Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-90-40

1990-12-05

Determining 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. Four approaches to this problem are presented, each exploiting more knowledge about this work's application domain and hence becoming progressively less memory intensive. The two most promising of these approaches are examined in depth.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Griswold, Victor Jon, "Determining Interior Vertices of Graph Intervals" Report Number: WUCS-90-40 (1990). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/713

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Determining Interior Vertices of Graph Intervals

Victor Jon Griswold

WUCS-90-40
(revision of WUCS-90-9)

December 5, 1990

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, Missouri 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. Four approaches to this problem are presented, each exploiting more knowledge about this work's application domain and hence becoming progressively less memory-intensive. The two most promising of these approaches are examined in depth.

This work has been supported by NSF grant DCI-8600947, Bellcore, BNR, Italtel, NEC, DEC, SynOptics, and NTT. The author may be reached via email address vjg@wucs1.wustl.edu.

TABLE OF CONTENTS

LIST OF FIGURES

Determining Interior Vertices of Graph Intervals

Victor Jon Griswold

1. Introduction

1.1 Background

The project leading to the work presented in this report involves the monitoring of distributed systems by means of observing "events" generated by the systems being monitored. In order to organize and interpret those events, the monitor must be able to determine which events occur "between" two bounding events **A** and **B** in quasi-ordered logical time.* Use of this temporal paradigm allows a directed acyclic graph to be constructed such that its vertices and edges are in one-to-one correspondence with, respectively, events and those temporal orderings which the monitor can explicitly recognize (through the use of various rules). The target of this report, the above "list all events V_i 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.

---

* The monitor interprets the temporal progress of a distributed system by means of quasi-ordered logical time[7], 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.

1.2 Terms

A history graph $H = \langle V, E \rangle$ is a directed acyclic graph. A vertex $v_i \in V$ corresponds to a single event $\mathbf{V}_i$ in our application. A directed edge $e_k = (v_t, v_h) \in E$ corresponds to the temporal relationship "$\mathbf{V}_t$ occurred before $\mathbf{V}_h$". Let $\nu \equiv |V|$, and $\varepsilon \equiv |E|$.

The quasi-ordering between any two vertices in H is defined by the relation $\prec$, called **precedes**. Specifically, $v_i \prec v_j$ if and only if there exists a directed path in H with origin v_i and terminus v_j. We say that v_j **follows** v_i, written $v_j \succ v_i$, if and only if $v_i \prec v_j$. The relations '$\preceq$' and '$\succeq$' are defined according to their classical meanings in terms of '$\prec$', '$\succ$', and '='. Given two vertices a and b, those vertices v_i such that $a \preceq v_i \preceq b$ are said to be between a and b (a and b inclusive).

The graph interval, or just interval, in H from a to b as the set containing the vertices on all directed paths with origin a and terminus b in H. This is written $[a \Rightarrow b]$; a is the start bound and b is the end bound of the interval. Intuitively, $[a \Rightarrow b]$ is all vertices between a and b. If and only if $a \nprec b$, $[a \Rightarrow b] = \varnothing$.

1.3 Problem Definition

The goal of this report is to be able to answer queries about intervals in H as H is constructed *incrementally*. Algorithms developed for this purpose can not depend on additional vertices and edges not being added to H after the first query is posed. Given these requirements, three basic operations must be supported:

ADD_VERTEX. Given a graph $H_{q-1,r} = \langle V_{q-1}, E_r \rangle$ and a vertex v_q, construct $H_{q,r} = \langle V_q, E_r \rangle$ where $V_q = V_{q-1} \cup \{v_q\}$.

ADD_EDGE. Given a graph $H_{q,r-1} = \langle V_q, E_{r-1} \rangle$ and an edge $e_r = (v_t, v_h)$, construct $H_{q,r} = \langle V_q, E_r \rangle$ where $E_r = E_{r-1} \cup \{e_r\}$.

LIST_INTERVAL. Given a graph $H = \langle V, E \rangle$ and two vertices $v_s \in V$ and $v_e \in V$, construct a set $I = [v_s \Rightarrow v_e]$. Define $\nu_I \equiv |I| = |[v_s \Rightarrow v_e]|$.

Perhaps the most common approach to optimizing a set of algorithms is to have the algorithms make use of regularities in their input data. For the monitor application, one might suppose that events generated by the same object could be grouped together in some fashion. This is indeed the case: events can be grouped with respect to both graph structure and sequencing of the above operations without loss of generality.

Consider an object in a distributed system, such as a processor or shared data object, which possesses a sequential event history. Events from that object are probably most frequently ordered with respect to other events from the same object. Also, given the object's sequential event history, a *total* ordering of those events is known. This ordering is valid for both real and logical time and means that events from the same source can be added to H in order. With this knowledge, we can define H in a different, though equivalent, manner, and adjust the definition of ADD_VERTEX to accommodate this:

A history graph $H = \langle G, T \rangle$ is composed of a directed acyclic graph $G = \langle V, E \rangle$ along with a set T of distinguished paths in that graph. A directed path $t \in T$, called a timeline, is an alternating sequence of vertices $v \in V(t)$ and edges $e \in E(t)$. T covers V; that is, $V = \bigcup_{t \in T} V(t)$. Any given edge or vertex occurs at most once as a component on a given path (by definition of path), but might be a component of more than one path. It is useful to identify those edges in E which are not a component of any path in T. These edges, called cross-timeline edges, make up the set $X = E - \bigcup_{t \in T} E(t)$. Let $\tau \equiv |T|$ and $\varepsilon_X \equiv |X|$. The index of a vertex within a path is referred to as its version on that path; the vertex is said to be ordered on that path. A path with origin v_{org} and terminus v_{term} is denoted by $(v_{\text{org}}, v_{\text{term}})$.

ADD_VERTEX. Given $H = \langle G, T \rangle$, a vertex v_q, a set $T_{\text{on}} \subseteq T$, and a non-negative integer τ_{new}, construct $H' = \langle G', T' \rangle$. T' consists of the union of three sets: $T - T_{\text{on}}$, the set of paths derived by appending v_q as a new terminus to each of the paths in T_{on} (along with an edge from each path's previous terminus to v_q), and a set of τ_{new} new paths each of which contains only v_q. $G' = \langle V', E' \rangle$, where $V' = V \cup \{v_q\}$, and $E' = E \cup \{$*the new edges added to the paths in* $T_{\text{on}}\}$.

These definitions of H and ADD_VERTEX are effectively equivalent to the original definitions if one enforces that every added vertex augment a unique timeline (i.e. $T_{\text{on}} = \varnothing$ and $\tau_{\text{new}} = 1$ for every ADD_VERTEX).

It has been found useful, in both a practical sense and an algorithmic one, for H to initially contain one distinguished vertex, v_0, which is the origin of every timeline. Practically, v_0 represents the "start of time" for the monitor. Algorithmically, the use of v_0 helps avoid explicit checks for several boundary conditions in the algorithms to be presented later. The existence of v_0 is not mandatory from a absolute point of view, but, since it does make the algorithms more easily understood, it shall be assumed to exist. Given this use of v_0, the construction of T' in the above ADD_VERTEX definition must be changed so that the τ_{new} new paths initially contain v_0, not v_q.

A second avenue towards optimization is to restrict the domain of operations which may be performed on H. For the monitor application, the domain (pairs of vertices) over which LIST_INTERVAL operations may be requested is known. Additionally, there is a significant amount of knowledge about the domain over which ADD_EDGE operations are performed. With such information, vertex sets B_s and B_e can be identified so that LIST_INTERVAL operations are restricted to intervals $[v_s \Rightarrow v_e]$ where $v_s \in B_s$ and $v_e \in B_e$. Similarly, vertex sets A_t and A_h can be identified so that ADD_EDGE operations are restricted to edges $\langle v_t, v_h \rangle$ such that $v_t \in A_t$ and $v_h \in A_h$. The definitions of the above operations are suitably amended, and one more operation is defined:

Vertex sets B_s, B_e, A_t, and A_h are the enabling sets for their elements to be an interval start or end bound or to be a tail or head in an ADD_EDGE operation, respectively. If a vertex $v_c \in B_s, B_e, A_t$, or A_h, v_c is said to be a candidate for use in the corresponding situation. A statement such as "$v_c \in B_s$" will often be phrased as "v_c is an s candidate".

ADD_VERTEX. The vertex v_q may be added to one or more of B_s, B_e, A_t, or A_h. This is the only time v_q may be added to an enabling set.

ADD_EDGE. It is required that $v_t \in A_t$ and that $v_h \in A_h$.

LIST_INTERVAL. It is required that $v_s \in B_s$ and that $v_e \in B_e$.

DISABLE_CANDIDATE. Given a vertex v_c and one or more of the enabling sets B_s, B_e, A_t, and A_h. Remove v_c from each of those enabling sets.

This set of definitions is still equivalent to the originals if each added vertex is placed into every enabling set and DISABLE_CANDIDATE is never invoked. It should be noted that every newly-added vertex v_q must initially be at least an **h** candidate. This is so that v_q can be the head of an edge from the previous terminus of each timeline(s) on which v_q is ordered (unless, of course, v_q is the origin of each of those timelines, though the use of v_0 removes even that possibility). Also, unless a vertex v_q is known to be the final terminus of a timeline, v_q must be at least a **t** candidate so that it can be the tail of the edge to the timeline's next terminus.

1.4 Two Complexity Issues

Though the speed of responding to LIST_INTERVAL is not unimportant, the monitor application makes the space requirements for that response of paramount importance. A distributed system might generate thousands of events, each corresponding to a vertex in H. Any algorithm requiring just $O(\mathrm{v}^2)$ space is therefore considered of no practical use. Given this, $O(\varepsilon)$ is adopted as the target space complexity.

The analysis of LIST_INTERVAL faces a problem akin to that present when analyzing database query algorithms.[13] Since it is possible for LIST_INTERVAL to return V in its entirety, the time cost for just building I in such a case is $\Omega(\mathrm{v})$ — for the same cost, an algorithm could determine which vertices to put into I by simply comparing every vertex in H to the interval's bounds. Such a complexity measure for LIST_INTERVAL, referred to as the <u>locate-and-copy</u> time, is generally considered too coarse to be useful. Instead, the <u>locate</u> and <u>copy</u> times for LIST_INTERVAL are differentiated in this report. The locate time can be viewed as the time required to distinguish I and the copy time as the time required to output I.*

1.5 History Graph Diagrams

The diagram format for history graphs in this report represents each vertex as a circle with its ADD_VERTEX sequence inside the circle and its candidacies to the side of the circle.** Edges are represented as arrows from tail to head. Vertices within the same timeline are arranged vertically with the timeline's origin towards the top (i.e. **precedes** order "flows down" the timeline path). If a vertex is ordered on more than one timeline, it is highlighted with a double instead

---

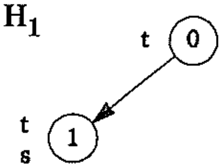
* Ideally, copy time for LIST_INTERVAL would be $O(\mathrm{v}_I)$. Unfortunately, this is not always the case because of scanning complications such as avoiding putting a vertex into I multiple times if that vertex is on more than one path between the interval bounds.

** It has been found that displaying vertices' candidacies at the sides of the vertices is easier to read than listing the enabling sets alongside the graph.

H_0

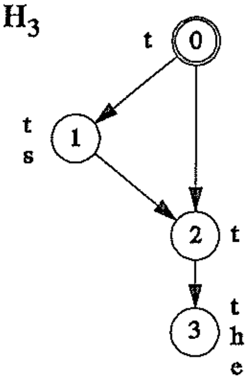
— Initial Conditions —

H_1



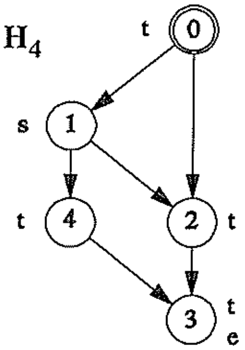
add v_1 on t_1 with {**t**,**h**,**s**}
add e_0= (v_0, v_1)
dis_cand v_1, {**h**}

H_3



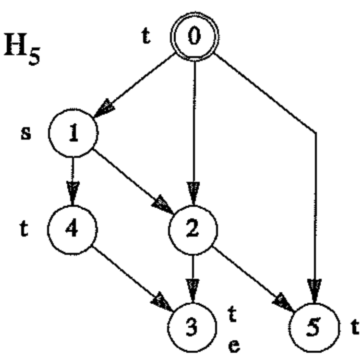
add v_2 on t_2 with {**t**,**h**}
add e_1= (v_0, v_2)
add e_2= (v_1, v_2)
dis_cand v_2, {**h**}
add v_3 on t_2 with {**t**,**h**,**e**}
add e_3= (v_2, v_3)

H_4



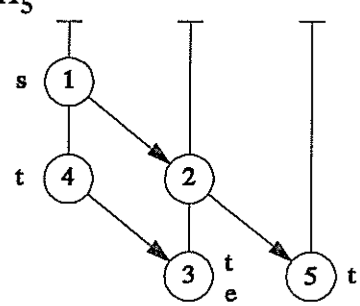
add v_4 on t_1 with {**t**,**h**}
add e_4= (v_1, v_4)
dis_cand v_1, {**t**}
dis_cand v_4, {**h**}
add e_5= (v_4, v_3)
dis_cand v_3, {**h**}

H_5



add v_5 on t_3 with {**t**,**h**}
add e_6= (v_0, v_5)
add e_7= (v_2, v_5)
dis_cand v_2, {**t**}
dis_cand v_5, {**h**}

H_5



As to the left, but:
— v_0 implied
— order within a timeline implied

Figure 1. History Graph Structure and Operations

of a single circle. A vertex v is said to be *ordered with* a timeline t if there exists a path in H with terminus v and origin any $v' \in V(t)$. Similarly, two vertices v and w are *ordered with respect to each other* if either $v \prec w$ or $w \prec v$.

Figure 1, which contains six examples of history graph diagrams, shows the construction of a history graph H from five vertices besides v_0, three timelines, and the potential for one interval query. In the following discussion, the operations and queries performed on H are referred to as being supplied by "the user," though in reality this "user" would be a program.

As shown in Figure 1, H_0 contains only v_0. Vertex v_0 is a **t** candidate, so it may be the tail of subsequent edges. It is not, however, an **s** candidate, so no interval query may designate v_0 as its start bound. Vertex v_1 is then added to H_0. Vertex v_1 is ordered on timeline t_1 and is version 1 of that timeline (v_0 is version 0 of t_1 and all other timelines). Initially, v_1 is a **t**, **h**, and **s** candidate, so it may be the tail or head of subsequent edges and the user may pose an interval query with v_1 as the start bound (but not an interval query with v_1 as the end bound). Next, edge e_0 is added to H from v_0 to v_1, as shown by the arrow. The user, in this example, determines that e_0 can be the only edge with head v_1, and therefore removes v_1 from A_h. If the user can not discern this property and does not disable v_1's **h** candidacy, proper query results are not affected but certain data structure optimizations can not be made. This completes 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 vertices and three edges. Of note is that v_3 is an **e** candidate; after v_3 and all its incident edges are added to H (i.e. after H_3 is completed), the user may pose an interval query for $[v_1 \Rightarrow v_3]$ (and would receive $\{v_1, v_2, v_3\}$ in response). Additionally, after the addition of v_2, v_0 is highlighted with a double circle since it is a component of both t_1 and t_2.

H_4 consists of H_3 with one more vertex and two more edges. Moreover, the user determines that no further edges may have tail v_1 and removes v_1 from A_t, providing an avenue for further data structure optimizations. H_5 adds the final vertex and edges to this example. In H_5, neither v_1 nor v_2 may be incident to any new edges to be added to H. For this graph, the response for an interval query of $[v_1 \Rightarrow v_3]$ is $\{v_1, v_2, v_3, v_4\}$. The response would not include v_5 because, though v_5 **follows** v_1, its ordering 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 used 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, those edges which show the progression of order along a timeline are represented simply by segments instead of by arrows, since arrows within a timeline would always point down in a graph representation.

2. Transitive Closure Method

2.1 Approach

A rather robust means of responding to LIST_INTERVAL queries is by maintaining complete transitive closure information about the history graph, making no assumptions about its structure other than that it is directed and acyclic. When a query is posed for $[a \Rightarrow b]$, the answer is simply all vertices $v_i \ni (a \preceq v_i \preceq b)$.

To the author's knowledge, the fastest published algorithm for incrementally maintaining the transitive closure of a directed acyclic graph was developed by Giuseppe F. Italiano.[5] This algorithm adds edges to a graph in $O(\mathrm{v})$ amortized time per edge and reports the ordering between two vertices in $O(1)$ (constant) time. Unfortunately, Italiano's algorithm requires prior knowledge of the maximum number of vertices in the graph (due to storage allocation considerations*) and has a space complexity of $\Theta(\mathrm{v}^2)$.

2.2 Algorithm

As stated above, Italiano's algorithm makes no assumptions about the structure of the history graph. For the monitor application, this general-purpose nature makes the algorithm's space complexity prohibitive and ADD_EDGE time undesirable. Nonetheless, the use of Italiano's algorithm remains of interest as a basis for comparison.

We take this opportunity to introduce the pseudocode representation employed for the expression of algorithms in this report. The pseudocode employs an Ada-like syntax, explained in detail in Appendix 7.1. The four operations defined in Section 1 are declared in Figure 2, along with the data types used in the declarations and some data structures which might support those 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 an array with which to make $O(1)$ path-existence checks and a set of trees to record the actual paths.

---

* It is possible to dynamically increase the maximum number of vertices, but such an adjustment would require a significant reorganization of the algorithm's index data structure (this need not increase the $O(\mathrm{v})$ running time, just the constant factor). Such restructuring would cause a bursty and unpredictable (and thus unacceptable) performance impact on the monitor application.

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

types
    natural =      range [0..] of integer;
    vertex_id =    range [id_null..ν_limit] of integer;
    edge_id =      range [id_null..ε_limit] of integer;
    timeline_id =  range [id_null..] of integer;
    version_index = natural;

    ordering = record                         //  version (order) of a vertex on a timeline
        tid : timeline_id;
        ver : version_index;
    end ordering;

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

    vertex = record
        //  whatever an implementation needs to keep track of
    end vertex;

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

globals
    V : array [0..ν_limit] of vertex;         //  any O(1) access time structure
    ν : natural := 0;                         //  current number of vertices
    E : array [0..ε_limit] of edge;           //  any O(1) access time structure
    ε : natural := 0;                         //  current number of edges

    A_t : set of vertex_id;                   //  vertices which may later be an edge tail
    A_h : set of vertex_id;                   //  vertices which may later be an edge head
    B_s : set of vertex_id;                   //  vertices which may be a query start bound
    B_e : set of vertex_id;                   //  vertices which may be a query end bound

//  Return the vertex_id corresponding to ⟨timeline_id, version_index⟩.
//
function get_vertex (ord : ^ordering) : vertex_id;

procedure add_vertex (new_V : vertex;
                      T_on : set of timeline_id; candidate_for : set of candidacy;
                      out v_q : vertex_id);
procedure add_edge (v_t, v_h : vertex_id; out e_r : edge_id);
function   list_interval (v_s, v_e : vertex_id) : set of vertex_id;
procedure disable_candidate (v_c : vertex_id; not_candidate_for : set of candidacy);
```

Figure 2. Declaration of Required Operations

Unless reorganization of the path existence lookup table is permitted, the maximum number of vertices is fixed for Italiano's algorithm. The **add_vertex** procedure is thus a no-op with respect to the Italiano data structures. Furthermore, since Italiano's algorithm makes no optimizations based on knowledge of future ADD_EDGE or LIST_INTERVAL operations, the **disable_candidate** procedure is also effectively a no-op. The procedure **add_edge** is not a no-op, though is trivial:

procedure add_edge (v_t, v_h : vertex_id; **out** e_r : edge_id);
begin
  Ital_add_edge(v_t, v_h);
  e_r := ε;
  return;
end add_edge;

Of particular use for LIST_INTERVAL is the $v \times v$ lookup table, **index**, maintained by Italiano's algorithm in order to directly check for the existence of a path from any vertex v_i to any

---

types
  vertex_id = **range** [0..v_limit] **of** integer; // no need for id_null

  Ital_node = **record**
    key : vertex_id;
    parent : ^Ital_node;
    child : ^Ital_node;
    sibling : ^Ital_node;
  end Ital_node;

globals
  // index[v_i, v_j] $\neq$ **null** $\rightarrow$ a path exists from v_i to v_j
  //
  index : **array** [vertex_id, vertex_id] **of** ^Ital_node := **null**;
  desc : **array** [vertex_id] **of** ^Ital_node;

procedure Ital_add_edge (v_t, v_h : vertex_id);
function Ital_check_path (v_{org}, v_{term} : vertex_id) : **Boolean**;
function Ital_get_path (v_{org}, v_{term} : vertex_id) : **list of** vertex_id;

Figure 3. Operations Provided by Italiano's Algorithm

other vertex v_j. The algorithm's ability to list a single path from v_i to v_j is of little use for LIST_INTERVAL's purpose of listing all such paths.* Hence, the query is resolved by using **index** to find the intersection of those vertices after the interval's start bound with those before its end bound. The following **list_interval** implementation, though quite straightforward, still takes $O(\mathrm{v})$ locate time. This is similar to the $O(\mathrm{v_I})$ locate-and-copy time limit for the query but is perhaps much larger. Copy time is $O(\mathrm{v_I})$.

```
function list_interval (vs, ve : vertex_id) : set of vertex_id;
    I : set of vertex_id := ∅;
    v :  vertex_id;
begin
    if index[vs, ve] ≠ null then
        I ∪= {vs, ve};
        for v in [0..v-1] do
            if index[vs, v] ≠ null and index[v, ve] ≠ null then
                I ∪= {v};
            endif;
        endfor;
    endif;

    return I
end list_interval;
```

* 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 $O(\mathrm{v})$ (instead of $O(\mathrm{v} \log \mathrm{v})$) ADD_EDGE time was the removal of all such "redundant" multiple-path information from the algorithm's data structures.

3. Search Tree Method

3.1 Approach

This second method of responding to LIST_INTERVAL 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 while requiring $O(\tau \varepsilon_X + v)$ space.* Such space costs at first appear worse than those of Italiano's algorithm because ε, for a general graph, is $O(v^2)$. The monitor application's removal of edges which are redundant through transitivity, however, makes ε closer to $O(\tau v)$. For graphs with a large number of vertices relative to the number of timelines, the search tree method (STM) may thus require considerably less time and space than the transitive closure method using Italiano's algorithm.

The core of the search tree method is its cross-timeline path data structures. For each timeline t_w in H, a sorted set of vertices** is maintained for the path t_w itself. Along with that sorted set are sorted sets for each timeline t_x with which some vertex on t_w is ordered. These sorted sets contain the origin and terminus of all paths from t_x to t_w which are not redundant through transitivity. For graphs in which vertices (and thus edges) are added in topological order, update of the cross-timeline structures when a new edge e_r is added to X can be performed with the following simplified procedure:

- Given $e_r = \langle v_t, v_h \rangle$. Determine timelines t_x and t_w such that $v_t \in V(t_x)$ and $v_h \in V(t_w)$.
- Through t_x's cross-timeline path records, find the origin of all cross-timeline paths to t_x with terminus v_t. This <u>includes</u> those paths not explicitly recorded as terminating with v_t but which are instead recorded as terminating with a vertex on t_x which has an earlier version than v_t (recording an explicit path to v_t would thus have been redundant). Since e_r has been added to H, each of these origins is *also* the origin of a path with terminus v_h.

---

* For brevity in the remainder of this report, all time and space complexity measurements shall be assumed to be asymptotic complexities ("O") unless otherwise stated.

** A <u>sorted set</u> is a set totally ordered by a relation over a <u>key</u> attribute of each of the set's elements.[12] A typical operation on a sorted set is, naturally, searching for an element with a particular key value. The most common implementations of sorted sets are search trees and hash tables. For the STM path records, sorted sets are implemented as threaded AVL trees[4][11] ordered by version on the timeline.

- For each path (v_{origin}, v_t) determined above, record the path (v_{origin}, v_h) if it is not already implied through transitivity. This is the case whenever v_{origin} is also the origin of a path to some vertex on t_w which has an earlier version than v_h.

The pairwise-timeline sorted sets are the reason for the τ^2 factors in the search tree method's complexity measures. If, for a particular H, ordering between timelines has a strong locality (for instance, each processor represented as a timeline might only communicate 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 vertices (not counting v_0); Figure 4b-d show the cross-timeline paths recorded for that graph, one sub-figure for the path information associated with each of the three timelines. In each of Figure 4b-d, the path-origin timelines of the underlying graph are de-emphasized by showing them as dotted lines while the terminus timeline and the cross-timeline paths themselves are shown as bold lines. Given the cross-timeline path data structures in this example, checking for the existence of a path from v_7 to v_{12} proceeds as follows:

1) Inspect those paths which originate from v_7's timeline (t_3) and terminate at v_{12}'s timeline (t_1). Of these, find the path the terminus of which has the highest version less than or equal to that of v_{12}. This terminus would be v_{10}.
2) Determine if the origin of that path has a version greater than or equal to that of v_7. In this case, the origin is v_8, which does **follow** v_7 on t_3 . It has thus been demonstrated that a path from v_7 to v_{12} exists by recognizing three of its sections: the path originates at v_7 on t_3, proceeds to v_8 along some number of edges on t_3, proceeds to v_{10} on t_1 along some number of edges across some number of intermediate timelines, and finally terminates at v_{12} along some number of edges on t_1.

3.2 Algorithm

Before examining the algorithms in this subsection, some elaboration is necessary. The existence of the sorted set operations described in Appendix 7.1 is assumed. Their implementation requires time per operation on the order of the log of the number of items in the set.[12] In addition to the data structures of Figure 2, the search tree method makes use of those presented

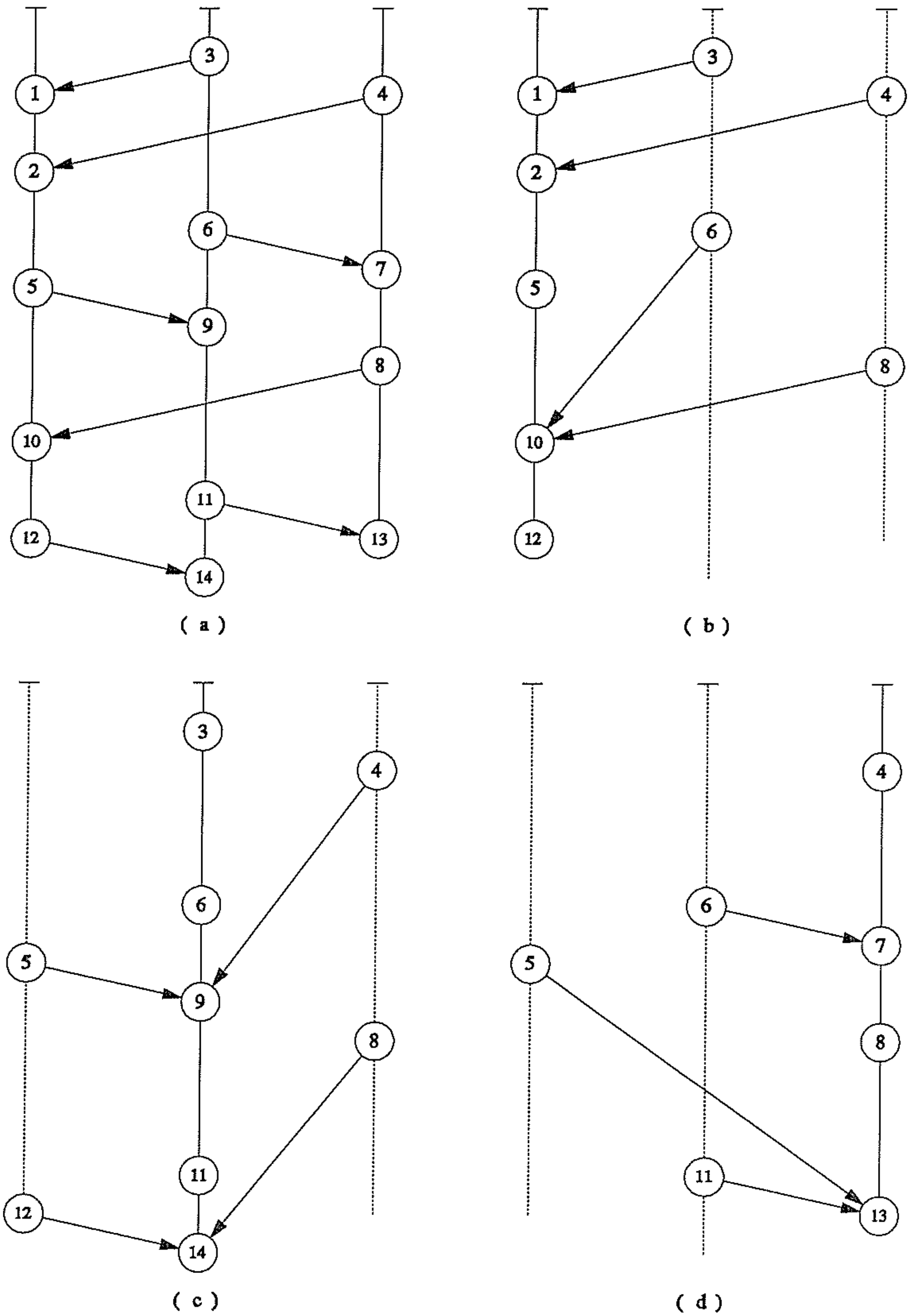


Figure 4. Cross-Timeline Path Information for Search Tree Method

```
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 vertex
    //  identifier of the terminus since the origin's version would simply be terminus
    //  version - 1.
    //
    x_tl_path = record
        case (cross_timeline, in_timeline) of
            cross_timeline : (org : version_index;);
            in_timeline :    (vid : vertex_id;);
        endcase;
        term : version_index;
    end x_tl_path;

    origin_paths = record
        org_tid :  timeline_id;                    //  id of tl on which origins are ordered
        path :  srt_set of x_tl_path key term, org;  //  we need to search by either field
    end origin_paths;

    timeline = record
        id :       timeline_id;
        self :     ^origin_paths;             //  convenience: always points to xtpaths[id]
        xtpaths : srt_set of origin_paths key org_tid;
    end timeline;

globals
    T : srt_set of timeline key id;

//  Return the version of v on t.
//
function version(v : vertex_id; t : timeline_id) : version_index;
```

Figure 5. Search Tree Method Data Structures

in Figure 5. Keep in mind that the cross-timeline data structures keep track not of individual edges between timelines, but of paths between timelines. For analysis purposes, it is considered trivial to determine each timeline on which a vertex is ordered and the vertex's version on that timeline.* Similarly, given a timeline and version, it is assumed that one can quickly find the corresponding vertex. Implicit "conversions" between vertices and vertex_ids are often made in

* In an actual implementation of these algorithms, the **add_vertex** T_{on} parameter is stored with the vertex in V along with the vertex's version on each timeline.

this subsection. It is proper to be able to search the **path** field of **origin_paths** by either **term** or by **org** because it is true that for all **x_tl_paths** in a particular **path** sorted set, x.**term** > y.**term** implies x.**org** > y.**org** (i.e. when **path** is sorted by **term**, it is also sorted by **org**). A search by **term** is denoted with **path**[*key*] and a search by **org** with **path.org**[*key*].

Since the search tree method makes no optimizations based on knowledge of future ADD_EDGE or LIST_INTERVAL operations, its **disable_candidate** procedure is effectively a no-op. The pseudocode presented in this subsection is a high-level description of the algorithms; a more detailed description is found in Appendix 7.3.

One optimization in the algorithms presented here should be noted before confusion arises. A procedure which responds to a LIST_INTERVAL query must report the identifiers of the vertices in the requested interval. The search tree method's path recording mechanism, however, generally tracks only the version of a vertex on a timeline (since vertices are ordered on a timeline by version, not by the vertex identifier). Either a separate data structure to record the vertex identifiers must be maintained or the identifiers must be maintained along with the paths. The optimization makes use of the property that, when a path is recorded between two vertices on the same timeline, the version of the origin is always 1 less than that of the terminus. The space ordinarily used to hold the origin's version is used, instead, to hold the terminus' vertex identifier.

The STM **add_vertex** procedure is based on the second definition of ADD_VERTEX, in which the edges from any previous terminus of v_q's timelines are added during ADD_VERTEX instead of later. Aside from the **add_edge** calls, the operation of **add_vertex** is self-explanatory. It should be realized that storage of **new_V** into V is of use only for the application invoking **add_vertex**; the STM routines make no direct use of V. Pseudocode for **add_vertex** is:

```
procedure add_vertex (new_V : vertex; T_on : set of timeline_id;
                      out v_q : vertex_id);
    t : timeline_id;
    e_r : edge_id;                        //  not used, in this case
begin
    ν += 1;
    v_q := ν;
    V[v_q] := new_V;
    for each t ∈ T_on do
        add_edge(T[t].self→last()→vid, v_q, e_r);
    endfor;
    return;
end add_vertex;
```

The simplification made in this section's introduction, that vertices (and thus edges) are added to H in topological order, can not be made in general. This complicates the **add_edge** procedure because an additional level of transitivity is involved. After adding an edge from v_t to v_h, v_h must **follow** all vertices $v_t' \prec v_t$. For the general case, all vertices $v_h' \succ v_h$ must *also* **follow** all vertices $v_t' \prec v_t$. Given all vertices $v_t' \prec v_t$ and all vertices $v_h' \succ v_h$, the path records must be updated so that $v_t' \prec v_t \prec v_h \prec v_h'$. Further complications result from the possibility that v_h is ordered on multiple timelines.

An important subroutine of **add_edge** is **update_tl_xt**, shown in Figure 6. This subroutine accepts a vertex v_{term} on a timeline t and a set of vertices (identified as ⟨timeline_id, version_index⟩'s) which are origins of paths to v_{term}. **Update_tl_xt** then updates t's cross-timeline records so that these paths are recorded. The creation of new cross-timeline structures (if t had no existing paths from a particular origin's timeline) is also handled by **update_tl_xt**, as is the

---

```
procedure update_tl_xt(t : ^timeline; v_term : vertex_id;
                        origins : ordering_set);
    xt :     ^origin_paths;
    origin : ^ordering;

begin
    for origin ∈ origins do
        xt := t→xtpaths[origin→tid];

        if no existing paths to t originate from that timeline then
            add a new cross-timeline path set to t→xtpaths;
            add the initial v0 to that set;
        endif;

        if origin→tid ≠ t→id then
            if a path from origin is not redundant then
                add the (origin, v_term) path to xt;
                remove existing paths made redundant by this new path;
            endif;
        else
            add (origin, v_term) to t→self, if not redundant;
        endif;
    endfor;
    return;
end update_tl_xt;
```

Figure 6. Search Tree Method **Update_tl_xt** Procedure

```
procedure add_edge (v_t, v_h : vertex_id; out e_r : edge_id);
    t : ^timeline;
    v_org, v_term : vertex_id;
    xt : ^origin_paths;
    origins : ordering_set := ∅;
begin
    ε += 1;
    e_r := ε;
    E[e_r] := ⟨v_t, v_h⟩;

    //  Find all vertices which are now ≺ v_h
    //
    t := any timeline such that v_t ∈ V(t);
    for xt ∈ t→xtpaths, xt ≠ t→self do      //  t itself is handled below
        find the latest v_org ≺ v_t on xt's origin timeline;
        if v_org ≠ v_0 then                  //  everything follows v_0; ignore it
            origins += ⟨xt→org_tid, version(v_org, xt→org_tid)⟩;
        endif;
    endfor;

    for each t such that v_t ∈ V(t) do
        origins += ⟨t, version(v_t, t)⟩;
    endfor;

    //  Update v_h to follow origins
    //
    for each t such that v_h ∈ V(t) do
        update_tl_xt(t, v_h, origins);
    endfor;

    //  Update all vertices which follow v_h to follow origins
    //
    for t ∈ T do
        v_term := the earliest vertex on t which follows v_h;
        if v_term ≠ id_null then
            update_tl_xt(t, v_term, origins);
        endif;
    endfor;

    return;
end add_edge;
```

Figure 7. Search Tree Method **Add_edge** Procedure

case when the new paths make existing paths redundant. This occurs in the following situation: Consider $v_{term}' \succ v_{term}$ on t. **Update_tl_xt** is given v_{org} on t_{org} so that it can record (v_{org}, v_{term}). Additionally, the path (v_{org}', v_{term}') was previously recorded, v_{org}' also on t_{org}. If $v_{org}' \preceq v_{org}$, explicitly recording (v_{org}', v_{term}') is no longer necessary because it can be determined through the transitive relationship $v_{org}' \preceq v_{org} \prec v_{term} \prec v_{term}'$. Figure 7 lists the search tree method's **add_edge** procedure.

Vertices in an interval $[v_s \Rightarrow v_e]$ are found through a three-step process:

- Determine the set of all timelines with which v_e is ordered. Call this set T_I.
- For each $t \in T_I$, determine the earliest vertex on t which **follows** v_s and the latest vertex on t which **precedes** v_e.
- For each $t \in T_I$, add to I all vertices after v_s and before v_e. This is referred to as the span of vertices of I on t. Do not add vertices which are on more than one timeline multiple times.

Pseudocode for **list_interval** is shown in Figure 8.

3.3 Analysis

The $O(\tau^2 \log \varepsilon_X + \tau \log v)$ time for **add_edge** is calculated by direct examination of the procedure's pseudocode. Begin with inspection of **update_tl_xt**. The top level of this subroutine is a loop for each **origin** which v_{term} should **follow**; there could be τ **origins**. Within the loop, v_{term}'s timeline is searched for the existing cross-timeline paths originating from **origin**'s timeline. This search is $O(\log \tau)$. If a structure containing these paths is not present, it is created with an $O(\log \tau)$ insert. If the new (**origin**, v_{term}) path is not redundant, it is recorded with either two $O(\log \varepsilon_X)$ or one $O(\log v)$ insertion(s) (depending upon whether or not the path originates on v_{term}'s own timeline, t). Whenever a path does not originate on t, an out-of-order situation must be checked. The pseudocode above remedies this out-of-order situation with a slow $O(\varepsilon_X \log \varepsilon_X)$ 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. If self-adjusting splay trees[12] are used instead of AVL trees for the path records, two splay tree splits and a splay 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 **origin** can be on v_{term}'s own timeline). One can, though, compare τ and ε_X in order

```
function list_interval (v_s, v_e : vertex_id) : set of vertex_id;
    I : srt_set of vertex_id := ∅;              //  avoid duplicates
    I_terms : list of ordering := [ ];          //  termini of all spans of vertices making up I
    I_term :  ^ordering;
    v_I_org, v_I_term, v_i : vertex_id;
    t, t_s : ^timeline;
    xt :   ^origin_paths;
begin
    //      Find the latest vertex before v_e for each timeline with which v_e is
    //  ordered.
    //
    t := any timeline such that v_e ∈ V(t);
    for xt ∈ t→xtpaths do
        if xt ≠ t→self then
            find the latest v_I_term ≺ v_e on xt's origin timeline;
        else                                    //  this will lead to putting v_e in I
            v_I_term is v_e itself;
        endif;
        if v_I_term ≠ v_0 then                  //  again, ignore v_0
            I_terms &= ⟨xt→org_tid, version(v_I_term, xt→org_tid)⟩;
        endif;
    endfor;

    //      Add all vertices after v_s and before v_e to I, scanning one timeline at a time
    //  between the first vertex after v_s and the latest vertex before v_e (stored in I_terms).
    //
    t_s := any timeline such that v_s ∈ V(t);
    for I_term ∈ I_terms do
        t := T[I_term→tid];
        v_I_term := get_vertex(I_term);
        xt := t→xtpaths[t_s→id];                //  we want paths from t_s to t
        if xt ≠ null then
            v_I_org := the earliest vertex ≥ v_s on t;
            if v_I_org ≠ id_null andif v_I_org ≼ v_I_term then
                I += all vertices v_i on t ∋ (v_I_org ≼ v_i ≼ v_I_term);
            endif;
        endif;
    endfor;

    return make_set(I);                         //  convert from srt_set to set
end list_interval;
```

Figure 8. Search Tree Method **List_interval** Procedure

to achieve a less verbose measure. A timeline has cross-timeline structures for itself and for all other timelines with which its vertices are ordered; its vertices 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 $O(\tau\log\varepsilon_X + \log\nu)$.

The pseudocode for **add_edge** consists of three primary phases: find the "new" vertices before v_h (i.e. v_t and all vertices which come before v_t), update v_h's cross-timeline paths, and update the cross-timeline paths of all vertices which **follow** v_h. Finding the vertices before v_t requires an $O(\log\tau)$ search to find a timeline t on which v_t is ordered and, for each of t's τ potential cross-timeline structures, an $O(\log\varepsilon_X)$ search on **xt** and possible $O(\log\tau)$ insert into **origins**.* 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 on which v_h is ordered, finding t with an $O(\log\tau)$ search and applying **update_tl_xt** to it. Given the above analysis for **update_tl_xt**, the time cost for this phase is $O(\tau(\tau\log\varepsilon_X + \log\nu))$.

To complete **add_edge**, the cross-timeline paths of all vertices which **follow** v_h must be updated. For each timeline t in the graph, **add_edge** must determine if any vertex on t is ordered with some timeline on which v_h is ordered (i.e. determine if a set of cross-timeline paths to t originate from some timeline on which v_h is ordered; $O(\log\tau)$). If so, **add_edge** finds the first vertex on t **following** v_h ($O(\log\varepsilon_X)$) and applies **update_tl_xt** when appropriate. Completion of **add_edge** thus requires $O(\tau^2\log\varepsilon_X)$ time, similar to updating v_h's cross-timeline structures. When combined with the analyses of the other two phases within **add_edge**, this result implies that **add_edge** as a whole is of time cost $O(\tau^2\log\varepsilon_X + \tau\log\nu)$.

As for **add_edge**, **list_interval**'s time complexity is calculated by examination of the pseudocode. The algorithm begins by finding a timeline t on which v_e is ordered (requires one $O(\log\tau)$ search), then finding the latest vertex v_{l_term} before v_e on each timeline containing the origin of a path to v_e. There could be τ timelines, and the v_{l_term} search requires an $O(\log\varepsilon_X)$

---

* It is possible to replace the $O(\log\tau)$ **origins** **srt_set** insert with an $O(1)$ **list** insert by simultaneously scanning the origin timelines from **xtpaths** and the timelines on which v_t is ordered. Since this change would not affect the overall time cost of **add_edge** and would make the algorithm more difficult to read, it was not done here.

lookup and an $O(1)$ append. Time for this phase of the algorithm is therefore $O(\tau \log \varepsilon_X)$, which classifies as part of the "locate" time for **list_interval**.

I, the set of vertices to be returned by **list_interval**, is built by scanning each timeline t containing the origin of a path terminating at v_e. Given such a t and a timeline t_s on which v_s is ordered, **list_interval** begins by locating t and its cross-timeline paths originating from t_s (both searches are $O(\log \tau)$). If any such paths exist, **list_interval** finds the first vertex v_{I_org} on t **following** v_s ($O(\log \varepsilon_X)$) and finds t's path itself ($O(\log \tau)$). Finally, the span of vertices on t's own path between v_{I_org} and v_{I_term} is traversed, adding each vertex to I ($O(\nu_I)$; see below). The list-building phase thus requires $O(\tau \log \varepsilon_X)$ additional locate time and $O(\tau \nu_I)$ copy time. Combined with the first phase of **list_interval**, this results in an $O(\tau \log \varepsilon_X)$ locate time and an $O(\tau \nu_I)$ copy time for **list_interval** as a whole.

The **list_interval** copy time cost is quite pessimistic; $\tau \nu_I$ is an accurate measure only if the number of instances when a vertex is on more than one timeline is $O(\tau)$. In most "realistic" systems, a vertex on multiple timeline signifies a rendezvous between two processes, $O(1)$, not between some $O(\tau)$ group of processes. For this common case, **list_interval** copy time is simply $O(\nu_I)$.

One may notice that an $O(1)$ time cost is attributed to adding each vertex into I, even though I is defined as a **srt_set** which should require $O(\log \nu_I)$ for adding each vertex. This is because the sole purpose of making I a **srt_set** in the algorithm as presented above is to avoid duplicate entries for a vertex. This can just as easily be done with a vertex-flagging strategy, followed at the end of **list_interval** with a scan through I to reset the flags. The problem with this has to do with any potential <u>distributed</u> implementations of the search tree method algorithms. Using a flagging strategy prohibits concurrent access to a vertex by more than one **list_interval** query at a time, while adding the vertices to a **srt_set** presents no such data structure locking problem. Since the current implementation is non-distributed, it uses flagging and has an $O(1)$ time. This issue, however, should be noted for future implementations.

The search tree method's space requirements (in terms of path records maintained in the cross-timeline structures) are measured by examining the data structures themselves instead of the algorithms which operate on them. Two approaches to deriving this space requirement are presented: one employs commutativity of sequences of ADD_VERTEX and ADD_EDGE operations, the other directly counts cross-timeline paths. For both approaches, it is a given that each timeline maintains knowledge of itself; space requirements can not, therefore, be less than $O(\nu)$.

For the first approach, recollect what happens when an edge is added. The tail of the edge, v_t, **follows** vertices on at most τ timelines, and, after the edge is added, the head v_h must also **follow** those vertices, **origins**. A potential of τ paths must be recorded for each new edge. The problem is that not only must v_h be recorded as **following origins**: all vertices **following** v_h must **follow origins**, as well. Since there may be vertices on τ timelines **following** v_h, this line of reasoning implies that τ^2 potential path entries might be 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 vertex additions —building the same history graph— so that there exist no vertices 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 vertices (and thus edges) are added to the graph in topological order, no vertices yet exist which **follow** the head of each new edge and the space required per new edge is at most τ. This yields a modest $O(\tau\varepsilon_X + \nu)$ space complexity. Since an arbitrarily-created history graph and its corresponding topologically-created history graph are the same graph represented by the same structures, they require the same space to store.

The 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 vertices, and each one may be ordered on at most τ timelines. This argument again yields an $O(\tau\varepsilon_X + \nu)$ space complexity.

The above space complexity is a tight bound. Though not all graphs reach it, the simple graph shown in Figure 9 does exhibit this worst-case space requirement.

3.4 Comparison With Transitive Closure Method

Comparison between the search tree and transitive closure methods is difficult because the search tree method uses the monitor application's underlying timeline structure. It is not realistic to compare the two methods according to the degenerate graph case in which each vertex augments a unique timeline (i.e. in which $\tau = \nu$). Therefore, a somewhat less unrealistic approach is taken. For the monitor application, the number of timelines is usually very small compared to the number of vertices and is often fixed. Hence, this discussion will consider τ a constant factor. Additionally, no distinction will be made between ε and ε_X.

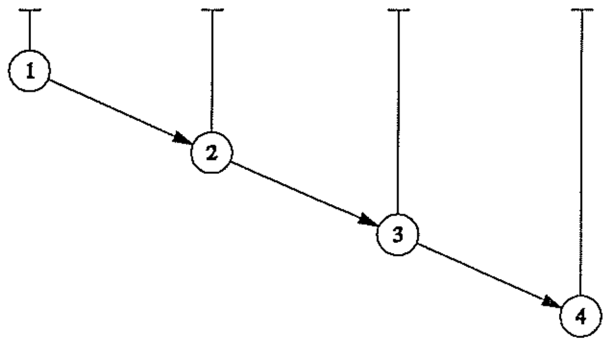


Figure 9. Worst-Case Space for Search Tree Method

Time for ADD_EDGE in the transitive closure method is $O(\mathrm{v})$ (amortized). For the search tree method, it is $O(\log\varepsilon)$. The search tree method time is clearly superior. Similarly, LIST_INTERVAL locate time in the transitive closure method is $O(\mathrm{v})$, verses $O(\log\varepsilon)$ for the search tree method. Copy time for both is $O(\mathrm{v}_I)$.

The search tree method shows a distinct space improvement over the transitive closure method for graphs which are not strongly connected. The transitive closure method takes $\Theta(\mathrm{v}^2)$ space, while the search tree method takes $O(\varepsilon)$.

Each of these comparisons demonstrate that, for graphs with a relatively small number of timelines relative to vertices, the search tree method should be preferred. This is especially true when a graph has substantial locality of connectivity between timelines.

4. Wavefront Method

4.1 Approach

The wavefront method (WVM), so named for the manner in which the LIST_INTERVAL query is resolved, uses information about future vertex operations to decrease both time and space costs. While the search tree method maintains information about every path terminating with a cross-timeline edge, the wavefront method maintains path information only when the path's terminus is an end-bound candidate or tail candidate. If the user is knowledgeable about which vertices can still be incident with new edges, this optimization saves considerable space over the search tree method. Its cost is the loss of rapidly available complete transitive closure information: it is no longer possible to determine the ordering of two arbitrary vertices.*

An example of this optimization is illustrated in Figure 10. Figure 10a presents a simple history graph. Figure 10b shows the search tree method's cross-timeline paths maintained for the second timeline of this graph, and Figure 10c shows the cross-timeline paths maintained by the wavefront method for the same timeline. The reduction of the cross-timeline paths of vertices v_4, v_5, and v_6 into that of v_7 demonstrates a space savings over the search tree method, while the path reduction from v_8 into v_9 merely moves data from one vertex to another (and loses information content while doing so). Notice that records of the paths from v_4 to v_5, v_5 to v_6, and v_7 to v_8 are <u>also</u> reduced from the wavefront method's cross-timeline records (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 vertex on each timeline which **follows** an interval's start bound. In order to resolve a LIST_INTERVAL query, a depth-first search originating at the start bound is used to determine the vertices in the interval. This search terminates at the last vertex on each timeline which **precedes** the interval's end bound (knowledge of which <u>is</u> maintained). The search is pruned before leading to any timelines which are unordered with respect to the end bound.

---

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

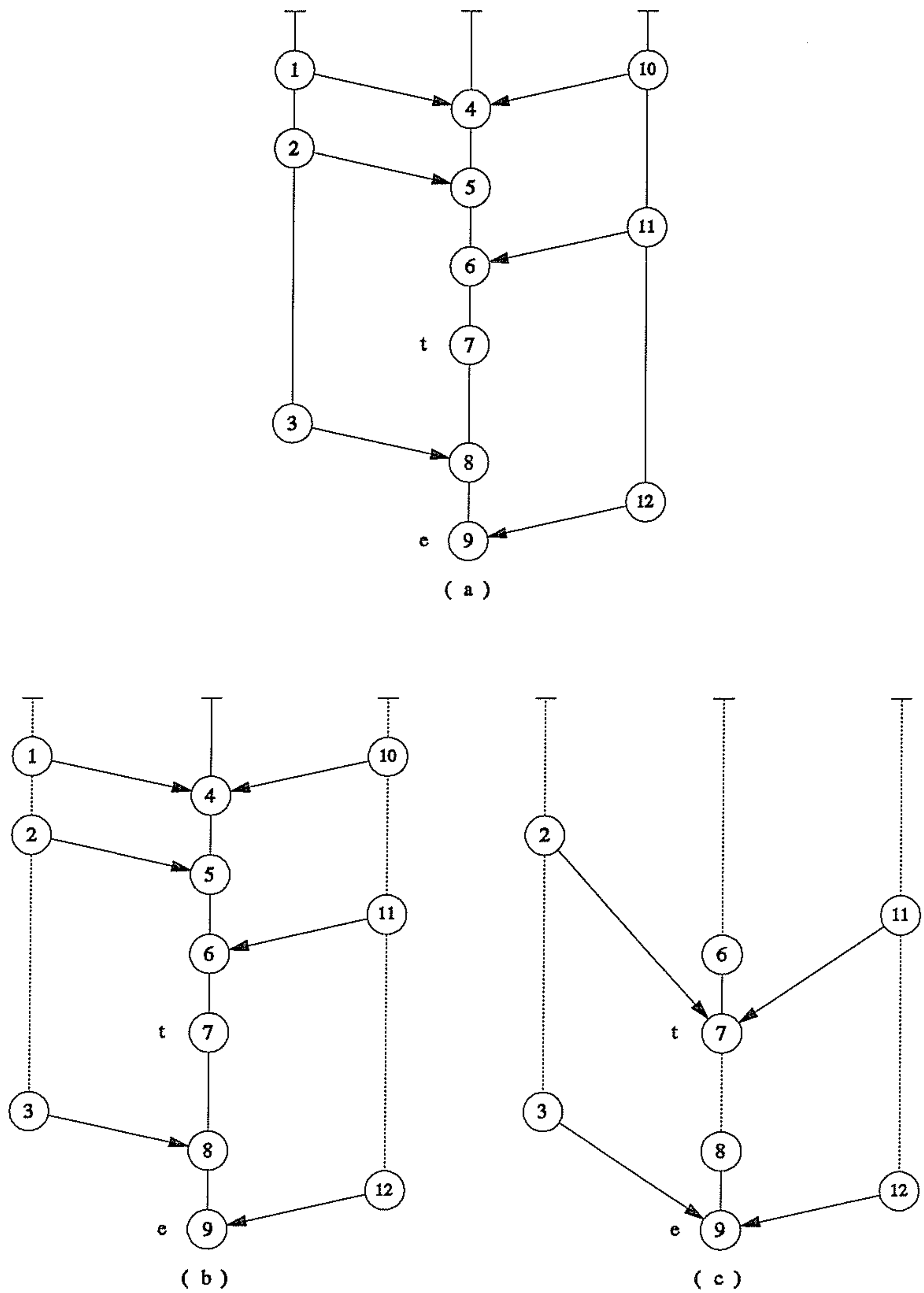


Figure 10. Cross-Timeline Path Information for Wavefront Method

4.2 Algorithm

Slight modifications to the basic Figure 2 data structures are necessary for implementation of the wavefront method. To facilitate the **list_interval** depth-first search, information is added to each vertex about all edge tails with which the vertex is incident. This is maintained as a circular list from the vertex through each such edge and back to the vertex; details are presented in Figure 11. As with the search tree method, the pseudocode presented here is quite high-level. The more detailed code is found in Appendix 7.4.

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

    wv_vertex = record
        //  in addition to what an implementation needs...
        //
        out :   edge_id;
    end wv_vertex;

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

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

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

globals
    V : array [0..ν_limit] of wv_vertex;        //  any O(1) access time structure
    E : array [0..ε_limit] of wv_edge;          //  any O(1) access time structure
```

Figure 11. Wavefront Method Data Structure Modifications

Remain aware that in the following algorithms only end-bound and tail candidate vertices are maintained in the cross-timeline path records. "Consecutive" vertices recorded on the same timeline will no longer necessarily have immediately consecutive versions (though they will, of course, be in order). Furthermore, a vertex b referenced as a cross-timeline path origin can later be removed from the path records when it is no longer an **e** or **t** candidate. Even with b itself removed from the path records, though, virtually no references to b are altered since all lookups in the WVM algorithms search relative to their target ($\leq$ or $\geq$ the target's version). For this example, lookup results would either find some vertex a **preceding** b or some vertex c **following** b, whichever is appropriate.

The **add_vertex** procedure is similar to that of the search tree method. The only additions are initializing the list of edges originating at the vertex and putting v_q into the appropriate enabling sets.

```
procedure add_vertex (new_V : vertex;
                      T_on : set of timeline_id; candidate_for : set of candidacy;
                      out v_q : vertex_id);
    t : timeline_id;
    e_T : edge_id;                        //  not used, in this case
begin
    v += 1;
    v_q := v;
    V[v_q] := ⟨new_V, id_null⟩;
    for each t ∈ T_on do
        add_edge(T[t].self→last()→vid, v_q, e_T);
    endfor;
    //  Check for each of t, h, s, and e candidacies and add to appropriate enabling sets.
    //
    if s ∈ candidate_for then
        B_s ∪= {v_q};
    endif;
        ...
    return;
end add_vertex;
```

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 vertex, it does not treat a path between two vertices on the same timeline as a special case. **Update_tl_xt** is shown in

```
procedure update_tl_xt(t : ^timeline; v_term : vertex_id;
                        origins : ordering_set);
    v_term′ :  vertex_id;
    xt :       ^origin_paths;
    origin : ^ordering;
begin
    v_term′ := the first e or t candidate ≥ v_term on t;

    for origin ∈ origins do
        xt := t→xtpaths[origin→tid];

        if no existing paths to t originate from that timeline then
            add a new cross-timeline path set to t→xtpaths;
            add the initial v_0 to that set;
        endif;

        if a path from origin is not redundant then
            add the (origin, v_term′) path to xt;
            remove existing paths made redundant by this new path;
        endif;
    endfor;

    return;
end update_tl_xt;
```

Figure 12. Wavefront Method **Update_tl_xt** Procedure

Figure 12; **add_edge** is shown in Figure 13.

The **disable_candidate** procedure, listed in Figure 14, executes in three basic steps:

- Remove v_c from the enabling sets designated in **not_candidate_for**. If v_c is still either an **e** or **t** candidate, **disable_candidate** is done.
- If not, remove v_c from the cross-timeline path records of each timeline t_w on which v_c is ordered. For each such t_w:
    - Find the next vertex v_c' on t_w **following** v_c.
    - For each path (v_{origin}, v_c), v_{origin} on t_x, change that path to (v_{origin}, v_c') unless there already exists a recorded path from some vertex on t_x to v_c'. In that case, remove (v_{origin}, v_c) because it is made redundant by the existing path terminating with v_c'.

On timelines for which v_c is the *origin* of a path, there is no need to alter records because all necessary references to v_c are made with '$\leq$' or '$\geq$', not '='. More importantly, however, those

```
procedure add_edge (vt, vh : vertex_id; out er : edge_id);
    t : ^timeline;
    v_org, v_term : vertex_id;
    xt : ^origin_paths;
    origins : ordering_set := ∅;
begin
    //  Add er to E and to edge list at vt
    //
    ε += 1;
    er := ε;
    if this is the first edge with tail vt then
        E[er] := ⟨vertex_link, vt, vh⟩;
    else
        E[er] := ⟨edge_link, V[vt].out, vh⟩;
    endif;
    V[vt].out := er;

    //  Find all vertices which are now ≺ vh
    //
    t := any timeline such that vt ∈ V(t);
    for xt ∈ t→xtpaths do
        find the latest v_org ≺ vt on xt's origin timeline;
        if v_org ≠ v0 then                     //  everything follows v0; ignore it
            origins += ⟨xt→org_tid, version(v_org, xt→org_tid)⟩;
        endif;
    endfor;

    for each t such that vt ∈ V(t) do
        origins += ⟨t, version(vt, t)⟩;
    endfor;

    //  Update vh to follow origins
    //
    for each t such that vh ∈ V(t) do
        update_tl_xt(t, vh, origins);
    endfor;

    //  Update all vertices which follow vh to follow origins
    //
    for t ∈ T do
        v_term := the earliest vertex on t which follows vh;
        if v_term ≠ id_null then
            update_tl_xt(t, v_term, origins);
        endif;
    endfor;

    return;
end add_edge;
```

Figure 13. Wavefront Method **Add_edge** Procedure

```
procedure disable_candidate (vc : vertex_id; not_candidate_for : set of candidacy);
    vc' : vertex_id;
    xt : ^origin_paths;
    p :  ^x_tl_path;
    t :  ^timeline;
begin
    //      Check for each of t, h, s, and e candidacies and remove from appropriate
    //  enabling sets.
    //
    if s ∈ not_candidate_for then
        Bs -= {vc};
    endif;
        ...
    //      If this operation made vc be neither an e nor t candidate, remove
    //  it from the path records of all timelines on which it is ordered.
    //
    if vc ∉ Be and vc ∉ At then
        for each t such that vc ∈ V(t) do
            vc' := the next vertex on t which follows vc;

            //      Remove vc and change those path records with vc as terminus
            //  to show vc' as terminus, instead.
            //
            for xt ∈ t→xtpaths do
                p := xt→path[vc];            //  find a path p with vc as terminus
                if p ≠ null then
                    remove p from xt;

                    //      If a path to vc' already exists, it is from a higher-version
                    //  origin than that of the path to vc and should not be changed.
                    //
                    if xt→path[vc'] = null then
                        add a (p→org, vc') path to xt;
                    endif;
                endif;
            endfor;
        endfor;
    endif;
    return;
end disable_candidate;
```

Figure 14. Wavefront Method **Disable_candidate** Procedure

records <u>must</u> not be altered because of the case in which v_c is the origin of a path with terminus v_e, the end bound of a potential **list_interval** query. In this case, **list_interval** must be able to determine <u>exactly</u> where to cease putting vertices from v_c's timeline into I. The correct vertex on which to stop is v_c, not v_c'.

The **list_interval** query progresses as a series of passes between two sets of bounds, **todo_set** and **done_set**. **Todo_set** stores the earliest vertex on each timeline which is known to **follow** v_s but which has not yet been added to I; **done_set** contains the earliest vertex on each timeline which should no longer be added to I, 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 vertices <u>one version after</u> the latest vertices which **precede** v_e on each timeline, along with the next vertex after v_e on its own timeline. **Todo_set** begins with v_s. Note that only those timelines with which v_e 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" with respect to **list_interval**.

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

```
function list_interval (v_s, v_e : vertex_id) : set of vertex_id;
    I : srt_set of vertex_id := ∅;            //  avoid duplicates
    v_h, v_I_term : vertex_id;
    e :  edge_id;
    t :  ^timeline;
    xt : ^origin_paths;
    doing, next : ^wv_ordering;
    todo_set :  srt_set of wv_ordering key tid := ∅;
    done_set : ordering_set := ∅;
begin
    //      Find the latest vertex v_I_term ≺ v_e on each timeline with which v_e is ordered.
    //
    t := any timeline such that v_e ∈ V(t);
    for xt ∈ t→xtpaths do
        if xt→org_tid ≠ t→id then
            find the latest v_I_term ≺ v_e on xt's origin timeline;
        else                                  //  this will lead to putting v_e in I
            v_I_term is v_e itself;
        endif;
```

```
        if v_I_term ≠ v_0 then
            //  Add the vertex on T[xt→org_tid] just after v_I_term to done_set.
            //
            done_set += ⟨xt→org_tid, 1 + version(v_I_term, xt→org_tid)⟩;
        endif;
    endfor;

    //      Add all vertices between v_s and v_e to I, doing one span of a timeline's vertices
    //  at a time.
    //
    if v_s ≤ v_e then
        t := any timeline such that v_s ∈ V(t);
        todo_set += ⟨v_s, t→id, version(v_s, t)⟩;      //  start todo_set with v_s

        while todo_set ≠ ∅ do
            doing := todo_set.first();          //  pick any element from todo_set
            todo_set -= doing;                  //  ... and remove it

            //      Find where this span of vertices should terminate, then update
            //  done_set to show that another span is about to be completed.
            //
            v_I_term := get_vertex(done_set[doing→tid]);
            done_set += ⟨doing→tid, doing→ver⟩;

            while doing→vid ≺ v_I_term do
                I += doing→vid;

                //  Find where vertex 'doing' leads.
                //
                next := end of span;            //  in case doing is the terminus of its timeline
                for e ∈ V[doing→vid].out do     //  every edge whose tail is V[doing→vid]
                    v_h := E[e].head;
                    for each t such that v_h ∈ V(t) do
                        if t→id = doing→tid then
                            next := ⟨v_h, t→id, version(v_h, t)⟩;      //  just keep going along t
                        else
                            //      If v_h should be in I, is not already in I, and we have not
                            //  already recorded that it should be in I, record v_h in todo_set.
                            //
                            if v_h ≺ v_e andif v_h ∉ I andif v_h ≺ todo_set[t]→vid then
                                todo_set += ⟨v_h, t→id, version(v_h, t)⟩;
                            endif;
                        endif;
                    endfor;
                endfor;

                doing := next;
            endwhile;
        endwhile;
    endif;

    return make_set(I);                         //  convert from srt_set to set
end list_interval;
```

4.3 Analysis

For this analysis, it is useful to define $v_W \equiv$ the number of vertices which are either **e** or **t** candidates. It is much more difficult to calculate ε_W, the number of cross-timeline edges with this characteristic: the cross-timeline paths terminating with that edge have not all been reduced from the path records. The paths might be "moved" to vertices **following** their true terminus, but some still exist. With the search tree method, this was simply ε_X; with the wavefront method's reduction of perhaps several vertices' cross-timeline paths into that of the **following e-** or **t-**candidate vertex, $\varepsilon_W \leq \varepsilon_X$.

The ε_W measure will not, however, necessarily be the count of those vertices which are both the head of a cross-timeline edge and are either **e** or **t** candidates, $v_{X \cap W}$. The wavefront method's **disable_candidate** procedure can move path records from one vertex to another, not necessarily performing any combination of information at all. This would happen in the case of a path record moved from one vertex to a **following** e-candidate vertex which was not previously the head of any cross-timeline edge. The bounds which can generally be determined are 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 **disable_candidate** pseudocode reveals this same time complexity. A vertex v_c may be on τ timelines, each ordered with τ others. Updating a path record takes $O(\log \varepsilon_W)$ time if that record is not for a timeline on which v_c is ordered, 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 of ***I*_terms**. 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 vertices within the interval. Also, for each edge whose tail is a vertex in the interval (let the count of such edges be ε_I), there is at least an $O(\log \tau)$ search and perhaps τ $O(\log \tau)$ updates of **todo_set**, one for each timeline on which the edge's head is ordered.

Total locate time for **list_interval** is therefore $O(\tau \log \varepsilon_W + \tau \varepsilon_I \log \tau)$, while copy time is $O(v_I(\tau + \log \tau))$, or just $O(\tau v_I)$. As with the search tree method, the τ factors in the $O(\tau \varepsilon_I \log \tau)$ and $O(\tau v_I)$ terms are considered quite pessimistic since they are present only due to the possibility of vertices ordered on $O(\tau)$ timelines. For the common case of vertices ordered on $O(1)$ timelines, copy time would be $O(v_I \log \tau)$.

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 $O(\tau \varepsilon_W + \nu_W)$.

5. Bounded-Search Method

5.1 Approach

The fourth method investigated to resolve LIST_INTERVAL attempts to minimize space requirements at the expense of speed. No cross-timeline path records are made except for the edges themselves. The interval $[v_s \Rightarrow v_e]$ is determined by what appears, at first, to be a sequence of two brute-force depth-first searches: one from v_s forward, the other from v_e back. The query is resolved when the two searches meet at common vertices.

It is obvious that a simple search from v_s forward will terminate only at v_e or at the end of the history graph. This, alone, might not be too inefficient if queries are posed shortly after their end bound becomes known. The search back from v_e, however, will not necessarily terminate until the beginning of the history graph: potentially thousands of vertices will be uselessly scanned. The backwards search *must* be bracketed. Preferably, the forward search should be bracketed as well.

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

List_interval begins with a forward depth-first search from v_s towards v_e. Each probe of the search is stopped when some vertex v_{I_term} is encountered such that $\mathbf{top}(v_{I\_term}) \geq \mathbf{top}(v_e)$. This guarantees that **list_interval** has not searched past v_e, but does not imply that all vertices v_i which have been scanned are in $[v_s \Rightarrow v_e]$ — it is known that $v_i \nsucc v_e$, but not that $v_i \prec v_e$ $(v_s \prec v_{I\_term} \nsucc v_e)$.

The second search, back from v_e, finishes **list_interval**. Each probe of this search stops when it encounters any vertex scanned by the first search, or when a vertex v_{I_org} is encountered such that $\mathbf{top}(v_{I\_org}) \leq \mathbf{top}(v_s)$. In other words, when v_{I_org} can no longer **follow** v_s and thus can not be in the interval $(v_{I\_org} \nsucc v_s)$. When, as described above, the full forward search is

---

* 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.

performed before the backwards search is done, one only need search back a single edge from v_e. It is for variations on this approach that the topological bound on the backwards search is needed.

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

6. Future Work

6.1 Simulation

Simulators have been developed for both the search tree and wavefront interval-detection methods. These programs support both a command-line interface suitable for batch performance analysis and a graphical interface which can animate all updates of the search tree and wavefront method path records in real time. Comparison of the actual time and space characteristics of these two methods is ongoing. The simulation test-case generators which drive these tests allow a variety of graphs to be presented to the algorithms, from graphs containing uniformly random cross-timeline edges to graphs with edges characteristic of localized "communication" between timelines such as that experienced in a ring or hypercube.

6.2 Enhanced Queries

One of the advantages of interval logic is its ability to express nested intervals. The algorithms presented in this report 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 vertices v_i which must **follow** the start bound v_s and **precede** the end bound v_e: $v_s \preceq v_i \preceq v_e$. In many situations, however, it may be desirable to know those vertices which could **follow** v_s and **precede** v_e: $v_s \nsucc v_i \wedge v_i \nsucc v_e$. This issue of temporal ambiguity is one inherent in distributed time, an area of concern for the monitor 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 leading to the work presented in this report is the temporal analysis of events generated in a distributed system. It is therefore useful to know whether these algorithms can themselves be distributed, or instead require a centralized control which could become a performance bottleneck. Study shows that the search tree and wavefront methods can be distributed without excessive inter-process communication. Maintenance of the topological ordering used by the bounded-search method, however, appears to best be performed in a centralized manner, though this is not certain. 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

```
statement;
{statement;}
```

Conditional

```
if condition then
    sequence;
else
    sequence;
endif;
```

Alternative

```
case expression of
    value_list:
        (sequence;);
    ...
    others:
        (sequence;);
endcase;
```

Iteration

```
for variable in range do
    sequence;
endfor;
```

Repetition, Test At Entry

```
while condition do
    sequence;
endwhile;
```

Repetition, Test At Exit

```
repeat
    sequence;
until condition;
```

Procedure

procedure *proc_name(formal_parameters)*;
    declarations;
begin
    sequence;
    return;
end *proc_name*;

Function

function *func_name(formal_parameters)* :
        result_type;
    declarations;
begin
    sequence;
    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**, and **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. As with C++, indexing of an array and of a dereferenced pointer to an array is <u>not</u> distinguished; if **a_p** is a pointer to an array, a_p^[i] and a_p[i] are equivalent. Records can have Pascal-like variant fields. Syntax is:

Subrange

subrange_type =
    range [*first..last*]
        of *base_type*;

Enumeration

enumeration_type =
    (*value*{, *value*});

Array

array_type =
    array [*range*{, *range*}]
    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^
Also,
pointer_variable→
is equivalent to
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*).

Algorithms which perform a search for a particular element in a sorted set and then scan successive elements of that set starting at that search point are quite common. To this end, operations **next** and **prev** are provided to scan in increasing and decreasing order, respectively. If no further elements exist in that "direction" in the set, these operations return **null**. So that a scan may begin at either the start or end of a sorted set, the operations **first** and **last** are provided. These operations return the appropriate element, or **null** if the set is empty.

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 ≥,
next(), prev(), first(), last(), and the assignment operators += and -=

constants: ∅ — 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.[6] Edges are added and paths reported in $O(\nu)$ amortized time per operation, $\nu \equiv |V|$; the existence of a path may be checked in $O(1)$ (constant) time. The data structures require $\Theta(\nu^2)$ space.

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

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

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

globals
    //  index[v_i, v_j] ≠ null → 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 [vertex_id, vertex_id] of ^Ital_node := null;

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

```
procedure Ital_initialize();
    v_i, v_j : vertex_id;
begin
    for v_i in [0..v_limit] do
        desc[v_i] := new(Ital_node);
        desc[v_i]^ := ⟨v_i, null, null, null⟩;
        for v_j in [0..v_limit] do
            index[v_i, v_j] := null;
        endfor;
    endfor;
    return;
end Ital_initialize;
```

```
function Ital_check_path (v_org, v_term : vertex_id) : Boolean;
begin
    return index[v_org, v_term] ≠ null;
end Ital_check_path;
```

```
function Ital_get_path (v_org, v_term : vertex_id) : list of vertex_id;
    p : list of vertex_id := [ ];              // path from v_org to v_term
    curr_vertex : ^Ital_node;
begin
    if index[v_org, v_term] ≠ null then         // v_term is reachable from v_org
        curr_vertex := index[v_org, v_term];    // locate terminus in desc[v_org]
        p := [v_term];
        repeat                                  // go up in desc[v_org]
            curr_vertex := curr_vertex→parent;
            p := [curr_vertex→key] & p;         // prepend vertex to path (&= appends)
        until curr_vertex→parent = null;        // ... until we reach v_org
    endif;
    return p;
end Ital_get_path;
```

```
procedure Ital_add_edge (v_t, v_h : vertex_id);
    v_org : vertex_id;                          //  some vertex ≺ v_t
begin
    if index[v_t, v_h] = null then              //  no path already recorded from v_t to v_h
        for v_org in [0..v_limit] do
            if index[v_org, v_t] ≠ null and index[v_org, v_h] = null then
                //  The edge ⟨v_t, v_h⟩ gives rise to a new path from v_org to v_h
                //
                meld(v_org, v_h, v_t, v_h);     //  update desc[v_org] by means of desc[v_h]
            endif;
        endfor;
    endif;
    return;
end Ital_add_edge;


//      Merge desc[v_org] with a pruned subtree of desc[v_meld] rooted at v_sub_meld.
//  The vertex of desc[v_org] to which the pruned subtree will be grafted is v_org_link.  By
//  "pruning," we mean removing those vertices in desc[v_meld] which are already in desc[v_org].
//
procedure meld(v_org, v_meld, v_org_link, v_sub_meld : vertex_id);
    parent, child : ^Ital_node;
begin
    //  Insert the root of v_sub_meld into desc[v_org] as a child of v_org_link
    //
    if v_org = v_org_link then                  //  index does not contain self-loops
        parent := desc[v_org_link];
    else
        parent := index[v_org, v_org_link];
    endif;
    index[v_org, v_sub_meld] := new(Ital_node);
    index[v_org, v_sub_meld]^ :=                //  ⟨key, parent, child, sibling⟩
        ⟨v_sub_meld, parent, null, parent→child⟩;
    parent→child := index[v_org, v_sub_meld];
    for each child of v_sub_meld in desc[v_meld] do    //  find child, then follow siblings
        //  If the child and its subtree are not already in desc[v_org], add them
        //
        if index[v_org, child→key] = null then
            meld(v_org, v_meld, v_sub_meld, 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
    ν_limit, ε_limit : integer := some large positive number     //  greatest # of elements
    id_null : integer := -1;                                       //  "no such object"

types
    natural =      range [0..] of integer;
    vertex_id =    range [id_null..ν_limit] of integer;
    edge_id =      range [id_null..ε_limit] of integer;
    timeline_id = range [id_null..] of integer;
    version_index = natural;

    ordering = record                               //  version (order) of a vertex on a timeline
        tid :  timeline_id;
        ver : version_index;
    end ordering;

    ordering_set = srt_set of ordering key tid;

    vertex = record
        on : list of ordering;                      //  though a list, this is sorted by tid
        //  ... and whatever an implementation needs to keep track of
    end vertex;

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

    //       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 vertex
    //  identifier of the terminus since the origin's version would simply be terminus
    //  version - 1.
    //
    x_tl_path = record
        case (cross_timeline, in_timeline) of
            cross_timeline : (org : version_index;);
            in_timeline :     (vid : vertex_id;);
        endcase;
        term : version_index;
    end x_tl_path;
```

```
origin_paths = record
    org_tid :  timeline_id;                      //  id of tl on which origins are ordered
    path :  srt_set of x_tl_path key term, org;  //  we need to search by either field
end origin_paths;

timeline = record
    id :       timeline_id;
    self :     ^origin_paths;                    //  convenience: always points to xtpaths[id]
    xtpaths : srt_set of origin_paths key org_tid;
end timeline;

globals
    V : array [0..ν_limit] of vertex;            //  any O(1) access time structure
    ν : natural := 0;                            //  current number of vertices
    E : array [0..ε_limit] of edge;              //  any O(1) access time structure
    ε : natural := 0;                            //  current number of edges
    T : srt_set of timeline key id;


procedure add_vertex (new_V : vertex; T_on : set of timeline_id;
                      out v_q : vertex_id);
    sorted_T_on : srt_set of timeline_id;        //  so that the vertex's timelines can later be
                                                 //      referenced in order
    t :  timeline_id;
    e_r : edge_id;                               //  not used, in this case
begin
    sorted_T_on := make_srt_set(T_on);
    ν += 1;
    v_q := ν;
    V[v_q] := new_V;                             //  store application-specific fields
    for t ∈ sorted_T_on do
        V[v_q].on &= ⟨t, T[t].self→last()→ver⟩;
        add_edge(T[t].self→last()→vid, v_q, e_r);
    endfor;
    return;
end add_vertex;
```

```
procedure update_tl_xt(t : ^timeline; v_term : vertex_id;
                       origins : list of ordering;
                       ver_term : version_index);   //  version of v_term on t
    xt : ^origin_paths;
    p : ^x_tl_path;
    origin : ^ordering;
begin
    for origin ∈ origins do
        xt := t→xtpaths[origin→tid];

        if xt = null then
            t→xtpaths += ⟨origin→tid, ∅⟩;
            xt := t→xtpaths[origin→tid];
            //  Record a path to t from v_0 on the new origin timeline.
            //
            if origin→tid ≠ t→id then       //  between t and some other timeline
                //      make that path terminate with the first vertex on t, which might no
                //  longer be version 0 if garbage collection has taken place
                //
                p := t→self→path(≥, 1);
                xt→path += ⟨0, p→term⟩;
            else                            //  t itself
                t→self := xt;
                xt→path += ⟨v_term, 1⟩;     //  remember the STM space optimization
            endif;
        endif;

        if origin→tid ≠ t→id then
            if xt→path(≤, ver_term)→org < origin→ver then
                xt→path += ⟨origin→ver, ver_term⟩;

                //  Remove out-of-order paths
                //
                p := xt→path(>, ver_term);
                while p ≠ null andif p→org ≤ origin.ver then
                    xt→path -= p;
                    p := xt→path(>, ver_term);
                endwhile;
            endif;
        else                                //  xt = t→self
            if xt→path(≤, ver_term)→term-1 < origin→ver then  //  term-1 ≡ org for t→self
                xt→path += ⟨v_term, ver_term⟩;
            endif;
        endif;
    endfor;

    return;
end update_tl_xt;
```

```
procedure add_edge (v_t, v_h : vertex_id; out e_r : edge_id);
    t : ^timeline;
    tid_h, tid_on : timeline_id;
    ver_org, ver_term, ver_t, ver_h, ver_on : version_index;
    xt :  ^origin_paths;
    ord : ^ordering;
    origins : list of ordering := [ ];
begin
    ε += 1;
    e_r := ε;
    E[e_r] := ⟨v_t, v_h⟩;
    //      Check if v_t = v_0. If so, only want to cross-reference each timeline on which v_h
    //  is ordered with each other such timeline, not with all the timelines in the graph (v_0
    //  is ordered on every timeline). To do otherwise would be quite inefficient, though not
    //  actually wrong, because it would increase search time for every timeline's xtpaths.
    //
    if v_t ≠ v_0 then
        //  Find all vertices which are now ≺ v_h. This is v_t and those vertices ≺ v_t.
        //
        tid_on := V[v_t].on(0)→tid;            //  Find any timeline on which v_t is ordered. The
        ver_on := V[v_t].on(0)→ver;            //      first such timeline is used because we must
        t := T[tid_on];                        //      scan V[v_t].on from the beginning, anyway.
        ver_t := ver_on;
        for xt ∈ t→xtpaths do                  //  scanned in increasing org_tid sequence
            if t→id ≠ tid_on then
                //  find the latest v_org ≺ v_t on xt's origin timeline
                //
                ver_org := xt→path(≤, ver_t)→org;
                if ver_org ≠ 0 then            //  everything follows v_0; ignore it
                    origins &= ⟨xt→org_tid, ver_org⟩;
                endif;
            else
                //  We want v_t's version itself, not that of the vertex before v_t
                //
                origins &= ⟨tid_on, ver_on⟩;
                ord := V[v_t].on.next();       //  next timeline on which v_t is ordered
                if ord ≠ null then
                    tid_on := ord→tid;
                    ver_on := ord→ver;
                else
                    tid_on := id_null;
                endif;
            endif;
        endfor;
```

```
    else
        for ord ∈ V[v_h].on do
            origins &= ⟨ord→tid, 0⟩;
        endfor;
    endif;

    // Update v_h to follow origins
    //
    for ord ∈ V[v_h].on do
        update_tl_xt(T[ord→tid], v_h, origins, ord→ver);
    endfor;

    // Update all vertices which follow v_h to follow origins
    //
    tid_h := V[v_h].on(0)→tid;          // a reference point for comparisons against v_h
    ver_h := V[v_h].on(0)→ver;
    for t ∈ T, t→id ≠ tid_h do          // if t→id=tid_h, the STM space optimization conflicts
        xt := t→xtpaths[tid_h];          // is any vertex on t ordered with T[tid_h]?
        if xt ≠ null andif xt→path.org(≥, ver_h) ≠ null then
            // find the earliest vertex on t which follows v_h
            //
            ver_term := xt→path.org(≥, ver_h)→term;
            update_tl_xt(t, id_null, origins, ver_term);    // v_term unnecessary here
        endif;
    endfor;

    return;
end add_edge;
```

```
function list_interval (v_s, v_e : vertex_id) : set of vertex_id;
    I : srt_set of vertex_id := ∅;              //  avoid duplicates
    I_terms : list of ordering := [ ];          //  termini of all spans of vertices making up I
    I_term :  ^ordering;
    ver_s, ver_e, ver_I_term : version_index;
    p, p_I_term : ^x_tl_path;
    t :    ^timeline;
    tid_s : timeline_id;
    xt :   ^origin_paths;
begin
    //      Find the latest vertex before v_e for each timeline with which v_e is ordered.
    //
    t := T[V[v_e].on(0)→tid];                   //  any (here, first) timeline on which v_e is ordered
    ver_e := V[v_e].on(0)→ver;
    for xt ∈ t→xtpaths do
        if xt ≠ t→self then
            //  find the latest v_I_term ≺ v_e on xt's origin timeline
            //
            ver_I_term := xt→path(≤, ver_e)→org;
        else                                    //  this will lead to putting v_e in I
            ver_I_term := ver_e;
        endif;
        if ver_I_term ≠ 0 then                  //  again, ignore v_0
            I_terms &= ⟨xt→org_tid, ver_I_term⟩;
        endif;
    endfor;

    //      Add all vertices after v_s and before v_e to I, scanning one timeline at a time
    //  between the first vertex after v_s and the latest vertex before v_e (stored in I_terms).
    //
    tid_s := V[v_s].on(0)→tid;                  //  any (here, first) timeline on which v_s is ordered
    ver_s := V[v_s].on(0)→ver;
    for I_term ∈ I_terms do
        t := T[I_term→tid];
        xt := t→xtpaths[tid_s];                 //  we want paths from t_s to t
        if xt ≠ null then
            if xt ≠ t→self then
                //  find the earliest vertex ≥ v_s on t
                //
                p_I_org := xt→path.org(≥, ver_s);  //  can not search by org on t→self
                if p_I_org ≠ null andif p_I_org→term ≤ I_term→ver then
                    xt := t→self;
                    for each p ∈ xt→path
                        with p→term ∈ [p_I_org→term, I_term→ver] do
                        I += p→vid;
                    endfor;
                endif;
```

```
            else
                if vers < I_term→ver then
                    for each p ∈ xt→path
                        with p→term ∈ [vers, I_term→ver] do
                        I += p→vid;
                    endfor;
                endif;
            endif;
        endif;
    endfor;
    return make_set(I);                    //  convert from srt_set to set
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.

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

types
    natural =      range [0..] of integer;
    vertex_id =    range [id_null..ν_limit] of integer;
    edge_id =      range [id_null..ε_limit] of integer;
    timeline_id = range [id_null..] of integer;
    version_index = natural;

    ordering = record                                 //  version (order) of a vertex on a timeline
        tid : timeline_id;
        ver : version_index;
    end ordering;

    ordering_set = srt_set of ordering key tid;

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

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

    wv_vertex = record
        on : list of ordering;                        //  though a list, this is sorted by tid
        out : edge_id;
        //  ... and whatever an implementation needs to keep track of
    end wv_vertex;

    next_edge = (edge_link, vertex_link);
```

```
wv_edge = record
    case link : next_edge of
        edge_link :   (next : edge_id;);
        vertex_link : (tail :  vertex_id;);
    endcase;
    head : vertex_id;
end wv_edge;

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

origin_paths = record
    org_tid :  timeline_id;                  //  id of tl on which origins are ordered
    path :  srt_set of x_tl_path key term, org; //  we need to search by either field
end origin_paths;

timeline = record
    id :       timeline_id;
    self :     ^origin_paths;               //  convenience: always points to xtpaths[id]
    xtpaths : srt_set of origin_paths key org_tid;
end timeline;

globals
    V : array [0..v_limit] of wv_vertex;     //  any O(1) access time structure
    ν : natural := 0;                        //  current number of vertices
    E : array [0..ε_limit] of wv_edge;       //  any O(1) access time structure
    ε : natural := 0;                        //  current number of edges
    T : srt_set of timeline key id;

    A_t : set of vertex_id;                  //  vertices which may later be an edge tail
    A_h : set of vertex_id;                  //  vertices which may later be an edge head
    B_s : set of vertex_id;                  //  vertices which may be a query start bound
    B_e : set of vertex_id;                  //  vertices which may be a query end bound
```

```
procedure add_vertex (new_V : vertex;
                      T_on : set of timeline_id; candidate_for : set of candidacy;
                      out v_q : vertex_id);
    sorted_t : srt_set of timeline_id;      //  so we can later reference a vertex's
                                            //      timelines in order
    t : timeline_id;
    e_r : edge_id;                          //  not used, in this case
begin
    sorted_t := make_srt_set(T_on);
    ν += 1;
    v_q := ν;
    V[v_q] := ⟨new_V, id_null⟩;
    for t ∈ sorted_t do
        V[v_q].on &= ⟨t, T[t].self→last()→ver⟩;
        add_edge(T[t].self→last()→vid, v_q, e_r);
    endfor;
    //  Check for each of t, h, s, and e candidacies and add to appropriate enabling sets.
    //
    if t ∈ candidate_for then
        A_t ∪= {v_q};
    endif;
    if h ∈ candidate_for then
        A_h ∪= {v_q};
    endif;
    if s ∈ candidate_for then
        B_s ∪= {v_q};
    endif;
    if e ∈ candidate_for then
        B_e ∪= {v_q};
    endif;
    return;
end add_vertex;
```

```
procedure update_tl_xt(t : ^timeline;
                       origins : list of ordering;
                       ver_term : version_index);   //   version of v_term on t
   ver_term′ : version_index;
   xt : ^origin_paths;
   p : ^x_tl_path;
   origin : ^ordering;
begin
   //   Find the first e or t candidate following v_term on t.
   //
   if t→self ≠ null then
      p := t→self→path(≥, ver_term);
      if p ≠ null then
         ver_term′ := p→term;
      else
         ver_term′ := ver_term;
      endif;
   else
      ver_term′ := ver_term;
   endif;

   for origin ∈ origins do
      xt := t→xtpaths[origin→tid];

      if xt = null then
         t→xtpaths += ⟨origin→tid, ∅⟩;
         xt := t→xtpaths[origin→tid];

         //   Record a path to t from v_0 on the new origin timeline.
         //
         if origin→tid ≠ t→id then        //   between t and some other timeline
            //       make that path terminate with the first vertex on t, which might no
            //   longer be version 0 if garbage collection has taken place
            //
            p := t→self→path(≥, 1);
            xt→path += ⟨0, p→term⟩;
         else                              //   t itself
            t→self := xt;
            xt→path += ⟨0, 1⟩;
         endif;
      endif;
```

```
        if xt→path(≤, ver_term′)→org < origin→ver then
            xt→path += ⟨origin→ver, ver_term′⟩;

            //  Remove out-of-order paths
            //
            p := xt→path(>, ver_term′);
            while p ≠ null andif p→org ≤ origin.ver then
                xt→path -= p;
                p := xt→path(>, ver_term′);
            endwhile;
        endif;
    endfor;

    return;
end update_tl_xt;


procedure add_edge (v_t, v_h : vertex_id; out e_r : edge_id);
    t : ^timeline;
    tid_h, tid_on : timeline_id;
    ver_org, ver_term, ver_t, ver_h, ver_on : version_index;
    xt :  ^origin_paths;
    ord : ^ordering;
    origins : list of ordering := [ ];
begin
    ε += 1;
    e_r := ε;
    if V[v_t].out = id_null then
        E[e_r] := ⟨vertex_link, v_t, v_h⟩;
    else
        E[e_r] := ⟨edge_link, V[v_t].out, v_h⟩;
    endif;
    V[v_t].out := e_r;

    //      Check if v_t = v_0.  If so, only want to cross-reference each timeline on which v_h
    //  is ordered with each other such timeline, not with all the timelines in the graph (v_0
    //  is ordered on every timeline).  To do otherwise would be quite inefficient, though not
    //  actually wrong, because it would increase search time for every timeline's xtpaths.
    //
    if v_t ≠ v_0 then
        //  Find all vertices which are now ≺ v_h.  This is v_t and those vertices ≺ v_t.
        //
        tid_on := V[v_t].on(0)→tid;          //  Find any timeline on which v_t is ordered.  The
        ver_on := V[v_t].on(0)→ver;          //      first such timeline is used because we must
        t := T[tid_on];                      //      scan V[v_t].on from the beginning, anyway.
        ver_t := ver_on;
```

```
        for xt ∈ t→xtpaths do                  //  scanned in increasing org_tid sequence
            if t→id ≠ tid_on then
                //  find the latest v_org ≺ v_t on xt's origin timeline
                //
                ver_org := xt→path(≤, ver_t)→org;
                if ver_org ≠ 0 then            //  everything follows v_0; ignore it
                    origins &= ⟨xt→org_tid, ver_org⟩;
                endif;
            else
                //  We want v_t's version itself, not that of the vertex before v_t
                //
                origins &= ⟨tid_on, ver_on⟩;
                ord := V[v_t].on.next();       //  next timeline on which v_t is ordered
                if ord ≠ null then
                    tid_on := ord→tid;
                    ver_on := ord→ver;
                else
                    tid_on := id_null;
                endif;
            endif;
        endfor;

    else
        for ord ∈ V[v_h].on do
            origins &= ⟨ord→tid, 0⟩;
        endfor;
    endif;

    //  Update v_h to follow origins
    //
    for ord ∈ V[v_h].on do
        update_tl_xt(T[ord→tid], origins, ord→ver);
    endfor;

    //  Update all vertices which follow v_h to follow origins
    //
    tid_h := V[v_h].on(0)→tid;                 //  a reference point for comparisons against v_h
    ver_h := V[v_h].on(0)→ver;
    for t ∈ T do
        xt := t→xtpaths[tid_h];                //  is any vertex on t ordered with T[tid_h]?
        if xt ≠ null andif xt→path.org(≥, ver_h) ≠ null then
            //  find the earliest vertex on t which follows v_h
            //
            ver_term := xt→path.org(≥, ver_h)→term;
            update_tl_xt(t, origins, ver_term);
        endif;
    endfor;

    return;
end add_edge;
```

```
procedure disable_candidate (v_c : vertex_id; not_candidate_for : set of candidacy);
    ver_c, ver_c' : vertex_index;
    xt :  ^origin_paths;
    p :   ^x_tl_path;
    t :   ^timeline;
    ord : ^ordering;
begin
    //      Check for each of t, h, s, and e candidacies and remove from appropriate
    //   enabling sets.
    //
    if t ∈ not_candidate_for then
        A_t -= {v_c};
    endif;
    if h ∈ not_candidate_for then
        A_h -= {v_c};
    endif;
    if s ∈ not_candidate_for then
        B_s -= {v_c};
    endif;
    if e ∈ not_candidate_for then
        B_e -= {v_c};
    endif;

    //      If this operation made v_c be neither an e nor t candidate, remove
    //   it from the path records of all timelines on which it is ordered.
    //
    if v_c ∉ B_e and v_c ∉ A_t then
        for ord ∈ V[v_c].on do
            t := T[ord→tid];
            ver_c := ord→ver;
            ver_c' := t→self→path(>, ver_c)→term;   //   next vertex on t following v_c

            //      Remove v_c and change those path records with v_c as terminus
            //   to show v_c' as terminus, instead.
            //
            for xt ∈ t→xtpaths do
                p := xt→path[ver_c];                //   find a path p with v_c as terminus
                if p ≠ null then
                    xt→path -= p;

                    //      If a path to v_c' already exists, it is from a higher-version
                    //   origin than that of the path to v_c and should not be changed.
                    //
                    if xt→path[ver_c'] = null then
                        xt→path += ⟨p→org, ver_c'⟩;
                    endif;
                endif;
            endfor;
        endfor;
    endif;

    return;
end disable_candidate;
```

```
function list_interval (v_s, v_e : vertex_id) : set of vertex_id;
    I : srt_set of vertex_id := ∅;             //  avoid duplicates
    v_h : vertex_id;
    ver_s, ver_e, ver_h, ver_I_term : version_index;
    tid_s, tid_h : timeline_id;
    ord : ordering;
    t :  ^timeline;
    e :  edge_id;
    xt : ^origin_paths;
    doing, next : ^wv_ordering;
    todo_set : srt_set of wv_ordering key tid := ∅;
    done_set : ordering_set := ∅;

begin
    //       Find the latest vertex v_I_term ≺ v_e on each timeline with which v_e is
    //   ordered.
    //
    t := T[V[v_e].on(0)→tid];                //  find some timeline on which v_e is ordered
    ver_e := V[v_e].on(0)→ver;
    for xt ∈ t→xtpaths do
        if xt→org_tid ≠ t→id then
            ver_I_term := xt→path(≤, ver_e)→org;    // latest vertex ≺ v_e on xt's origin timeline
        else                                  //  this will lead to putting v_e in I
            ver_I_term := ver_e;
        endif;
        if ver_I_term ≠ 0 then
            //   Add the vertex on T[xt→org_tid] just after v_I_term to done_set.
            //
            done_set += ⟨xt→org_tid, ver_I_term + 1⟩;
        endif;
    endfor;

    //       Add all vertices between v_s and v_e to I, doing one span of a timeline's vertices
    //   at a time.
    //
    tid_s := V[v_s].on(0)→tid;               //  find some timeline on which v_s is ordered
    ver_s := V[v_s].on(0)→ver;
    if done_set[tid_s] ≠ null andif done_set[tid_s]→ver > ver_s then  //  if v_s ≤ v_e then
        todo_set += ⟨v_s, tid_s, ver_s⟩;      //  start todo_set with v_s

        while todo_set ≠ ∅ do
            doing := todo_set.first();        //  pick any element from todo_set
            todo_set -= doing;                //  ... and remove it
```

```
                //      Find where this span of vertices should terminate, then update
                //  done_set to show that we are about to complete another span.
                //
                ver_I_term := done_set[doing→tid]→ver;
                done_set += ⟨doing→tid, doing→ver⟩;

                while doing→ver < ver_I_term do
                    I += doing→vid;

                    //  Find where vertex 'doing' leads.
                    //
                    next := ⟨id_null, doing→tid, ver_I_term⟩;  //  in case doing is the terminus
                                                               //      of its timeline
                    for e ∈ V[doing→vid].out do          //  every edge whose tail is V[doing→vid]
                        v_h := E[e].head;
                        for ord ∈ V[v_h].on do
                            tid_h := ord→tid;
                            ver_h := ord→ver;
                            if tid_h = doing→tid then
                                next := ⟨v_h, tid_h, ver_h⟩;      //  just keep going along t
                            else
                                //      If v_h should be in I, is not already in I, and we have not
                                //  already recorded that it should be in I, record v_h in todo_set.
                                //
                                if done_set[tid_h] ≠ null andif ver_h < done_set[tid_h]→ver
                                  andif (todo_set[tid_h] = null
                                         orelse ver_h < todo_set[tid_h]→ver) then
                                    todo_set += ⟨v_h, tid_h, ver_h⟩;
                                endif;
                            endif;
                        endfor;
                    endfor;

                    doing := next;
                endwhile;
            endwhile;
        endif;

        return make_set(I);                               //  convert from srt_set to set
    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. Horowitz, Ellis, and Sahni, Sartaj. Fundamentals of Data Structures. Rockville, MD: Computer Science Press, Inc., 1982.

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

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

7. 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.

8. 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.

9. 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.

10. 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.

11. Standish, Thomas A. Data Structure Techniques. Reading, MA: Addison-Wesley Publishing Company, Inc., 1980.

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

13. Willard, Dan E. "New Data Structures For Orthogonal Range Queries." SIAM Journal on Computing Vol. 14, no. 1 (February 1985): 232-253.