward, is still **O**(v) time, similar to the **O**(v_I) application limit for the query but perhaps much larger.

```
function list_interval (s_vid, e_vid : event_id) : list of event_id;
    Vlist : list of event_id := [ ];
    vid :   event_id;

begin
    if index[s_vid, e_vid] then
        Vlist &= [s_vid, e_vid];
        for vid in 0..V_total - 1 do
            if index[s_vid, vid] and index[vid, e_vid] then
                Vlist &= [vid];
            endif;
        endfor;
    endif;

    return Vlist;
end list_interval;
```

3. Search Tree Method

3.1 Approach

Our next method of responding to the **list_interval** query relies on the history graph's timeline structure to achieve $O(\tau^2\log(\varepsilon_X)+\tau\log v)$ **add_edge** and $O(\tau(\log(\varepsilon_X)+v_I))$ **list_interval** time, and to require $O(\tau\varepsilon_X+v)$ space. We define $\varepsilon_X \equiv \varepsilon\text{-}v$, a measure of the edge cross-connectivity between timelines. Though this space bound may at first appear worse than that of Italiano's algorithm because ε, for a general graph, is $O(v^2)$, properties of our application make ε of $O(\tau v)$. For subjects with a large number of events relative to the number of timelines, the search tree method may thus require considerably less time and space than the transitive closure method using Italiano's algorithm.

In the search tree method, we maintain each timeline as a sorted set of events* and, for each pair of timelines, maintain a sorted set containing information about all paths from one timeline to the other. Specifically, for each edge e_k incident with a vertex v_j, the algorithm checks every timeline for its highest-version vertex v_i such that a path from v_i to v_j exists which uses e_k as the terminal edge. If a path does not exist from v_i to the previous vertex on v_j's timeline (i.e. an indirect path does not already exist from v_i through the previous vertex to v_j), a record of that path's existence is made. These pairwise-timeline sorted sets are the reason for the τ^2 factors in the above complexity measures; if, for a particular system, temporal reference between timelines has a strong locality (for instance, each processor represented by a timeline might only talk with its "neighbors"), the τ^2 factors will actually be τ or $\tau\log\tau$.

Figure 4 illustrates a history graph along with the cross-timeline path information maintained for that graph. In Figure 4a, we see a history graph with three timelines and fourteen events (not counting V_0); Figure 4b-d show the recorded path information for that graph, one sub-figure for the path information associated with each of the three timelines. In each of Figure 4b-d, the "other" timelines of the underlying graph are de-emphasized by showing them as dotted lines while the paths leading to the sub-figure's timeline are shown as bold lines. Given these cross-timeline path data structures, checking for the existence of a path from, for example, V_7 to V_{12} proceeds as follows:

* Specifically, we use threaded AVL search trees ordered by event version.

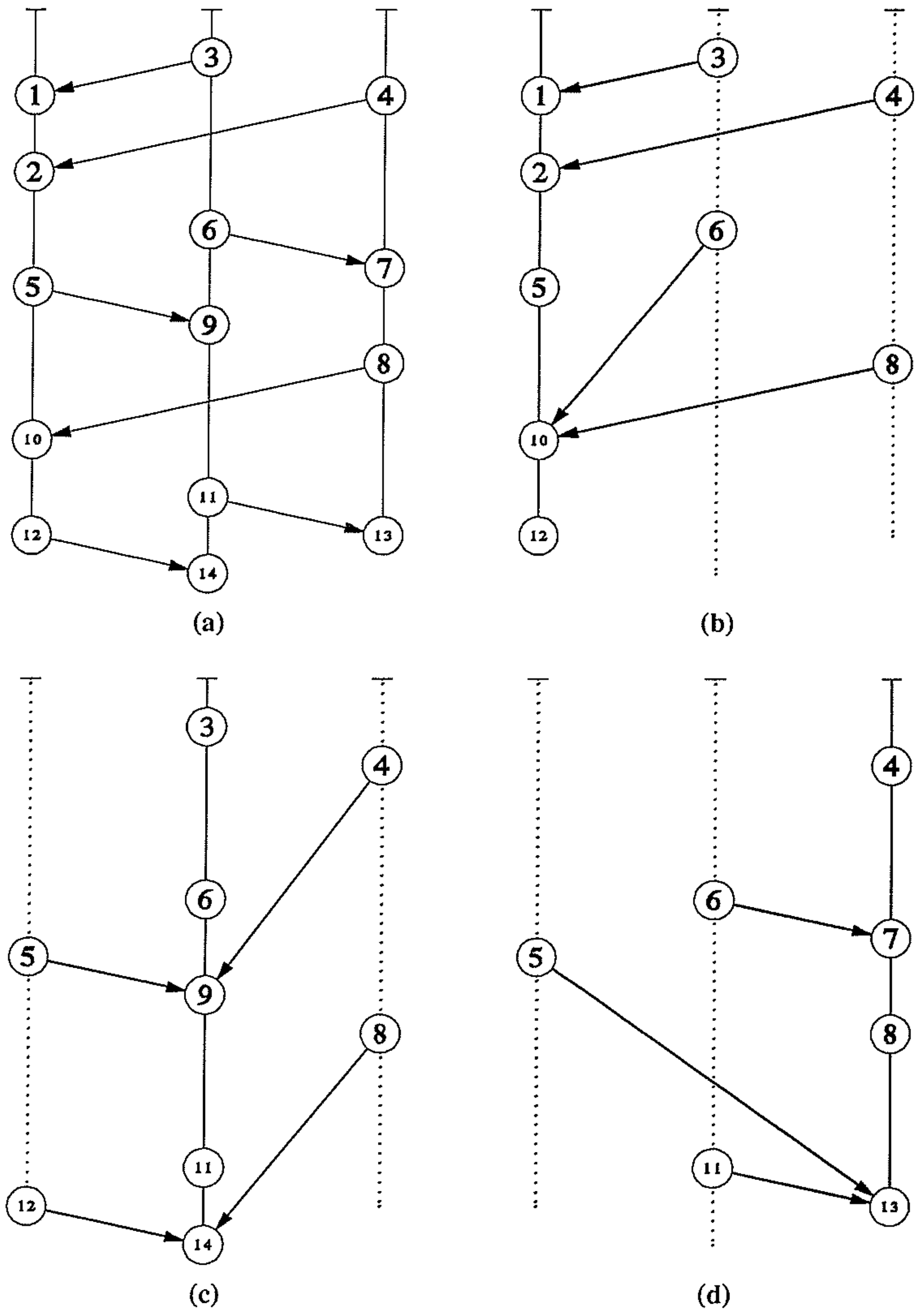


Figure 4. Cross-Timeline Path Information for Search Tree Method

1) Check for the most recent terminus (at or before V_{12}) of a path which originates from V_7's timeline and terminates at V_{12}'s timeline, and which contains no edges on V_{12}'s timeline. This terminus would be V_{10}.
2) Check to see if the origin of that path has a version greater than that of V_7. In this case, the path origin is V_8, which occurs later on the appropriate timeline than V_7, so a path does exist from V_7 to V_{12}. In the abstract sense, this path originates at V_7, proceeds to V_8 along some number of edges on their mutual timeline, proceeds to V_{10} along some number of edges across some number of intermediate timelines, and finally terminates at V_{12} along some number of edges on the timeline V_{10} and V_{12} have in common.

3.2 Algorithm

We assume the existence of the sorted set operations described in Appendix 7.1 and that their implementation requires O(log *number of items in tree*) time per operation.[10] In addition to the data structures of Figure 2, we use those presented in Figure 5. Remember that the cross-timeline data structures keep track not of individual edges, but of paths between timelines. Since the search tree method still makes no optimizations based on knowledge of future **add_edge** or **list_interval** operations, the **rmchar** operation is effectively a no-op. The pseudocode presented here is a high-level description of the algorithm; a more detailed description is found in Appendix 7.3.

Due to the following optimization in the **cross_tl_data** structure, the search tree method's **add_event** operation just records the event's attributes. In order to fulfill the **list_interval** query, we must report the identifiers of the events in the requested interval. The search tree method's path recording mechanism, however, generally tracks only the version of an event on a particular timeline (since that is how events are ordered on a timeline, not by their identifier). We either must maintain a separate data structure to record the event identifiers or must somehow maintain the identifiers along with the paths. This optimization makes use of the knowledge that, if a path is ever recorded between two events on the same timeline, the version of the origin is always that of the terminus, minus 1. The space ordinarily used to hold the origin's version is used, instead, to hold the event identifier of the terminus.

As per its definition for the *new_event* phase, no edges are added during **add_event**, even to other events in the same timelines as **new_V**. Pseudocode for **add_event** is:

```
types
    ordering_set = srt_set of ordering key tid;

    //      Versions of origin and terminus of a path from one timeline to another.  If
    //  both timelines are identical, the origin's version is replaced with the event
    //  identifier of the terminus since the origin's version would simply be terminus
    //  version - 1.
    //
    tl_path = record
        case (cross_timeline, in_timeline) of
            cross_timeline : (org : version_index;);
            in_timeline :      (vid : event_id;);
        endcase;
        term : version_index;
    end tl_path;

    cross_tl_data = record
        org_tid :      timeline_id;                    //  tl id of origins of recorded paths
        p_by_org :  srt_set of tl_path key org;
        p_by_term : srt_set of tl_path key term;
    end cross_tl_data;

    timeline = record
        tid :      timeline_id;
        xtdata : srt_set of cross_tl_data key org_tid;
    end timeline;

globals
    T : srt_set of timeline key tid;
```

Figure 5. Search Tree Method Data Structures

```
    procedure add_event (new_V : event, out vid : event_id);
    begin
        vid := a unique event identifier;
        V[vid] := new_V;

        return;
    end add_event;
```

The **add_edge** operation is rather complex because, after adding an edge from an event V_i to V_j, V_j and all events which follow it must now follow all events which V_i follows (i.e. for all events which **precede** V_i, the data structures must be updated so that they **precede** all events which now follow V_i). Further complications result from the possibility that V_j is ordered on multiple timelines.

An important subroutine of **add_edge** is **update_tl_xt**, which accepts an event V_j on one timeline T_u and a set of events (timeline versions, actually) which must come before V_j, then updates T_u's cross-timeline data structure so that these orderings are recorded. The creation of a new cross-timeline structure (if T_u had no existing orderings w.r.t a particular timeline) is also handled by **update_tl_xt**, as is the case when the new **precedes** information makes existing **precedes** information out-of-date (by showing that V_j comes after a later event on a particular timeline than higher-version events on T_u were known to come after; the **precedes** information for those higher-version events on T_u is no longer relevant after V_j's ordering is recorded). Below, we list **update_tl_xt**, followed by **add_edge**:

```
procedure update_tl_xt(tl : ^timeline; ver : version_index;
                            vid : event_id; prcd : ordering_set);
    xt :  ^cross_tl_data;
    ord : ordering;
begin
    for each ord of prcd do
        xt := tl's cross-timeline data for ord's timeline;

        if no existing data for that timeline then
            add a new cross-timeline structure to tl^.xtdata;
            add the initial $V_0$ to that structure;
        endif;

        if ord's timeline is not tl itself then
            if ord's information is not redundant then
                add ord --> ver path to xt;
                remove information made redundant by ord;
            endif;
        else
            add ord's information to xt^.p_by_term only;
        endif;
    endfor;

    return;
end update_tl_xt;
```

```
procedure add_edge (t_vid, h_vid : event_id; out eid : edge_id);
    tl : ^timeline;
    h_ver, term_ver : version_index;
    xt : ^cross_tl_data;
    prcd : ordering_set := new_srt_set;
begin
    eid := a unique event identifier;
    e[eid] := ⟨t_vid, h_vid⟩;

    //  Find all events which now precede the head (V[h_vid])
    //
    find any timeline on which the tail (V[t_vid]) is ordered;
    for each xt in that timeline's cross-timeline data do
        if xt is not for the tail's timeline itself then      //  handled below
            find the latest event on xt's timeline before the tail;
            if the event is not just $V_0$ then      //  everything comes after $V_0$; ignore it
                add that event to prcd;
            endif;
        endif;
    endfor;

    for each timeline on which the tail is ordered do
        add the tail's version on that timeline to prcd;
    endfor;

    //  Update the head to follow prcd
    //
    for each timeline on which the head is ordered do
        tl := that timeline;
        h_ver := the head's version on that timeline;
        update_tl_xt(tl, h_ver, h_vid, prcd);
    endfor;

    //  Update all events which the head precedes to follow prcd
    //
    find any timeline on which the head is ordered;
    for each tl of T except the above timeline do
        xt := tl's cross-timeline data for the head's timeline;
        if xt ≠ null andif there is an event on tl after the head then
            term_ver := the version of that event on tl;
            update_tl_xt(tl, term_ver, id_null, prcd);
        endif;
    endfor;

    return;
end add_edge;
```

Events in an interval [$V_s \Rightarrow V_e$] are found by determining all timelines with which V_e is ordered, then determining the first event on each of those timelines which occurs after V_s. For each such timeline, the interval includes all events after V_s and before V_e. Care must be taken to not multiply list events which are on more than one timeline. Pseudocode for **list_interval** is:

```
function list_interval (s_vid, e_vid : event_id) : list of event_id;
    Vlist : srt_set of event_id := new_srt_set; //    avoid duplicates
    mid_e_set : ordering_set := new_srt_set;
    ord : ordering;
    tl :   ^timeline;
    xt :   ^cross_tl_data;
begin
    //      Find the latest event before $V_e$ (V[e_vid]) on each timeline with which $V_e$ is
    //  ordered.
    //
    find any timeline on which $V_e$ is ordered;
    for each xt in that timeline's cross-timeline data do
        if xt is not for $V_e$'s timeline itself then
            find the latest event on xt's timeline before $V_e$;
        else                                    //   this will lead to putting $V_e$ in Vlist
            the event we use is $V_e$ itself;
        endif;
        if the event is not just $V_0$ then
            add that event to mid_e_set;
        endif;
    endfor;
    //      Add all events after $V_s$ and before $V_e$ to Vlist, doing one timeline at a time
    //  between the first event after $V_s$ and the latest event before $V_e$ (stored in mid_e_set).
    //
    find any timeline on which $V_s$ (V[s_vid]) is ordered;
    for each ord of mid_e_set do
        tl := ord's timeline;
        xt := tl's cross-timeline data for $V_s$'s timeline;
        if xt $\neq$ null then
            if $V_s$ is not on tl then
                find the first event on tl after $V_s$;
                if there is such an event andif it is not after ord's event then
                    add all events between this event and ord's event to Vlist (inclusive);
                endif;
            else                                //   $V_s$ is on this timeline
                if $V_s$ is not after ord's event then
                    add all events between $V_s$ and ord's event to Vlist (inclusive);
                endif;
            endif;
        endif;
    endfor;
    return srt_set_to_list(Vlist);
end list_interval;
```

3.3 Analysis

The $O(\tau^2\log(\varepsilon_X)+\tau\log v)$ time for **add_edge** is calculated by direct examination of the algorithm's pseudocode. Let us begin by looking at **update_tl_xt**. The top level of this subroutine is a loop for each timeline with which V[**vid**] should be ordered; there could be τ timelines. Within the loop, V[**vid**]'s timeline is searched for a cross-timeline structure corresponding to the loop variable, **ord**. This search is $O(\log\tau)$. If this structure is not present, it is created with an $O(\log\tau)$ insert and another $O(\log\tau)$ search. If the new ordering is relevant to V[**vid**], it is recorded with either two $O(\log(\varepsilon_X))$ or one $O(\log v)$ insertion(s) (depending upon whether or not the ordering is within V[**vid**]'s own timeline). Whenever the ordering is not within V[**vid**]'s own timeline, an out-of-order situation must be checked. The pseudocode above remedies this out-of-order situation with a slow $O(\varepsilon_X)$ delete loop for purposes of storage reclamation. This is desirable in many cases, but is not the fastest way to remove the out-of-order information; two tree splits and a tree join, $O(\log(\varepsilon_X))$, are all that is required to rectify the problem.

The above analysis yields an $O(\tau(\log\tau+\log(\varepsilon_X))+\log v)$ running time for **update_tl_xt** (only one of timelines in **pred** can be V[**vid**]'s own). Actually, though, we can compare τ and ε_X in order to achieve a less verbose measure. A timeline has cross-timeline structures for itself and for all other timelines with which it is ordered; it can be ordered with no more timelines than there are edges between timelines, ε_X. Therefore, for this calculation, $\tau \le \varepsilon_X + 1$ and thus $O(\log\tau) \le O(\log(\varepsilon_X))$. The time required by **update_tl_xt** is hence simplified to $O(\tau\log(\varepsilon_X)+\log v)$.

The pseudocode for **add_edge** consists of three primary operations: find all events before V_h, update V_h's cross-timeline structures, and update the cross-timeline structures of all events which follow V_h. Finding the events before V_t (and therefore before V_h) requires a $O(\log\tau)$ search to find a V_t's timeline T_u and, for each of T_u's τ potential cross-timeline structures, an $O(\log(\varepsilon_X))$ search and possible $O(\log\tau)$ insert. The ordering of V_t itself with respect to V_h is handled with an $O(\log\tau)$ insert for each timeline on which V_t is ordered (τ possible). Total time is $O(\tau\log(\varepsilon_X))$, using the same $O(\log\tau) \le O(\log(\varepsilon_X))$ argument as above.

Updating V_h's cross-timeline structures involves, for each of τ possible timelines T_w on which V_h is ordered, finding T_w with an $O(\log\tau)$ search and properly applying **update_tl_xt** to it. Given the above analysis for **update_tl_xt**, this operation is $O(\tau^2\log(\varepsilon_X)+\tau\log v)$.

To complete **add_edge**, we must update the cross-timeline structures of all events which follow V_h. For each timeline T_z in the graph, we must check to see if it is ordered with respect to a timeline T_w on which V_h is ordered ($O(\log\tau)$). If so, we find the first event on T_z after V_h ($O(\log(\varepsilon_X))$) and apply **update_tl_xt** when appropriate. Completion of **add_edge** thus requires $O(\tau^2\log(\varepsilon_X))$, similar to updating V_h's cross-timeline structures. With the analysis of the other two operations within **add_edge**, this implies that **add_edge** as a whole is of $O(\tau^2\log(\varepsilon_X)+\tau\log v)$.

As for **add_edge**, **list_interval**'s time complexity is calculated by examination of the pseudocode. The algorithm begins by finding V_s and V_e (requires one $O(\log\tau)$ search), then finding the latest event V_{mid_e} before V_e on each timeline with which V_e is ordered. There may be τ timelines, and the search requires an $O(\log(\varepsilon_X))$ lookup and an $O(\log\tau)$ insert. Time for this part of the algorithm is therefore $O(\tau(\log\tau+\log(\varepsilon_X))$, or $O(\tau\log(\varepsilon_X))$ by means of the $O(\log\tau) \leq O(\log(\varepsilon_X))$ argument presented above.

We build the list of events to be returned by **list_interval** one timeline at a time, for each timeline T_u ordered with respect to V_e. Given a T_u, we begin by locating it and its orderings with respect to V_s' timeline T_s ($O(\log\tau)$ searches). If such an ordering exists and $T_u \neq T_s$, we find the first event V_{mid_s} on T_u after V_s ($O(\log(\varepsilon_X))$ and find T_u's self-referential cross-timeline structure ($O(\log\tau)$). Finally, we sequentially scan that structure's event list between T_u's V_{mid_s} and V_{mid_e}, adding to the interval list as we proceed ($O(v_I)$). If $T_u = T_s$, we just immediately begin scanning between V_s ($\equiv V_{mid\_s}$, in this case) and V_{mid_e}. The list building operation thus requires $O(\tau(\log(\varepsilon_X)+v_I))$, making this also the time for **list_interval** as a whole.

We consider the above **list_interval** measure to be quite pessimistic; the τv_I term is an accurate measure only if the number of times an event is on more than one timeline is $O(\tau)$. In most "realistic" systems, an event on more than one timeline signifies a rendezvous between two processes ($O(1)$), not between some fraction of τ processes. For this common case, **list_interval** time is $O(\tau\log(\varepsilon_X)+v_I)$.

The search tree method's space requirements (in terms of path entries maintained in the cross-timeline structures) are measured by examining the data structures themselves instead of the algorithms which operate on them. We present two approaches to deriving this space requirement; one employs commutativity of sequences of **add_edge** operations, the other directly counts cross-timeline paths. For both approaches, it is a given that each timeline maintains knowledge of all paths between its own events; space requirements can not, therefore, be less than $O(v)$.

For the first approach, we remember what happens when an edge is added. The tail of the edge is ordered with respect to at most τ timelines, and, after the edge is added, the head must also be ordered with respect to those timelines; a potential of τ path entries must be recorded for each new edge. Our problem is that not only the head must be ordered with respect to the tail's timeline orderings: all events after the head must be ordered, also. Since there may be τ events ordered immediately after the head, this results in τ^2 potential path entries being added for the new edge. The question is whether or not this implies an $O(\tau^2\varepsilon_X)$ space requirement. The answer is no, because it is possible to rearrange the sequence of event additions —building the same history graph— so that there exist no events after the head of a new edge. This is because a history graph is a directed acyclic graph and thus possesses a topological ordering of vertices. If events (and thus edges) are added to the graph in topological order, no events yet exist after the head of each new edge and the space required per new edge is at most τ. This yields our desired $O(\tau\varepsilon_X+v)$ space complexity. Since an arbitrarily-created history graph and its corresponding topologically-created history graph are the same graph described with the same structures, they require the same space to store.

Our second approach counts the maximum cross-timeline paths directly. Each path is recorded only at its terminus, the head of its last component edge. There are exactly ε_X of these head events, and each one may be ordered with at most τ timelines. This argument again yields an $O(\tau\varepsilon_X+v)$ space complexity.

The above space complexity is a tight bound; the simple graph shown in Figure 6 exhibits this worst-case space requirement.

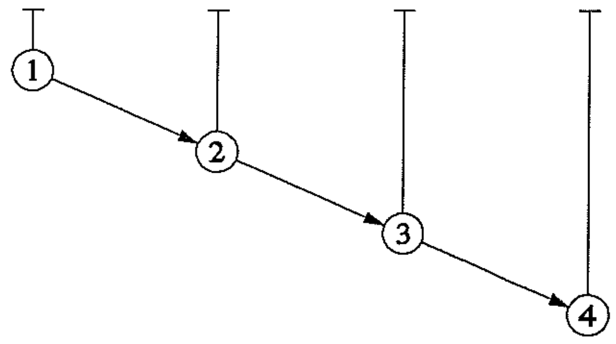


Figure 6. Worst-Case Space for Search Tree Method

4. Wavefront Method

4.1 Approach

The wavefront method, so named for the manner in which the **list_interval** query is satisfied, is our first approach which uses information about event characteristics to diminish complexity. Specifically, the cross-timeline information used by the search tree method is maintained only for end bound and tail candidate events (those with characteristics **e** or **t**). If the user is knowledgeable about the events which may still be incident with new edges, this optimization might save considerable space over the search tree method. Its cost is the loss of rapidly available complete transitive closure information.*

An example of this optimization is illustrated in Figure 7. Figure 7a presents a simple history graph. Figure 7b shows the search tree method's cross-timeline structures maintained for the second timeline of this graph, and Figure 7c shows the cross-timeline structures maintained by the wavefront method for the same timeline. The collapse of the cross-timeline structures of events 4, 5, and 6 into that of event 7 demonstrates a space savings over the search tree method, while the cross-timeline collapse from event 8 into event 9 merely moves data from one event to another (and loses information content in the process). Notice that records of the edges from event 4 to 5, 5 to 6, and 7 to 8 are also collapsed out of the wavefront method's cross-timeline structures (though they must be recorded elsewhere in order to satisfy a **list_interval** query).

Since complete transitive closure information is not readily available, it is not possible to immediately determine the first event on each timeline which occurs after an interval's start bound. In order to satisfy a **list_interval** query, a depth-first search originating at the start bound is used to determine the interval's events. This search terminates at the last event on each timeline which occurs before the interval's end bound (knowledge of which is maintained) and is pruned before leading to any timelines which are unordered with respect to the end bound.

4.2 Algorithm

Slight modifications to our basic data structures are necessary for implementation of the wavefront method. To facilitate the **list_interval** depth-first search, we add information to each event about all edge tails with which the event is incident. This is maintained as a circular list from the event through each such edge and back to the event; details are presented in Figure 8.

* Transitive closure information may very well, however, be regenerated efficiently over individual intervals when necessary for query purposes.

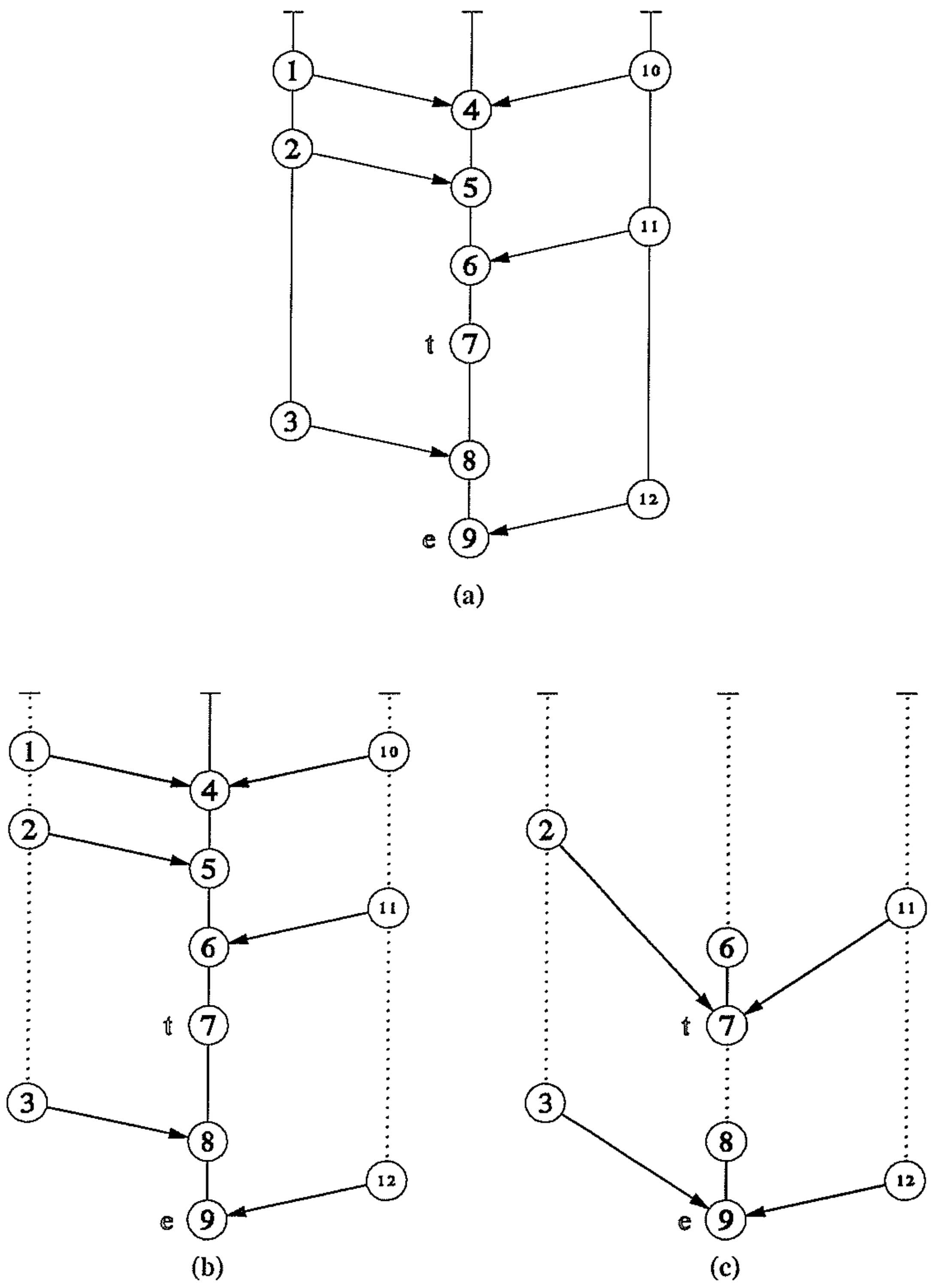


Figure 7. Cross-Timeline Path Information for Wavefront Method

As with the Search Tree Method, the pseudocode presented here is quite high-level. The more detailed code is found in Appendix 7.4.

Remain aware that, in the following algorithms, only events with characteristics **e** and **t** are maintained in the cross-timeline structures. "Consecutive" events recorded on the same timeline do not necessarily have immediately consecutive versions (though they will, of course, be in order). Furthermore, an event **B** referenced as an origin for a **precedes** path will later be

```
types
    next_edge = (edge_link, event_link);

    wv_event = record
        attrib : list of attribute;
        chr :    set of characteristic;
        out :    edge_id;
    end wv_event;

    wv_edge = record
        case link : next_edge of
            edge_link :  (next : edge_id;);
            event_link : (tail :  event_id;);
        endcase;
        head : event_id;
    end wv_edge;

    //  Versions of origin and terminus of a path from one timeline to another.
    //
    tl_path = record
        org :   version_index;
        term : version_index;
    end tl_path;

    wv_ordering = record
        vid : event_id;
        tid :  timeline_id;
        ver : version_index;
    end wv_ordering;

globals
    V : array [0..V_limit] of wv_event;          //  any O(1) access time structure
    e :  array [0..e_limit] of wv_edge;           //  any O(1) access time structure
```

Figure 8. Wavefront Method Data Structure Modifications

removed from the cross-timeline structures if its **e** and **t** characteristics are both removed. Even with **B** itself removed from the cross-timeline structures, though, virtually no references to **B** are altered since all lookups in these algorithms search relative to their target (≤ or ≥ the target's version). With this example, lookup results would either find some event **A** before **B** or some event **C** after **B**, whichever is appropriate.

The **add_event** procedure is similar to that of the search tree method; the only difference is the need to initialize the list of edges originating at the event.

```
procedure add_event (new_V : event, out vid : event_id);
begin
    vid := a unique event identifier;
    V[vid] := ⟨new_V.attrib, new_V.chr, id_null⟩;

    return;
end add_event;
```

The wavefront method's **add_edge** procedure (and thus **update_tl_xt**) is actually simpler than that of the search tree method, though almost identical in general approach. While the wavefront method must maintain the list of edges originating at each event, it does not treat an edge between two events on the same timeline as a special case.

```
procedure update_tl_xt(tl : ^timeline; ver : version_index;
                            pred : ordering_set);
    xt :  ^cross_tl_data;
    ord : ordering;
    term_ver : version_index;

begin
    term_ver := version of the first event on tl at or after ver with characteristic e or t;

    for each ord of pred do
        xt := tl's cross-timeline data for ord's timeline;

        if no existing data for that timeline then
            add a new cross-timeline structure to tl^.xtdata;
            add the initial $V_0$ to that structure;
        endif;

        if ord's information is not redundant then
            add ord → term_ver path to xt;
            remove information made redundant by ord;
        endif;
    endfor;

    return;
end update_tl_xt;
```

```
procedure add_edge (t_vid, h_vid : event_id; out eid : edge_id);
    tl : ^timeline;
    h_ver, term_ver : version_index;
    xt : ^cross_tl_data;
    prcd : ordering_set := new_srt_set;
begin
    //  Add edge to e and to list out of V[t_vid]
    //
    eid := a unique event identifier;
    if this is the first edge out of the tail event (V[t_vid]) then
        e[eid] := ⟨event_link, t_vid, h_vid⟩;
    else
        e[eid] := ⟨edge_link, V[t_vid].out, h_vid⟩;
    endif;
    V[t_vid].out := eid;

    //  Find all events which now precede the head (V[h_vid])
    //
    find any timeline on which the tail (V[t_vid]) is ordered;
    for each xt in that timeline's cross-timeline data do
        find the latest event on xt's timeline before the tail;
        if the event is not just $V_0$ then
            add that event to prcd;
        endif;
    endfor;

    for each timeline on which the tail is ordered do
        add the tail's version on that timeline to prcd;
    endfor;

    //  Update the head to follow prcd
    //
    for each timeline on which the head is ordered do
        tl := that timeline;
        h_ver := the head's version on that timeline;
        update_tl_xt(tl, h_ver, prcd);
    endfor;

    //  Update all events which the head precedes to follow prcd
    //
    find any timeline on which the head is ordered;
    for each tl of T do
        xt := tl's cross-timeline data for the head's timeline;
        if xt ≠ null andif there is an event on tl after the head then
            term_ver := the version of that event on tl;
            update_tl_xt(tl, term_ver, prcd);
        endif;
    endfor;

    return;
end add_edge;
```

The **rmchar** operation begins by removing the **chr** characteristics from event V[**rm_vid**], then checks whether or not that event still possesses the **e** or **t** characteristic. If not, it is removed from all timelines' cross-timeline structures. This is done, for each timeline on which V[**rm_vid**] is ordered, by finding the next event after V[**rm_vid**] and propagating V[**rm_vid**]'s path information to that next event.

```
procedure rmchar (rm_vid : event_id; chr : set of characteristic);
    xt : ^cross_tl_data;
    tl : ^timeline;
begin
    remove the characteristics in chr from V[rm_vid].chr;
    //      If this operation made V[rm_vid] have neither the e nor t characteristic, remove
    //  it from all xt structures.
    //
    if V[rm_vid] now has neither the e nor the t characteristic then
        for each timeline on which V[rm_vid] is ordered do
            tl := that timeline;
            find the next event on tl after V[rm_vid];
            //      Remove V[rm_vid] and propagate its path information, if any, to the
            //  next event.
            //
            for each xt in tl's cross-timeline data do
                if V[rm_vid] was the terminus of any path from that other timeline then
                    remove the record of that path to V[rm_vid] from xt;
                    //      If a path to the next event already exists, it is from a
                    //  higher-version origin than that to V[rm_vid] and should not be
                    //  overwritten.
                    //
                    if there is not already a path to the next event after V[rm_vid] then
                        record the path to V[rm_vid] as going to that next event;
                    endif;
                endif;
            endfor;
        endif;
    return;
end rmchar;
```

The **list_interval** operation proceeds as a series of passes between two sets of bounds, todo_set and **done_set**. The earliest (smallest version) event on each timeline which is known to follow V[s_vid] but which has not yet been added to the interval list is stored in **todo_set**;

done_set contains the earliest event on each timeline which should no longer be added to the interval list, either because it has already been added or because it is known to not be in the interval. The initial value of **done_set** is those events one version after the latest events on each timeline which **precede** V[e_vid] and the next event after V[e_vid]; **todo_set** begins with V[s_vid]. Note that only those timelines with which V[e_vid] is ordered have an entry in **done_set**. A failed reference to any timeline is therefore considered to mean that the entire timeline is "done" as far as **list_interval** is concerned.

During execution, **todo_set** is broadened to contain an entry for another timeline whenever an edge extends from the currently scanned event to some event **B** on a different timeline, so long as **B** is not "done." A timeline's entry in **todo_set** may be pulled back to an earlier version when new edges are encountered. Timeline entries in **done_set** are updated at the beginning of every pass to reflect the span of versions to be added to the interval list during that pass.

```
function list_interval (s_vid, e_vid : event_id) : list of event_id;
    Vlist : srt_set of event_id := new_srt_set; //   avoid duplicates
    xt : ^cross_tl_data;
    doing, next : wv_ordering;
    todo_set : srt_set of wv_ordering key tid := new_srt_set;
    done_set : ordering_set := new_srt_set;

begin
    //      Find the latest event before $V_e$ (V[e_vid]) on each timeline with which $V_e$ is
    //   ordered.
    //
    find any timeline on which $V_e$ is ordered;
    for each xt in that timeline's cross-timeline data do
        if xt is not for $V_e$'s timeline itself then
            find the latest event on xt's timeline before $V_e$;
        else                                    //   this will lead to putting $V_e$ in Vlist
            the event we use is $V_e$ itself;
        endif;
        if the event is not just $V_0$ then
            add the event just after the one found above to done_set;
        endif;
    endfor;
```

```
//      Add all events after $V_s$ and before $V_e$ to Vlist, doing one segment of a timeline
//   at a time.
//
find any timeline on which $V_s$ (V[s_vid]) is ordered;
if $V_e$ is at or after $V_s$ then
    start todo_set at $V_s$;

    while todo_set not empty do
        doing := any event of todo_set, removing it from todo_set after the selection;
        note where, according to done_set, this pass should end;
        update done_set to show we have scanned beginning at doing;

        while we are not done with this pass do
            add doing to Vlist;

            //  Find where event 'doing' leads.
            //
            next := end of pass;                //  in case end of timeline
            for each edge with tail at doing do
                find the head of that edge;
                for each timeline on which the head is ordered do
                    if this timeline is doing's timeline then
                        next := the edge's head;
                    else
                        if $V_e$ is after the head andif the head is not already scanned
                          andif ( the head is earlier than any event we already know
                                  we need to scan on the head's timeline ) then
                          add the head to todo_set;
                        endif;
                    endif;
                endfor;
            endfor;
            doing := next;
        endwhile;
    endwhile;
endif;

return srt_set_to_list(Vlist);
end list_interval;
```

4.3 Analysis

For this analysis, we define $v_W \equiv$ the number of events with either the **e** or **t** characteristic. It is more difficult to measure ε_W, the number of events which are recorded as incident with heads of cross-timeline edges. With the search tree method, this was simply ε_X;

with the wavefront method's collapse of perhaps several events' **precede** information into that of the next event with either the e or t characteristic, the measure will be something $\leq \varepsilon_X$. It will not, however, necessarily be the count of those events which are both at the head of a cross-timeline edge and have either the e or t characteristic, $v_{X \cap W}$. The wavefront method's cross-timeline structures, remember, might simply <u>move</u> **precedes** information from one event to another, not necessarily performing any combination of information at all. The limit of what we can safely determine is that $v_{X \cap W} \leq \varepsilon_W \leq \varepsilon_X$.

The **add_edge** time for the wavefront method is derived effectively the same as for the search tree method and is $O(\tau^2 \log(\varepsilon_W) + \tau \log(v_W))$. Examination of the **rmchar** pseudocode reveals this same time complexity. An event A may be on τ timelines, each ordered with τ others, and updating the path information takes $O(\log(\varepsilon_W))$ time if that information is not for A's own timeline, or $O(\log(v_W))$ if it is.

Initialization for the wavefront method's **list_interval** involves **done_set** in a similar manner as does the search tree method's **list_interval** initialization with **mid_e_set**. The required time is $O(\tau \log(\varepsilon_W))$. During scanning from **todo_set** to **done_set**, **list_interval** might require an $O(\log\tau)$ update to **done_set** for each of v_I events within the interval. Also, for each cross-timeline edge whose tail is incident with an event in the interval (let the count of such edges be $\varepsilon_{W \cap I}$), there is at least an $O(\log\tau)$ search and perhaps an $O(\log\tau)$ update of **todo_set**. Total time for **list_interval** is therefore $O(\tau \log(\varepsilon_W) + \log\tau(v_I + \varepsilon_{W \cap I}))$.

The wavefront method's space requirement, not surprisingly, is also derived in a similar manner to that of the search tree method. This requirement is $O(\tau\varepsilon_W + v_W)$.

5. Bounded-Search Method

5.1 Approach

The fourth method we have investigated to satisfy a **list_interval** query is one which attempts to minimize the method's space requirement at the expense of speed. No cross-timeline information is maintained except, of course, the edges themselves. The **list_interval** operation is performed by what appears, at first, to be a sequence of two brute-force depth-first searches: one from the start bound forward, the other from the end bound back. The interval is known when the two searches meet at a common events.

It is obvious that a simple search from the start bound forward will terminate only at the end bound itself or the end of the history graph. This, alone, might not be too terrible if queries are made shortly after the end bound becomes known. The search back from the end bound, however, will not necessarily end until the beginning of the history graph: potentially thousands of events (or more) will be uselessly scanned. We must bracket this backwards search and, preferably, the forward search as well.

The searches are limited by maintaining knowledge of the topological order of events. Topological order requires that, for two events A and B, **top(A) < top(B)** if **A $\prec$ B**. Note that this is if, not iff. Maintaining topological numbering is trivial if events are added in topological order, but requires the use of a "differences" tree* or pruned O(ε) renumbering when events are not added in order.

We begin **list_interval** with a forward depth-first search from **S**, the start bound, towards **E**, the end bound. Each probe of the search is stopped when an event **A** is encountered such that **top(A) $\geq$ top(E)**. This guarantees that we have not searched past **E**, but does not imply that all events which have been scanned are in the desired interval (S $\prec$ A $\not\succ$ E). A second search back from **E** finishes **list_interval**. Probes of this search stop when some event scanned by the first search is found, or when an event **B** is encountered such that **top(B) $\leq$ top(S)**. In other words, when **B** can no longer be after S and thus can not be in the interval (B $\not\succ$ S). When, as described here, the full forward search is performed before the backwards search is done, one only need

* Such a data structure maintains, at each node, the difference of some attribute between itself and its parent. This allows the search for a node X to calculate the value of X's attribute by summation along the path to X, and also allows adding a constant to the attribute of all nodes after X by adjusting X's attribute difference.

search back one step (it is for variations on this approach that we need the topological bound on the backwards search).

Optimizations to this algorithm might involve heuristics which perform breadth-first searches between S and E, alternating between the searches in hope that they will "meet in the middle." Another possibility is to delay updating the topological ordering until it is required by a **list_interval**, expecting that many intermediate updates might not need to be performed.

6. Future Work

6.1 Simulation

We wish to compare actual time and space characteristics of the search tree and wavefront methods, as well as perhaps the bounded-search method. A simulation driver is being developed which will allow a variety of loads to be placed on the algorithms, generating from purely random cross-timeline edges to localized "communication" between timelines such as that experienced in a hypercube.

6.2 Enhanced Queries

One of the advantages of interval logic is its ability to express nested intervals. The algorithms presented in this document address only the problem of simple intervals. Their extension to nested intervals is of considerable importance.

The **list_interval** query, as defined, returns only those events which must be after the start bound S and before the end bound E. In many situations, however, it may be desirable to know those events which could be after S and before E. This issue of temporal ambiguity is one inherent in distributed time, the area of our algorithms' application, and should be addressed.

A potential disadvantage of the wavefront method is that it does not maintain complete transitive closure information. Since some history graph queries might find such information necessary, it is important to know how difficult it is to generate transitive closure information for an interval listed by the wavefront method.

6.3 Distributed Implementations

The application utilizing the algorithms presented in this paper is the temporal analysis of events generated in a distributed system. It is therefore useful to know whether the analysis algorithms themselves can be distributed, or instead require a centralized control (i.e. a potential bottleneck). We currently believe that the search tree and wavefront methods can be distributed without excessive inter-process communication; maintenance of the topological numbering used by the wavefront method appears to best be performed in a centralized manner, though we are not certain of this. The interaction of distributing the algorithms along with supporting enhanced queries is an area of tradeoffs and perhaps considerable future investigation.

7. APPENDICES

APPENDIX 7.1 PSEUDOCODE REPRESENTATION

The representation of algorithms in this report is done using pseudocode which resembles a mixture of Pascal, Ada, and C++. All the standard control structures are available, defined types may be expressed, and a variety of operators may be used.

Below are listed the details of this representation. In pseudocode tradition, however, the more obvious operations in our algorithms are generally expressed with a certain amount of English instead of detailed statements (such as "for every child of..." instead of "child:= foo^.child; while child ≠ null do..."). When such use of English is made instead of formal code, this will be clarified by italicizing any English in our algorithms (e.g. "**for** *every* child *of*..." in the above example).

In the following discussion, bold brackets (**[ ]**) indicate 0 or 1 occurrence of the enclosed item, and bold braces (**{ }**) indicate 0 or more occurrences. Comments in this pseudocode are as in C++: '//' indicates that the rest of the line is a comment.

7.1.1 CONTROL STRUCTURES

Flow of control is Ada-like. Semicolons are statement terminators, not separators, and loop entry statements are paired with matching loop exit statements. Procedures and functions may be defined and nested, following the usual scope rules. Syntax is:

```
Sequence            Conditional             Alternative
statement;          if condition then       case expression of
{statement;}            sequence;               value_list:
                    else                            (sequence;);
                        sequence;               ...
                    endif;                      others:
                                                    (sequence;);
                                            endcase;

Iteration           Repetition, Test At Entry   Repetition, Test At Exit
for variable in range do   while condition do       repeat
    sequence;                  sequence;                sequence;
endfor;                    endwhile;                until condition;
```

```
              Procedure                                      Function
procedure proc_name(formal_parameters);       function func_name(formal_parameters) :
    declarations;                                                 result_type;
begin                                             declarations;
    sequence;                                 begin
    return;                                       sequence;
end proc_name;                                    return value;
                                              end func_name;
```

— where *formal_parameters* is a list, the elements of which are separated by semicolons and have the form *variable_name*{, *variable_name*} : *type*

7.1.2 OPERATORS

assignment:	:=	// *var := value*
arithmetic:	+, -, *, /, %	// add, subtract, multiply, divide, modulus
arithmetic assign:	+=, -=, *=, /=, %=	// *var op= value* ≡ *var* := *var op value*
comparison:	=, ≠, <, ≤, >, ≥	
logical:	and, or, xor, not, andif, orelse	// two "short circuit" operators

7.1.3 SIMPLE AND STRUCTURED TYPES

Basic types include the standard **integer**, **real**, **Boolean**, **character**. Derived types include enumerations and subranges of any ordinal type. Structure is expressed by use of **array**, **record**, and pointer types which may be arbitrarily nested. Similar to Pascal, records may have variant fields. Syntax is:

```
      Subrange              Enumeration              Array
subrange_type =         enumeration_type =      array_type =
    range [first..last]     (value{, value});       array [range{, range}]
        of base_type;                               of base_type;
```

Record

```
record_type = record
    field_name : type;
    ...
end record_type;
```

Variant Record

```
record_type = record
  {[field_name : type;]
   [case [tag :] type of
     value_list:
          (field_name : type;
           ... );
     ...
     others:
          (field_name : type;
           ... );
   endcase;]}
end record_type;
```

Pointer

```
pointer_type = ^base_type;
```

Pointer Dereference

```
pointer_variable^
```

7.1.4 HIGH-LEVEL STRUCTURED TYPES

Collections of elements of any other type may be built as sets, lists, and sorted sets (search trees). The syntax for declaring such collections and the operations allowed with them are as follows:

Sets

Sets are defined as unordered collections of objects with no duplicates. Basic set operations of union, intersection, symmetric difference, proper subset and superset, construction, and element containment may be expressed $\cup$, $\cap$, -, $\subset$, $\supset$, { *element*{, *element*} } and $\in$, respectively.

declaration:	*type_name* = set of *base_type*;
operators:	$\cup$, $\cap$, -, =, $\subset$, $\subseteq$, $\supset$, $\supseteq$, $\in$, and the assignment operators $\cup$=, $\cap$=, and -=
constants:	$\varnothing$ — the empty set

Lists

Lists are defined as collections of objects ordered by their sequence of appearance within the list; duplicates are allowed. Operations include concatenation, construction, element reference, and sublist reference expressed by &, [*element*{, *element*}], *list*(*element_number*), and *list*[*element_range*], respectively.

declaration:	*type_name* = list of *base_type*;
operators:	&, (*element_number*), [*element_range*], and the assignment operator &=
constants:	[ ] — the empty list

Sorted Sets

Sorted sets are defined as collections of objects ordered by means of a "key" value, with no duplicate key values allowed between two elements. This key may either be the element itself, if the sorted set is of a simple type, or is the value of one field of an element, if the sorted set is of a record type. Operations include insertion and removal of elements and search according to a key.

Insertion of an element into a sorted set either adds an entirely new element or replaces an existing element of the same key. This operation is expressed as *set* + *element*. Removal of an element from a sorted set, expressed as *set* - *element*, fails if the element is not part of the sorted set. Reference to an element by key has many search criteria and returns a pointer to that element (or **null** if no such element is found). The search may be for the element with key equal to the search key ('=' search); for the element with the greatest key less than the search key ('<' search); for the element either with the search key or, if not found, with the greatest key less than the search key ('≤' search); and so on for '>' and '≥' search. Equal-to search is common enough to be expressed as *sorted_set*[*key*]; searches with other criteria are expressed as *sorted_set*(*criterion*, *key*).

declaration:	*type_name* = srt_set of *base_type* [key *field_name*];
operators:	+, -, [*key*] — equivalent to '=' *criterion* below,
	(*criterion*, *key*), where *criterion* is one of =, <, >, ≤, or ≥
constants:	new_srt_set — the empty sorted set

APPENDIX 7.2 ITALIANO'S PATH RETRIEVAL ALGORITHM

Developed by Giuseppe F. Italiano, the following data structures and algorithms permit the incremental construction of a directed acyclic graph $G = \langle V, E\rangle$ in such a way that queries may be made in order to check for the existence of a path between any two vertices in G and to report the vertices along a path between any origin and terminus vertices in G.[5] Edges are added and paths reported in O(v) amortized time per operation, $v \equiv |V|$; the existence of a path may be checked in O(1) (constant) time. The data structures require $\Theta(v^2)$ space.

```
constants
    V_limit : integer := some large positive number      //  greatest # of elements

types
    event_id = range [0..] of integer;                    //  used as indices, not just as ids

    Ital_node = record
        key :     event_id;
        parent : ^Ital_node;
        child :   ^Ital_node;
        sibling : ^Ital_node;
    end Ital_node;

globals
    //  index[i, j] $\neq$ null $\longrightarrow$ a path exists from $v_i$ to $v_j$
    //      If the path exists, this points to $v_j$ in the descendent tree
    //  of $v_i$.
    //
    index : array [0..V_limit, 0..V_limit] of ^Ital_node := null;

    //  Trees of all descendants of each vertex in the graph
    //
    desc :  array [0..V_limit] of ^Ital_node;
```

```
procedure Ital_initialize();
    i, j : event_id;
begin
    for i := 0 to V_limit do
        desc[i] := new(Ital_node);
        desc[i]^ := ⟨i, null, null, null⟩;
        for j := 0 to V_limit do index[i, j] := null; endfor;
    endfor;
    return;
end Ital_initialize;
```

```
function Ital_check_path (org, term : event_id) : Boolean;
begin
    return index[org, term] ≠ null;
end Ital_check_path;
```

```
function Ital_get_path (org, term : event_id) : list of event_id;
    Vlist : list of event_id := [ ];              //  path from origin to terminus
    curr_vertex : ^Ital_node;
begin
    if index[org, term] ≠ null then               //  terminus is reachable from origin
        curr_vertex := index[org, term];          //  locate terminus in desc[origin]
        Vlist := [term];
        repeat                                    //  go up in desc[origin]
            curr_vertex := curr_vertex^.parent;
            Vlist := [curr_vertex^.key] & Vlist;
        until curr_vertex^.parent = null;         //  ...until the root origin
    endif;
    return Vlist;
end Ital_get_path;
```

```
procedure Ital_add_edge (tail, head : event_id);
    x : event_id;
begin
    if index[tail, head] = null then                  //  no previous path from tail to head
        for x := 0 to V_limit do
            if index[x, tail] ≠ null and index[x, head] = null then
                //  The edge ($v_t$, $v_h$) gives rise to a new path from $v_x$ to $v_h$
                //
                meld(x, head, tail, head);            //  update desc[x] by means of desc[head]
            endif;
        endfor;
    endif;

    return;
end Ital_add_edge;


//      Merge desc[meldto] with a pruned subtree of desc[meldwith] rooted at with_subtree.
//  The vertex of desc[meldto] to which the pruned subtree will be linked is linkto.  By
//  "pruning," we mean removing those vertices already in desc[meldto].
//
procedure meld(meldto, meldwith, linkto, with_subtree : event_id);
    parent, child : ^Ital_node;
begin
    //  Insert the root of with_subtree into desc[meldto] as a child of linkto
    //
    index[meldto, with_subtree] := new(Ital_node);
    if meldto = linkto then                           //  index does not contain self-loops
        parent := desc[linkto];
    then
        parent := index[meldto, linkto];
    endif;
    index[meldto, with_subtree]^ :=                   //  ⟨key, parent, child, sibling⟩
        ⟨with_subtree, parent, null, parent^.child⟩;
    parent^.child := index[meldto, with_subtree];

    for each child of with_subtree in desc[meldwith] do
        //  If the child and its subtree are not already in desc[meldto], add them
        //
        if index[meldto, child^.key] = null then
            meld(meldto, meldwith, with_subtree, child^.key);
        endif;
    endfor;

    return;
end meld;
```

APPENDIX 7.3 SEARCH TREE METHOD ALGORITHM

The following data structures and algorithms detail the Search Tree Method of interval detection as presented in this report.

```
constants
    V_limit, e_limit : integer := some large positive number    //  greatest # of elements
    id_null : integer := -1;                                    //  "no such object"

types
    natural = range [0..] of integer;
    event_id, edge_id, timeline_id = range [id_null..] of integer;
    version_index = natural;

    attribute_type = (timeline, other);          // we are unconcerned about other attribute
    ordering = record                            //  to order an event w.r.t. a specific timeline
        tid : timeline_id;
        ver : version_index;
    end ordering;

    attribute = record
        case atype : attribute_type of
            timeline : (value_tl : ordering;);
            other :    (value : tuple;);         // arbitrary structure
        endcase;
    end attribute;

    characteristic = (t, h, s, e);               // edge tail or head, interval start or end

    event = record
        attrib : list of attribute;
        chr :    set of characteristic;
    end event;

    edge = record
        tail, head : event_id;
    end edge;

    ordering_set = srt_set of ordering key tid;
    //      Versions of origin and terminus of a path from one timeline to another.  If
    //  both timelines are identical, the origin's version is replaced with the event
    //  identifier of the terminus since the origin's version would simply be terminus
    //  version - 1.
    //
    tl_path = record
        case (cross_timeline, in_timeline) of
            cross_timeline : (org : version_index;);
            in_timeline :    (vid : event_id;);
        endcase;
        term : version_index;
    end tl_path;

    cross_tl_data = record
        org_tid :     timeline_id;              //  tl id of origins of recorded paths
        p_by_org :  srt_set of tl_path key org;
        p_by_term : srt_set of tl_path key term;
    end cross_tl_data;

    timeline = record
        tid :     timeline_id;
        xtdata : srt_set of cross_tl_data key org_tid;
    end timeline;

globals
    V : array [0..V_limit] of event;           //  any O(1) access time structure
    e : array [0..e_limit] of edge;            //  any O(1) access time structure
    T : srt_set of timeline key tid;



procedure add_event (new_V : event, out vid : event_id);
begin
    vid := new_event_id();
    V[vid] := new_V;

    return;
end add_event;
```

```
procedure update_tl_xt(tl : ^timeline; ver : version_index;
                         vid : event_id; pred : ordering_set);
    xt :   ^cross_tl_data;
    ord :  ordering;
    path : ^tl_path;
begin
    for each ord of pred do
        xt := tl^.xtdata[ord.tid];
        if xt = null then                          //   new xt, possibly with self
            tl^.xtdata += ⟨ord.tid, new_srt_set, new_srt_set⟩;
            xt := tl^.xtdata[ord.tid];
            //   $V_0$ is on each timeline
            //
            if ord.tid ≠ tl^.tid then              //   between tl and another timeline
                //       make it before first tl event, which might no longer be version 0
                //   if garbage collection has taken place
                //
                path := tl^.xtdata[tl^.tid]^.p_by_term(≥, 1);
                xt^.p_by_org += ⟨0, path^.term⟩;
                xt^.p_by_term += ⟨0, path^.term⟩;
            else                                   //   tl with self
                xt^.p_by_term += ⟨vid, 1⟩;
            endif;
        endif;
        if ord.tid ≠ tl^.tid then
            if xt^.p_by_term(≤, ver)^.org < ord.ver then
                xt^.p_by_org += ⟨ord.ver, ver⟩;    //       overwrites previous
                xt^.p_by_term += ⟨ord.ver, ver⟩;   //   ordering, if any
                //   Remove out-of-order information
                //
                path := xt^.p_by_term(>, ver);
                while path ≠ null andif path^.org ≤ ord.ver do
                    xt^.p_by_org -= path;
                    xt^.p_by_term -= path;
                    path := xt^.p_by_term(>, ver);
                endwhile;
            endif;
        else                                       //   tl's xt for itself
            if xt^.p_by_term(≤, ver)^.term - 1 < ord.ver then
                xt^.p_by_term += ⟨vid, ver⟩;
            endif;
        endif;
    endfor;
    return;
end update_tl_xt;
```

```
procedure add_edge (t_vid, h_vid : event_id; out eid : edge_id);
    attrib : attribute;
    h_tid : timeline_id;
    h_ver, org_ver, term_ver : version_index;
    tl : ^timeline;
    xt : ^cross_tl_data;
    prcd : ordering_set := new_srt_set;
begin
    eid := new_edge_id();
    e[eid] := ⟨t_vid, h_vid⟩;

    //  Find all events which now precede V[h_vid]
    //
    attrib := any attrib of V[t_vid] with attrib.atype = timeline;
    tl := T[attrib.value_tl.tid];
    term_ver := attrib.value_tl.ver;
    for each xt in tl^.xtdata do
        if xt^.org_tid ≠ tl^.tid then              //  handled below (avoid invalid org ref)
            org_ver := xt^.p_by_term(≤, term_ver)^.org;
            if org_ver > 0 then                    //  everything comes after $V_0$; ignore it
                prcd += ⟨xt^.org_tid, org_ver⟩;
            endif;
        endif;
    endfor;

    //      Check if the tail is V[0].  If so, we want only the head's timelines, not
    //  all the timelines in the graph (that would be quite inefficient, though not
    //  actually wrong).
    //
    if t_vid ≠ 0 then
        for each attrib of V[t_vid] with attrib.atype = timeline do
            prcd += attrib.value_tl;               //  overwrites previous ordering, if any
        endfor;
    else
        for each attrib of V[h_vid] with attrib.atype = timeline do
            prcd += attrib.value_tl;               //  overwrites previous ordering, if any
        endfor;
    endif;

    //  Update V[h_vid] to follow prcd
    //
    for each attrib of V[h_vid] with attrib.atype = timeline do
        tl := T[attrib.value_tl.tid];
        h_ver := attrib.value_tl.ver;
        update_tl_xt(tl, h_ver, h_vid, prcd);
    endfor;
    //  Update all events which V[h_vid] precedes to follow prcd
    //
    attrib := any attrib of V[h_vid] with attrib.atype = timeline;
    h_tid := attrib.value_tl.tid;
    h_ver := attrib.value_tl.ver;
    for each tl of T with tl^.tid ≠ h_tid do
        xt := tl^.xtdata[h_tid];
        if xt ≠ null andif xt^.p_by_org(≥, h_ver) ≠ null then
            term_ver := xt^.p_by_org(≥, h_ver)^.term;
            update_tl_xt(tl, term_ver, id_null, prcd);
        endif;
    endfor;

    return;
end add_edge;


function list_interval (s_vid, e_vid : event_id) : list of event_id;
    Vlist : srt_set of event_id := new_srt_set; //  avoid duplicates
    mid_e_set : ordering_set := new_srt_set;
    attrib : attribute;
    ord :   ordering;
    s_tid : timeline_id;
    tl :    ^timeline;
    xt :    ^cross_tl_data;
    s_ver, e_ver, mid_e_ver : version_index;
    mid_s, path : ^tl_path;
begin
    //      Find the latest event before V[e_vid] on each timeline with which V[e_vid] is
    //  ordered.
    //
    attrib := any attrib of V[e_vid] with attrib.atype = timeline;
    tl := T[attrib.value_tl.tid];
    e_ver := attrib.value_tl.ver;
    for each xt of tl^.xtdata do
        if xt^.org_tid ≠ tl^.tid then
            mid_e_ver := xt^.p_by_term(≤, e_ver)^.org;
        else                                    //  this will lead to putting V[e_vid] in Vlist
            mid_e_ver := e_ver;
        endif;
        if mid_e_ver > 0 then
            mid_e_set += ⟨xt^.org_tid, mid_e_ver⟩;
        endif;
    endfor;
    //      Add all events after V[s_vid] and before V[e_vid] to Vlist, doing one timeline
    //  at a time between the first event after V[s_vid] and the latest event before V[e_vid]
    //  (stored in mid_e_set).
    //
    attrib := any attrib of V[s_vid] with attrib.atype = timeline;
    s_tid := attrib.value_tl.tid;
    s_ver := attrib.value_tl.ver;
    for each ord of mid_e_set do
        tl := T[ord.tid];
        xt := tl^.xtdata[s_tid];
        if xt ≠ null then
            if xt^.org_tid ≠ tl^.tid then
                mid_s := xt^.p_by_org(≥, s_ver);
                if mid_s ≠ null andif mid_s^.term ≤ ord.ver then
                    xt := tl^.xtdata[tl^.tid];
                    for each path of xt^.p_by_term
                        with path^.term ∈ [mid_s^.term, ord.ver] do
                        Vlist += path^.vid;
                    endfor;
                endif;
            else                                //  V[s_vid] is on this timeline
                if s_ver ≤ ord.ver then
                    for each path of xt^.p_by_term
                        with path^.term ∈ [s_ver, ord.ver] do
                        Vlist += path^.vid;
                    endfor;
                endif;
            endif;
        endif;
    endfor;

    return srt_set_to_list(Vlist);
end list_interval;
```

APPENDIX 7.4 WAVEFRONT METHOD ALGORITHM

The following data structures and algorithms detail the Wavefront Method of interval detection as presented in this report.

```
types
    next_edge = (edge_link, event_link);

    wv_event = record
        attrib : list of attribute;
        chr :    set of characteristic;
        out :    edge_id;
    end wv_event;

    wv_edge = record
        case link : next_edge of
            edge_link :  (next : edge_id;);
            event_link : (tail :  event_id;);
        endcase;
        head : event_id;
    end wv_edge;

    //  Versions of origin and terminus of a path from one timeline to another.
    //
    tl_path = record
        org :  version_index;
        term : version_index;
    end tl_path;

    wv_ordering = record
        vid : event_id;
        tid :  timeline_id;
        ver : version_index;
    end wv_ordering;

globals
    V : array [0..V_limit] of wv_event;        //  any O(1) access time structure
    e :  array [0..e_limit] of wv_edge;         //  any O(1) access time structure


procedure add_event (new_V : event, out vid : event_id);
begin
    vid := new_event_id();
    V[vid] := ⟨new_V.attrib, new_V.chr, id_null⟩;
    return;
end add_event;
```

```
procedure update_tl_xt(tl : ^timeline; ver : version_index;
                       pred : ordering_set);
    xt :   ^cross_tl_data;
    ord :  ordering;
    path : ^tl_path;
    term_ver : version_index;
begin
    //   Find first event with characteristic e or t at or after ver
    //
    path := tl^.xtdata[tl^.tid]^.p_by_term(≥, ver);
    if path = null then
        term_ver := ver;
    else
        term_ver := path^.term;
    endif;
    for each ord of pred do
        xt := tl^.xtdata[ord.tid];
        if xt = null then                        //   new xt, possibly with self
            tl^.xtdata += ⟨ord.tid, new_srt_set, new_srt_set⟩;
            xt := tl^.xtdata[ord.tid];
            //   V_0 is on each timeline
            //
            if ord.tid ≠ tl^.tid then               //   between tl and another timeline
                //        make it before first tl event, which might no longer be version 0
                //   if garbage collection has taken place
                //
                path := tl^.xtdata[tl^.tid]^.p_by_term(≥, 1);
                xt^.p_by_org += ⟨0, path^.term⟩;
                xt^.p_by_term += ⟨0, path^.term⟩;
            else                                    //   tl with self
                xt^.p_by_org += ⟨0, 1⟩;
                xt^.p_by_term += ⟨0, 1⟩;
            endif;
        endif;
        if xt^.p_by_term(≤, ver)^.org < ord.ver then
            xt^.p_by_org += ⟨ord.ver, term_ver⟩;       //        overwrites previous
            xt^.p_by_term += ⟨ord.ver, term_ver⟩;      //   ordering, if any
            //   Remove out-of-order information
            //
            path := xt^.p_by_term(>, term_ver);
            while path ≠ null andif path^.org ≤ ord.ver do
                xt^.p_by_org -= path;
                xt^.p_by_term -= path;
                path := xt^.p_by_term(>, term_ver);
            endwhile;
        endif;
    endfor;
    return;
end update_tl_xt;
```

```
procedure add_edge (t_vid, h_vid : event_id; out eid : edge_id);
    attrib : attribute;
    h_tid : timeline_id
    h_ver, org_ver, term_ver : version_index;
    tl : ^timeline;
    xt : ^cross_tl_data;
    pred : ordering_set := new_srt_set;
begin
    //  Add edge to e and to list out of V[t_vid]
    //
    eid := new_edge_id();
    if V[t_vid].out = id_null then
        e[eid] := ⟨event_link, t_vid, h_vid⟩;
    else
        e[eid] := ⟨edge_link, V[t_vid].out, h_vid⟩;
    endif;
    V[t_vid].out := eid;

    //  Find all events which now precede V[h_vid]
    //
    attrib := any attrib of V[t_vid] with attrib.atype = timeline;
    tl := T[attrib.value_tl.tid];
    term_ver := attrib.value_tl.ver;
    for each xt in tl^.xtdata do
        org_ver := xt^.p_by_term(≤, term_ver)^.org;
        if org_ver > 0 then                    //  everything comes after $V_0$; ignore it
            pred += ⟨xt^.org_tid, org_ver⟩;
        endif;
    endfor;

    //      Check if the tail is V[0]. If so, we want only the head's timelines, not
    //  all the timelines in the graph (that would be quite inefficient, though not
    //  actually wrong).
    //
    if t_vid ≠ 0 then
        for each attrib of V[t_vid] with attrib.atype = timeline do
            pred += attrib.value_tl;           //  overwrites previous ordering, if any
        endfor;
    else
        for each attrib of V[h_vid] with attrib.atype = timeline do
            pred += attrib.value_tl;           //  overwrites previous ordering, if any
        endfor;
    endif;
    //  Update V[h_vid] to follow prcd
    //
    for each attrib of V[h_vid] with attrib.atype = timeline do
        tl := T[attrib.value_tl.tid];
        h_ver := attrib.value_tl.ver;
        update_tl_xt(tl, h_ver, prcd);
    endfor;

    //  Update all events which V[h_vid] precedes to follow prcd
    //
    attrib := any attrib of V[h_vid] with attrib.atype = timeline;
    h_tid := attrib.value_tl.tid;
    h_ver := attrib.value_tl.ver;
    for each tl of T do
        xt := tl^.xtdata[h_tid];
        if xt ≠ null andif xt^.p_by_org(≥, h_ver) ≠ null then
            term_ver := xt^.p_by_org(≥, h_ver)^.term
            update_tl_xt(tl, term_ver, prcd);
        endif;
    endfor;

    return;
end add_edge;
```

```
procedure rmchar (rm_vid : event_id; chr : set of characteristic);
    attrib : attribute;
    path :  ^tl_path;
    xt :    ^cross_tl_data;
    tl :    ^timeline;
    rm_ver, next_ver : version_index;
begin
    V[rm_vid].chr -= chr;

    //      If this operation made V[rm_vid] have neither the e nor t characteristic, remove
    //  it from all xt structures.
    //
    if not (e ∈ V[rm_vid].chr or t ∈ V[rm_vid].chr) then
        for each attrib of V[rm_vid] with attrib.atype = timeline do
            tl := T[attrib.value_tl.tid];
            rm_ver := attrib.value_tl.ver;

            //  Get the next event after V[rm_vid]
            //
            xt := tl^.xtdata[tl^.tid];
            next_ver := xt^.p_by_term(>, rm_ver)^.term;

            //      Remove V[rm_vid] and propagate its path information, if any, to the
            //  next event.
            //
            for each xt of tl^.xtdata do
                path := xt^.p_by_term[rm_ver];
                if path ≠ null then
                    xt^.p_by_org -= path^;
                    xt^.p_by_term -= path^;

                    //      If a path to the next event already exists, it is from a
                    //  higher-version origin than that to V[rm_vid] and should not be
                    //  overwritten.
                    //
                    if xt^.p_by_term[next_ver] = null then
                        xt^.p_by_org += ⟨path^.org, next_ver⟩;
                        xt^.p_by_term += ⟨path^.org, next_ver⟩;
                    endif;
                endif;
            endfor;
        endfor;
    endif;

    return;
end rmchar;
```

```
function list_interval (s_vid, e_vid : event_id) : list of event_id;
    Vlist : srt_set of event_id := new_srt_set;       //   avoid duplicates
    attrib : attribute;
    doing, next : wv_ordering;
    out : edge_id;
    tl :   ^timeline;
    xt :   ^cross_tl_data;
    h_vid : event_id;
    s_tid, h_tid : timeline_id;
    s_ver, e_ver, h_ver, done_ver : version_index;
    todo_set : srt_set of wv_ordering key tid := new_srt_set;
    done_set : ordering_set := new_srt_set;

begin
    //      Find the latest event before V[e_vid] on each timeline with which V[e_vid] is
    //   ordered.
    //
    attrib := any attrib of V[e_vid] with attrib.atype = timeline;
    tl := T[attrib.value_tl.tid];
    e_ver := attrib.value_tl.ver;
    for each xt of tl^.xtdata do
        if xt^.org_tid ≠ tl^.tid then
            done_ver := xt^.p_by_term(≤, e_ver)^.org;
        else                                     //   this will lead to putting V[e_vid] in Vlist
            done_ver := e_ver;
        endif;
        if done_ver > 0 then
            done_set += ⟨xt^.org_tid, done_ver + 1⟩;
        endif;
    endfor;

    //      Add all events after V[s_vid] and before V[e_vid] to Vlist, doing one segment
    //   of a timeline at a time.
    //
    attrib := any attrib of V[s_vid] with attrib.atype = timeline;
    s_tid := attrib.value_tl.tid;
    s_ver := attrib.value_tl.ver;
    if done_set[s_tid] ≠ null andif done_set[s_tid]^.ver > s_ver then
        todo_set += ⟨s_vid, s_tid, s_ver⟩;

        while todo_set not empty do
            doing := any element of todo_set;
            todo_set -= doing;
            done_ver := done_set[doing.tid]^.ver;   //   record when to end this pass
            done_set += ⟨doing.tid, doing.ver⟩;      //   do not repeat this pass
```

```
                while doing.ver < done_ver do
                    Vlist += doing.vid;

                    //  Find where V[doing.vid] leads.
                    //
                    next := ⟨id_null, doing.tid, done_ver⟩;        //  in case end of timeline
                    for each out of V[doing.vid] until E[out].link = event_link do
                        h_vid := E[out].head;
                        for each attrib of V[h_vid] with attrib.atype = timeline do
                            h_tid := attrib.value_tl.tid;
                            h_ver := attrib.value_tl.ver;

                            if doing.tid = h_tid then
                                next := ⟨h_vid, h_tid, h_ver⟩;
                            else
                                if done_set[h_tid] ≠ null andif done_set[h_tid]^.ver > h_ver
                                  andif ( todo_set[h_tid] = null
                                          orelse todo_set[h_tid]^.ver > h_ver ) then
                                   todo_set += ⟨h_vid, h_tid, h_ver⟩;
                                endif;
                            endif;
                        endfor;
                    endfor;
                    doing := next;
                endwhile;
            endwhile;
        endif;

        return srt_set_to_list(Vlist);
    end list_interval;
```

8. BIBLIOGRAPHY

1. Bates, Peter Charles, and Wileden, Jack C. "EDL: A basis for distributed system debugging tools." Proceedings of the 15th Hawaii International Conference on Systems Science (January 1982): 86-93.

2. Chandy, K. M., and Lamport, Leslie. "Distributed Snapshots: Determining Global States of Distributed Systems." ACM Transactions on Computer Systems Vol. 3, no. 1 (February 1985): 63-75.

3. Harter, Paul K.; Heimbigner, Dennis M.; and King, Roger. "IDD: An Interactive Distributed Debugger." The 5th International Conference on Distributed Computing Systems (May 1985): 498-506.

4. Italiano, Giuseppe F. "Amortized efficiency of a path retrieval data structure." Theoretical Computer Science Vol. 48 (1986): 273-281.

5. Italiano, Giuseppe F. "Finding paths and deleting edges in directed acyclic graphs." Information Processing Letters Vol. 28 (30 May 1988): 5-11.

6. Lamport, Leslie. "Time, Clocks, and the Ordering of Events in a Distributed System." Communications of the ACM Vol. 21, no. 7 (July 1978): 558-65.

7. Lamport, Leslie. "'Sometime' is Sometimes 'NOT Never': On the Temporal Logic of Programs." Conference Record of the 7th Annual ACM Symposium on the Principles Of Programming Languages (January 1980): 174-85.

8. LeBlanc, Thomas J., and Mellor-Crummey, John M. "Debugging Programs with Instant Replay." IEEE Transactions on Computers Vol. C-36, no. 4 (April 1987): 471-81.

9. LeBlanc, Thomas J., and Miller, Barton P., ed. "Summary of ACM Workshop on Parallel and Distributed Debugging." Operating Systems Review Vol. 22, no. 4 (October 1988): 7-19.

10. Tarjan, Robert Endre. Data Structures and Network Algorithms. Philadelphia, PA: Society For Industrial And Applied Mathematics, 1983.