Determining Interior Vertices of Graph Intervals

Authors: 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
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 DCR-8600947, Bellcore, BNR, Italtel, NEC, DEC, SynOptics, and NTT. The author may be reached via email address vjg@wucs1.wustl.edu.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>No.</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.2 Terms</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.3 Problem Definition</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.4 Two Complexity Issues</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.5 History Graph Diagrams</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Transitive Closure Method</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.1 Approach</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.2 Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Search Tree Method</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>3.1 Approach</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>3.2 Algorithm</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>3.3 Analysis</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>3.4 Comparison With Transitive Closure Method</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Wavefront Method</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.1 Approach</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.2 Algorithm</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>4.3 Analysis</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Bounded-Search Method</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>5.1 Approach</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>Future Work</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>6.1 Simulation</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>6.2 Enhanced Queries</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>6.3 Distributed Implementations</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>Appendices</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Appendix 7.1 Pseudocode Representation</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>7.1.1 Control Structures</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>7.1.2 Operators</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>7.1.3 Simple and Structured Types</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>7.1.4 High-level Structured Types</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Appendix 7.2 Italiano’s Path Retrieval Algorithm</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Appendix 7.3 Search Tree Method Algorithm</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Appendix 7.4 Wavefront Method Algorithm</td>
<td>61</td>
</tr>
<tr>
<td>8</td>
<td>Bibliography</td>
<td>71</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>No.</th>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Figure 1.</td>
<td>History Graph Structure and Operations</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Figure 2.</td>
<td>Declaration of Required Operations</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Figure 3.</td>
<td>Operations Provided by Italiano’s Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Figure 4.</td>
<td>Cross-Timeline Path Information for Search Tree Method</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Figure 5.</td>
<td>Search Tree Method Data Structures</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>Figure 6.</td>
<td>Search Tree Method Update_tl_xt Procedure</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Figure 7.</td>
<td>Search Tree Method Add_edge Procedure</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Figure 8.</td>
<td>Search Tree Method List_interval Procedure</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Figure 9.</td>
<td>Worst-Case Space for Search Tree Method</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Figure 10.</td>
<td>Cross-Timeline Path Information for Wavefront Method</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>Figure 11.</td>
<td>Wavefront Method Data Structure Modifications</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>Figure 12.</td>
<td>Wavefront Method Update_tl_xt Procedure</td>
<td>31</td>
</tr>
<tr>
<td>13</td>
<td>Figure 13.</td>
<td>Wavefront Method Add_edge Procedure</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>Figure 14.</td>
<td>Wavefront Method Disable_candidate Procedure</td>
<td>33</td>
</tr>
</tbody>
</table>
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 \( H = (V, E) \) is a directed acyclic graph. A vertex \( v_i \in V \) corresponds to a single event \( V_i \) in our application. A directed edge \( e_k = (v_i, v_h) \in E \) corresponds to the temporal relationship "\( V_i \) occurred before \( V_h \)". Let \( n = |V| \), and \( c = |E| \).

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 '\( \leq \)' 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 \( a \leq v_i \leq b \) 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 \( a \not\prec b \), \( [a \Rightarrow b] = \emptyset \).

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 \( H_{q-1,r} = (V_{q-1}, E_r) \) and a vertex \( v_q \), construct \( H_{q,r} = (V_q, E_r) \) where \( V_q = V_{q-1} \cup \{v_q\} \).

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

- **LIST_INTERVAL.** Given a graph \( H = (V, E) \) and two vertices \( v_s \in V \) and \( v_e \in V \), construct a set \( I = [v_s \Rightarrow v_e] \). Define \( v_I \equiv |I| \equiv |[v_s \Rightarrow v_e]| \).

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 \textsc{Add\textunderscore Vertex} to accommodate this:

A \textbf{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 \textbf{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 \textbf{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 \textbf{cross-time edges}, make up the set
\( X = E - \bigcup_{t \in T} E(t) \). Let \( \tau = |T| \) and \( \epsilon_X = |X| \). The index of a vertex
within a path is referred to as its \underline{version} on that path; the vertex is said
to be \underline{ordered} on that path. A path with origin \( v_{\text{org}} \) and terminus \( v_{\text{term}} \) is
denoted by \( \langle v_{\text{org}}, v_{\text{term}} \rangle \).

\textsc{Add\textunderscore Vertex}. Given \( H = \langle G, T \rangle \), a vertex \( v_q \), a set \( T_{on} \subseteq T \), and a non-
negative integer \( \tau_{\text{new}} \), construct \( H' = \langle G', T' \rangle \). \( T' \) consists of the union of
three sets: \( T - T_{on} \) 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 \( \tau_{\text{new}} \) new paths each of which
contains only \( v_q \) : \( G' = \langle V', E' \rangle \), where \( V' = V \cup \{v_q\} \), and \( E' = E \cup \{\text{the new edges added to the paths in } T_{on}\} \).

These definitions of \( H \) and \textsc{Add\textunderscore Vertex} are effectively equivalent to the original
definitions if one enforces that every added vertex augment a unique timeline (i.e. \( T_{on} = \emptyset \) and
\( \tau_{\text{new}} = 1 \) for every \textsc{Add\textunderscore 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 \( \tau_{new} \) 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 \( A_i \) and \( A_h \) can be identified so that ADD_EDGE operations are restricted to edges \( (v_t, v_h) \) such that \( v_t \in A_i \) and \( v_h \in A_h \). The definitions of the above operations are suitably amended, and one more operation is defined:

Vertex sets \( B_s, B_e, A_i, \) 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 \( v_c \in B_s, B_e, A_i, \) 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 \( B_s, B_e, A_i, \) or \( A_h \). This is the only time \( v_q \) may be added to an enabling set.

**ADD_EDGE.** It is required that \( v_t \in A_i \) 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 \( B_s, B_e, A_i, \) 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 $O(n^2)$ space is therefore considered of no practical use. Given this, $O(e)$ 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 $\Omega(v)$ — 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 locate-and-copy time, is generally considered too coarse to be useful. Instead, the locate and copy 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 $O(v)$. 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.
Figure 1. History Graph Structure and Operations
of a single circle. A vertex \( v \) is said to be \textit{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 \textit{ordered with respect to each other} if either \( v < w \) or \( w < v \).

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 "use:" 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 cannot discern this property and does not disable \( v_1 \)'s \( h \) candidacy, proper query results are not affected but certain data structure optimizations cannot 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 \( A_v \), 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 \( \varepsilon \) 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 \(v_i \ni (a \leq v_i \leq b)\).

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 \(O(v)\) 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 \(\Theta(v^2)\).

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 \(O(v)\) running time, just the constant factor). Such restructuring would cause a bursty and unpredictable (and thus unacceptable) performance impact on the monitor application.
constants
v_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..v_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..v_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_ν : vertex;
  T_on : set of timeline_id; candidate_for : set of candidacy;
  out ν_q : vertex_id);

procedure add_edge (ν_s, ν_e : vertex_id; out ε_e : edge_id);

function list_interval (ν_s, ν_e : vertex_id) : set of vertex_id;

procedure disable_candidate (ν_e : 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:

```plaintext
procedure add_edge (vᵣ, v₈ : vertex_id; out eᵣ : edge_id);
begin
  Ital_add_edge(vᵣ, v₈);
  eᵣ := 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 $vᵣ$ 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[vᵣ, v₈] ≠ null → a path exists from vᵣ to v₈
  //
  index : array [vertex_id, vertex_id] of `^Ital_node` := null;
  desc : array [vertex_id] of `^Ital_node`;

procedure Ital_add_edge (vᵣ, v₈ : vertex_id);
function Ital_check_path (vᵣᵩ, v₈ᵩ : vertex_id) : Boolean;
function Ital_get_path (vᵣᵩ, v₈ᵩ : 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 $O(v)$ locate time. This is similar to the $O(v_j)$ locate-and-copy time limit for the query but
is perhaps much larger. Copy time is $O(v_j)$.

```java
function list_interval ($v_s$, $v_e$ : vertex_id) : set of vertex_id;
  $I$ : set of vertex_id := $\emptyset$;
  $v$ : vertex_id;
begin
  if index[$v_s$, $v_e$] $\neq$ null then
    $I$ $\cup$= [$v_s$, $v_e$];
    for $v$ in [0..v-1] do
      if index[$v_s$, $v$] $\neq$ null and index[$v$, $v_e$] $\neq$ null then
        $I$ $\cup$= [$v$];
      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 $O(v)$ (instead of $O(V\log V)$) 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 $O(\tau^2 \log \xi + \tau \log v)$ add_edge and $O(\tau(\log \xi + v P))$ list_interval time while requiring $O(\tau \xi + v)$ space.* Such space costs at first appear worse than those of Italiano’s algorithm because $e$, for a general graph, is $O(v^2)$. The monitor application’s removal of edges which are redundant through transitivity, however, makes $e$ 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 $e_t$ is added to $X$ can be performed with the following simplified procedure:

- Given $e_t = (v_t, v_h)$. 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 includes 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 $e_t$ 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 sorted set is a set totally ordered by a relation over a key attribute of each of the set’s elements. 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 \((v_{\text{origin}}, v_h)\) determined above, record the path \((v_{\text{origin}}, v_h)\) if it is not already implied through transitivity. This is the case whenever \(v_{\text{origin}}\) is also the origin of a path to some vertex on \(t_v\) which has an earlier version than \(v_h\).

The pairwise-timeline sorted sets are the reason for the \(\tau^2\) 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 \(\tau^2\) factors will actually be \(\tau\) 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 xpaths[id]
  xpaths : 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_oa 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
  v += 1;
  v_q := v;
  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 $v_t' < v_t$. For the general case, all vertices $v_h' > v_h$ must also follow all vertices $v_t' < v_t$. Given all vertices $v_t' < v_t$ and all vertices $v_h' > v_h$, the path records must be updated so that $v_t' < v_t < v_h < v_h'$. 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 $v_{\text{term}}$ on a timeline $t$ and a set of vertices (identified as $(\text{timeline_id}, \text{version_index})$'s) which are origins of paths to $v_{\text{term}}$. 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_{\text{term}} : vertex_id;
    origins : ordering_set);
    xt :   ^origin_paths;
    origin : ^ordering;
begin
    for origin \in 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 origin->tid \neq t->id then
            if a path from origin is not redundant then
                add the (origin, $v_{\text{term}}$) path to xt;
                remove existing paths made redundant by this new path;
            endif;
        else
            add (origin, $v_{\text{term}}$) 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
  e := 1;
  e_r := e;
  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 \( v_{\text{term}}' > v_{\text{term}} \) on \( t \). \text{Update_tl_xt} is given \( v_{\text{org}} \) on \( t_{\text{org}} \) so that it can record \( (v_{\text{org}}, v_{\text{term}}) \).

Additionally, the path \( (v_{\text{org}}', v_{\text{term}}') \) was previously recorded, \( v_{\text{org}}' \) also on \( t_{\text{org}} \). If \( v_{\text{org}}' \leq v_{\text{org}} \), explicitly recording \( (v_{\text{org}}', v_{\text{term}}') \) is no longer necessary because it can be determined through the transitive relationship \( v_{\text{org}}' \leq v_{\text{org}} < v_{\text{term}} < v_{\text{term}}' \). Figure 7 lists the search tree method's \text{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 \( T_f \).
- For each \( t \in T_f \), determine the earliest vertex on \( t \) which follows \( v_s \) and the latest vertex on \( t \) which precedes \( v_e \).
- For each \( t \in T_f \), 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 \text{list_interval} is shown in Figure 8.

3.3 Analysis

The \( O(\tau^2 \log e_X + \tau \log \nu) \) time for \text{add_edge} is calculated by direct examination of the procedure's pseudocode. Begin with inspection of \text{update_tl_xt}. The top level of this subroutine is a loop for each \text{origin} which \( v_{\text{term}} \) should follow; there could be \( \tau \) origins. Within the loop, \( v_{\text{term}} \)'s timeline is searched for the existing cross-timeline paths originating from \text{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 \((\text{origin}, v_{\text{term}})\) path is not redundant, it is recorded with either two \( O(\log e_X) \) or one \( O(\log \nu) \) insertion(s) (depending upon whether or not the path originates on \( v_{\text{term}} \)'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 \( O(e_X \log e_X) \) 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 \text{splay trees}[12] are used instead of \text{AVL trees} for the path records, two splay tree splits and a splay tree join, \( O(\log e_X) \), are all that is required to rectify the problem.

The above analysis yields an \( O(\tau (\log \tau + \log e_X) + \log \nu) \) running time for \text{update_tl_xt} (only one \text{origin} can be on \( v_{\text{term}} \)'s own timeline). One can, though, compare \( \tau \) and \( e_X \) in order.
function list_interval (v_s, v_e : vertex_id) : set of vertex_id;

l : srt_set of vertex_id := ∅;            // avoid duplicates
l_terms : list of ordering := [ ];       // termini of all spans of vertices making up l
l_term : ^ordering;

v_l_term, v_l : 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→xpaths do
    if xt ≠ t→self then
      find the latest v_l_term < v_e on xt's origin timeline;
    else
      v_l_term is v_e itself;
    endif;
    if v_l_term ≠ v_0 then
      l_terms &← (xt→org_tid, version(v_l_term, xt→org_tid));
    endif;
  endfor;

  // Add all vertices after v_s and before v_e to l, scanning one timeline at a time
  // between the first vertex after v_s and the latest vertex before v_e (stored in l_terms).
  //
  t_s := any timeline such that v_s ∈ V(t);
  for l_term ∈ l_terms do
    t := T(l_term→tid);
    v_l_term := get_vertex(l_term);
    xt := t→xpaths[t_s→tid];            // we want paths from t_s to t
    if xt ≠ null then
      v_l_org := the earliest vertex ≥ v_s on t;
      if v_l_org ≠ id_null and if v_l_org ≤ v_l_term then
        l := all vertices v_i on t s.t. (v_l_org ≤ v_i ≤ v_l_term);
      endif;
    endif;
  endfor;

return make_set(l);

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, \( \varepsilon_X \). Therefore, for this calculation, \( \tau \leq \varepsilon_X + 1 \) and thus \( O(\log \tau) \leq O(\log \varepsilon_X) \). The time required by \texttt{update_tl_xt} is hence simplified to \( O(\tau \log \varepsilon_X + \log v) \).

The pseudocode for \texttt{add_edge} consists of three primary phases: find the "new" vertices before \( v_h \) (i.e. \( v_i \) and all vertices which come before \( v_i \)), update \( v_h \)'s cross-timeline paths, and update the cross-timeline paths of all vertices which follow \( v_h \). Finding the vertices before \( v_i \) requires an \( O(\log \tau) \) search to find a timeline \( t \) on which \( v_i \) is ordered and, for each of \( t \)'s \( \tau \) potential cross-timeline structures, an \( O(\log \varepsilon_X) \) search on \( xt \) and possible \( O(\log \tau) \) insert into origins.* The ordering of \( v_i \) itself with respect to \( v_h \) is handled with an \( O(\log \tau) \) insert for each timeline on which \( v_t \) is ordered (\( \tau \) possible). Total time is \( O(\tau \log \varepsilon_X) \), using the same \( O(\log \tau) \leq O(\log \varepsilon_X) \) argument as above.

Updating \( v_h \)'s cross-timeline structures involves, for each of \( \tau \) possible timelines \( t \) on which \( v_h \) is ordered, finding \( t \) with an \( O(\log \tau) \) search and applying \texttt{update_tl_xt} to it. Given the above analysis for \texttt{update_tl_xt}, the time cost for this phase is \( O(\tau(\tau \log \varepsilon_X + \log v)) \).

To complete \texttt{add_edge}, the cross-timeline paths of all vertices which follow \( v_h \) must be updated. For each timeline \( t \) in the graph, \texttt{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, \texttt{add_edge} finds the first vertex on \( t \) following \( v_h \) (\( O(\log \varepsilon_X) \)) and applies \texttt{update_tl_xt} when appropriate. Completion of \texttt{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 \texttt{add_edge}, this result implies that \texttt{add_edge} as a whole is of time cost \( O(\tau^2 \log \varepsilon_X + \tau \log v) \).

As for \texttt{add_edge}, \texttt{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 \( v_l_{\text{term}} \) before \( v_e \) on each timeline containing the origin of a path to \( v_e \). There could be \( \tau \) timelines, and the \( v_l_{\text{term}} \) search requires an \( O(\log \varepsilon_X) \)

---

* It is possible to replace the \( O(\log \tau) \) origins set insert with an \( O(1) \) list insert by simultaneously scanning the origin timelines from \( xt \) paths and the timelines on which \( v_i \) is ordered. Since this change would not affect the overall time cost of \texttt{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 $O(\tau \log e_X)$, 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 $v_{l_{\text{org}}}$ on $t$ following $v_s$ ($O(\log e_X)$) and finds $t$'s path itself ($O(\log \tau)$). Finally, the span of vertices on $t$'s own path between $v_{l_{\text{org}}}$ and $v_{l_{\text{term}}}$ is traversed, adding each vertex to $I$ ($O(\nu_I)$; see below). The list-building phase thus requires $O(\tau \log e_X)$ additional locate time and $O(\tau \nu_I)$ copy time. Combined with the first phase of list_interval, this results in an $O(\tau \log e_X)$ 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; $\nu_I$ 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 timelines 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 distributed 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 $O(v)$.

For the first approach, recollect what happens when an edge is added. The tail of the edge, $v_t$, follows vertices on at most $\tau$ timelines, and, after the edge is added, the head $v_h$ must also follow those vertices, origins. A potential of $\tau$ 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 $\tau$ timelines following $v_h$, this line of reasoning implies that $\tau^2$ potential path entries might be added for the new edge. The question is whether or not this implies an $O(\tau^2 e_X)$ 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 $\tau$. This yields a modest $O(\tau e_X + v)$ 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 $e_X$ of these head vertices, and each one may be ordered on at most $\tau$ timelines. This argument again yields an $O(\tau e_X + v)$ 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 $\tau = v$). 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 $\tau$ a constant factor. Additionally, no distinction will be made between $\varepsilon$ and $e_X$. 
Time for ADD\_EDGE in the transitive closure method is $O(v)$ (amortized). For the search tree method, it is $O(\log e)$. The search tree method time is clearly superior. Similarly, LIST\_INTERVAL locate time in the transitive closure method is $O(v)$, verses $O(\log e)$ for the search tree method. Copy time for both is $O(v_l)$.

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 $\Theta(v^2)$ space, while the search tree method takes $O(e)$.

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 also 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 is 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..v_limit] of wv_vertex;         // any O(1) access time structure
    E : array [0..e_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 \texttt{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.

\begin{verbatim}
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_q : edge_id;           // not used, in this case
begin
   v += 1;
   v_q := v;
   V[v_q] := (new_V, id_null);
   for each t \in T_on do
      add_edge(T[t].self \rightarrow \text{last}() \rightarrow \text{vid}, v_q, e_q);
   endfor;
   // Check for each of t, h, s, and e candidacies and add to appropriate enabling sets.
   //
   if s \in candidate_for then
      B_s \cup= \{v_q\};
   endif;
   ...
   return;
end add_vertex;
\end{verbatim}

The wavefront method's \texttt{add_edge} procedure (and thus \texttt{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. \texttt{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 '≤' or '≥', not '='. More importantly, however, those
procedure add_edge (vt, vh : vertex_id; out et : edge_id);
    t : timeline;
    vorg, vterm : vertex_id;
    xt : origin_paths;
    origins : ordering_set := ∅;
begin
    // Add et to E and to edge list at vt
    //
    ε += 1;
    et := ε;
    if this is the first edge with tail vt then
        E[et] := (vertex_link, vt, vh);
    else
        E[et] := (edge_link, V[vt].out, vh);
    endif;
    V[vt].out := et;
    // Find all vertices which are now < vh
    //
    t := any timeline such that vt ∈ V(t);
    for xt ∈ t→xtpaths do
        find the latest vorg < vt on xt’s origin timeline;
        if vorg ≠ v0 then // everything follows v0; ignore it
            origins += (xt→org_tid, version(vorg, 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
        vterm := the earliest vertex on t which follows vh;
        if vterm ≠ id_null then
            update_tl_xt(t, vterm, origins);
        endif;
    endfor;
    return;
end add_edge;

Figure 13. Wavefront Method Add_edge Procedure
procedure disable_candidate (v_c : vertex_id; not_candidate_for : set of candidacy);
    v_c' : 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
        B_s := {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_s and v_c ∉ A_t then
        for each t such that v_c ∈ V(t) do
            v_c' := the next vertex on t which follows 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[v_c];      // find a path p with v_c as terminus
                if p ≠ null then
                    remove p from xt;
                    // 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[v_c'] = null then
                        add a (p¬org, v_c') path to xt;
                    endif;
                endif;
            endfor;
        endfor;
    endif;
return;
end disable_candidate;

Figure 14. Wavefront Method Disable_candidate Procedure
records must not be altered because of the case in which \(v_e\) is the origin of a path with terminus \(v_o\), the end bound of a potential list_interval query. In this case, list_interval must be able to determine exactly where to cease putting vertices from \(v_e\)'s timeline into \(I\). The correct vertex on which to stop is \(v_e\), not \(v_e'\).

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 \(v_\delta\) 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 one version after 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 \(v_\delta\). 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\(\rightarrow\)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.

```plaintext
function list_interval (v_e, v_o : vertex_id) : set of vertex_id;
  I : srt_set of vertex_id := \emptyset;        // avoid duplicates
  v_h, v_l_term : vertex_id;
  e : edge_id;
  t : \^timeline;
  xt : \^origin_paths;
  doing, next : \^wv_ordering;
  todo_set : srt_set of wv_ordering key tid := \emptyset;
  done_set : ordering_set := \emptyset;

begin
  // Find the latest vertex v_l_term < v_o on each timeline with which v_o is ordered.
  //
  t := any timeline such that v_e \in V(t);
  for xt \in t\rightarrow xt\text{paths do}
    if xt\rightarrow org\_tid \neq t\rightarrow id then
      find the latest v_l_term < v_o on xt's origin timeline;
    else
      v_l_term is v_e itself;                         // this will lead to putting v_e in I
    endif;
end
```
if \( v_{t_{\text{term}}} \neq v_0 \) then

  // Add the vertex on \( T[xt \rightarrow \text{org}_{\text{tid}}] \) just after \( v_{t_{\text{term}}} \) to \( \text{done}_\text{set} \).
  //
  done_set += \( \langle xt \rightarrow \text{org}_{\text{tid}}, 1 + \text{version}(v_{t_{\text{term}}}, xt \rightarrow \text{org}_{\text{tid}}) \rangle \);

  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 \( v_s \leq v_e \) then
  \( t := \text{any timeline such that } v_s \in V(t) \);
  todo_set += \( \langle v_s, t \rightarrow \text{id}, \text{version}(v_s, t) \rangle \); // start todo_set with \( v_s \)

  while todo_set \( \neq \emptyset \) 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_{t_{\text{term}}} := \text{get_vertex(done_set[doing \rightarrow \text{tid}])};
    done_set += \( \langle \text{doing} \rightarrow \text{tid}, \text{doing} \rightarrow \text{ver} \rangle \);

    while doing \rightarrow \text{vid} < v_{t_{\text{term}}} do
      \( I := \text{doing} \rightarrow \text{vid} \);

      // Find where vertex ‘doing’ leads.
      //
      next := \text{end of span}; // in case doing is the terminus of its timeline
      for \( e \in V[\text{doing} \rightarrow \text{vid}] \) .out do // every edge whose tail is \( V[\text{doing} \rightarrow \text{vid}] \)
        \( v_h := E[e].\text{head} \);
        for each \( t \) such that \( v_h \in V(t) \) do
          if \( t \rightarrow \text{id} = \text{doing} \rightarrow \text{tid} \) then
            next := \( \langle v_h, t \rightarrow \text{id}, \text{version}(v_h, t) \rangle \); // 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 \( v_h < v_e \) and if \( v_h \not\in I \) and if \( v_h < \text{todo_set}[t] \rightarrow \text{vid} \) then
              todo_set += \( \langle v_h, t \rightarrow \text{id}, \text{version}(v_h, t) \rangle \);
            endif;
          endif;
        endfor;
      endfor;
    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 \( v_W \) = the number of vertices which are either e or t candidates. It is much more difficult to calculate \( e_W \), 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 \( e_X \); with the wavefront method's reduction of perhaps several vertices' cross-timeline paths into that of the following e- or t-candidate vertex, \( e_W \leq e_X \).

The \( e_W \) 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 \( v_{X \cap W} \leq e_W \leq e_X \).

The add_edge time for the wavefront method is derived effectively the same as for the search tree method and is \( O(\tau^2 \log e_W + \tau \log v_W) \). Examination of the disable_candidate pseudocode reveals this same time complexity. A vertex \( v_c \) may be on \( \tau \) timelines, each ordered with \( \tau \) others. Updating a path record takes \( O(\log e_W) \) 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 \( O(\tau \log e_W) \). 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 \( e_t \)), there is at least an \( O(\log \tau) \) search and perhaps \( \tau \) \( O(\log \tau) \) updates of todo_set, one for each timeline on which the edge's head is ordered.

Total locate time for list_interval is therefore \( O(\tau \log e_W + \tau e_t \log \tau) \), while copy time is \( O(v_I(\tau + \log \tau)) \), or just \( O(\tau v_I) \). As with the search tree method, the \( \tau \) factors in the \( O(\tau e_t \log \tau) \) 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 $O(\tau e_w + v_w)$. 
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 $b$, $\text{top}(a) < \text{top}(b)$ if $a < b$. 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 $O(e)$ 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 $v_{l-term}$ is encountered such that $\text{top}(v_{l-term}) \geq \text{top}(v_e)$. 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 $v_i \Rightarrow v_e$, but not that $v_i < v_e$ ($v_s < v_{l-term} \Rightarrow v_e$).

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 $v_{l-org}$ is encountered such that $\text{top}(v_{l-org}) \leq \text{top}(v_s)$. In other words, when $v_{l-org}$ can no longer follow $v_s$ and thus can not be in the interval ($v_{l-org} \Rightarrow v_s$). 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 \( v_f \) which must follow the start bound \( v_s \) and precede the end bound \( v_e \) : \( v_s \leq v_f \leq v_e \). In many situations, however, it may be desirable to know those vertices which could follow \( v_s \) and precede \( v_e \) : \( v_s \prec v_f \land v_f \prec v_e \). 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

\[
\text{Statement; } \\
\{\text{Statement;}\} \\
\]

Conditional

\[
\text{If condition then } \\
\text{Sequence; } \\
\text{else} \\
\text{Sequence; } \\
\text{endif; } \\
\]

Alternative

\[
\text{Case expression of } \\
\text{Value list: } \\
\text{(Sequence;)} \\
\ldots \\
\text{others: } \\
\text{(Sequence;)} \\
\text{Endcase; } \\
\]

Iteration

\[
\text{For variable in range do } \\
\text{Sequence; } \\
\text{Endfor; } \\
\]

Repetition, Test At Entry

\[
\text{While condition do } \\
\text{Sequence; } \\
\text{Endwhile; } \\
\]

Repetition, Test At Exit

\[
\text{Repeat } \\
\text{Sequence; } \\
\text{Until condition; } \\
\]

### 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 **not** 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**
  
  \[
  \text{subrange\_type} = \\
  \text{range \{first, last\}} \\
  \text{of base\_type;}
  \]

- **Enumeration**
  
  \[
  \text{enumeration\_type} = \\
  (value{, value});
  \]

- **Array**
  
  \[
  \text{array\_type} = \\
  \text{array \{range\{, range\}} \\
  \text{of base\_type;}
  \]
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 \( \cup, \cap, \cdot, \subseteq, \supseteq \), \{ element, element \} and \( \in \), respectively.

- **declaration:** \( \text{type\_name} = \text{set of base\_type}; \)
- **operators:** \( \cup, \cap, \cdot, \subseteq, \supseteq, \in \), and the assignment operators \( \cup=, \cap=, \cdot= \)
- **constants:** \( \emptyset \) — 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 \( \&, [\ \text{element}, \text{element}], \text{list(element\_number)}, \) and \( \text{list(element\_range)} \), respectively.

- **declaration:** \( \text{type\_name} = \text{list of base\_type}; \)
- **operators:** \( \&, (\text{element\_number}), [\text{element\_range}] \), 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**: `Ø` — 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 \( G = (V, E) \) 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 \( G \). Edges are added and paths reported in \( O(v) \) amortized time per operation, \( v = |V| \); the existence of a path may be checked in \( O(1) \) (constant) time. The data structures require \( \Theta(v^2) \) space.

```plaintext

constants

\( \text{v\_limit} : \text{integer} := \text{some large positive number} \quad \// \quad \text{greatest \# of elements} \\

```

```plaintext
types

\( \text{vertex\_id} = \text{range} \ [0..\text{v\_limit}] \text{ of integer} \quad \// \quad \text{used as indices, not just as ids} \\

\( \text{Ital\_node} = \text{record} \\
\quad \text{key} : \text{vertex\_id}; \\
\quad \text{parent} : \text{^Ital\_node}; \\
\quad \text{child} : \text{^Ital\_node}; \\
\quad \text{sibling} : \text{^Ital\_node}; \\
\quad \text{end Ital\_node; } \\

```

```plaintext
globals

\( \// \quad \text{index}[v_i, v_j] \neq \text{null} \rightarrow \text{a path exists from } v_i \text{ to } v_j \)
\( \// \quad \text{If the path exists, this points to } v_j \text{ in the descendent tree} \)
\( \// \quad \text{of } v_i. \)
\( \// \\
\quad \text{index} : \text{array [vertex\_id, vertex\_id] of ^Ital\_node} := \text{null}; \\

\( \// \quad \text{Trees of all descendants of each vertex in the graph} \)
\( \// \\
\quad \text{desc} : \text{array [vertex\_id] of ^Ital\_node;} \\
```

procedure Ital_initialize();
    \( v_p, v_j \) : vertex_id;
begin
    for \( v_i \) in [0..v_limit] do
        desc[\( v_i \)] := new(Ital_node);
        desc[\( v_i \)]^ := (\( v_p \), 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 (\( v_{\text{org}}, v_{\text{term}} \) : vertex_id) : Boolean;
begin
    return index[\( v_{\text{org}}, v_{\text{term}} \)] \neq null;
end Ital_check_path;

function Ital_get_path (\( v_{\text{org}}, v_{\text{term}} \) : vertex_id) : list of vertex_id;
    \( p \) : list of vertex_id := [ ]; // path from \( v_{\text{org}} \) to \( v_{\text{term}} \)
    curr_vertex : Ital_node;
begin
    if index[\( v_{\text{org}}, v_{\text{term}} \)] \neq null then // \( v_{\text{term}} \) is reachable from \( v_{\text{org}} \)
        curr_vertex := index[\( v_{\text{org}}, v_{\text{term}} \)]; // locate terminus in desc[\( v_{\text{org}} \)]
        \( p \) := [\( v_{\text{term}} \)];
        repeat // go up in desc[\( v_{\text{org}} \)]
            curr_vertex := curr_vertex\rightarrow parent;
            \( p \) := [curr_vertex\rightarrow key] & \( p \); // prepend vertex to path (&= appends)
            until curr_vertex\rightarrow parent = null; // ... until we reach \( v_{\text{org}} \)
        endif;
    return \( p \);
end Ital_get_path;
procedure Ital_add_edge (ν₁, ν₂ : vertex_id);
    ν₀ : vertex_id; // some vertex < ν₁
begin
    if index[ν₁, ν₂] = null then // no path already recorded from ν₁ to ν₂
        for ν₀ in [0..ν₁_limit] do
            if index[ν₀, ν₁] ≠ null and index[ν₀, ν₂] = null then
                // The edge <ν₁, ν₂> gives rise to a new path from ν₀ to ν₂
                //
                meld(ν₀, ν₁, ν₂, ν₂); // update desc[ν₀] by means of desc[ν₂]
            endif;
        endfor;
    endif;
    return;
end Ital_add_edge;

// Merge desc[ν₀] with a pruned subtree of desc[ν₁ meld] rooted at ν₁ meld.
// The vertex of desc[ν₀] to which the pruned subtree will be grafted is ν₀ link. By
// "pruning," we mean removing those vertices in desc[ν₁ meld] which are already in desc[ν₀].
//
procedure meld(ν₀, ν₁ meld, ν₀ link, ν₁ meld : vertex_id);
    parent, child : Ital_node;
begin
    // Insert the root of ν₁ meld into desc[ν₀] as a child of ν₀ link
    //
    if ν₀ = ν₀ link then // index does not contain self-loops
        parent := desc[ν₀ link];
    else
        parent := index[ν₀, ν₀ link];
    endif;

    index[ν₀, ν₁ meld] := new(Ital_node);
    index[ν₀, ν₁ meld].addChild(ν₁ meld, parent, null, parent.child);
    for each child of ν₁ meld in desc[ν₁ meld] do // find child, then follow siblings
        // If the child and its subtree are not already in desc[ν₀], add them
        //
        if index[ν₀, child → key] = null then
            meld(ν₀ meld, ν₁ 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

\[ \text{v\_limit, e\_limit : integer := some large positive number} \quad \text{// greatest # of elements} \]
\[ \text{id\_null : integer := -1;} \quad \text{// "no such object"} \]

types

\[ \text{natural = range [0..] of integer;} \]
\[ \text{vertex\_id = range [id\_null..v\_limit] of integer;} \]
\[ \text{edge\_id = range [id\_null..e\_limit] of integer;} \]
\[ \text{timeline\_id = range [id\_null..] of integer;} \]
\[ \text{version\_index = natural;} \]

ordering = record
\[ \text{tid : timeline\_id;} \]
\[ \text{ver : version\_index;} \]
\[ \text{end ordering;} \]

ordering\_set = srt\_set of ordering key tid;

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

edge = record
\[ \text{tail, head : vertex\_id;} \]
\[ \text{end edge;} \]

// 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
\[ \text{case (cross\_timeline, in\_timeline) of} \]
\[ \text{\quad cross\_timeline : (org : version\_index);} \]
\[ \text{\quad in\_timeline : (vid : vertex\_id);} \]
\[ \text{endcase;} \]
\[ \text{term : version\_index;} \]
\[ \text{end x\_tl\_path;} \]
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 xpaths[id]
  xpaths : srt_set of origin_paths key org_tid;
end timeline;

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

procedure add_vertex (new_V : vertex; Tan : set of timeline_id;
  out v_q : vertex_id);
  sorted_Ton : srt_set of timeline_id; // so that the vertex's timelines can later be
    // referenced in order
  t : timeline_id;
  e_t : edge_id; // not used, in this case
begin
  sorted_Ton := make_srt_set(Ton);
  v := 1;
  v_q := v;
  V[v_q] := new_V; // store application-specific fields
  for t in sorted_Ton do
    V[v_q].on &:= (t, T[t].self->last()->ver);
    add_edge(T[t].self->last()->vid, v_q, e_t);
  endfor;
  return;
end add_vertex;
procedure update_t1_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_t1_path;
origin : ^ordering;

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

if xt = null then
  t→xtpaths += (origin→tid, Ø);
  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_t1_xt;
procedure add_edge (νₜ, νₜʰ : vertex_id; out eₑ : edge_id);
  t : ^timeline;
  tidₜʰ, tid_on : timeline_id;
  ver₀, verₑ, verᵣ, verₜʰ, ver₀ʰ : version_index;
  xt : ^origin_paths;
  ord : ^ordering;
  origins : list of ordering := [ ];

begin
  ε += 1;
  eₑ := ε;
  E[eₑ] := (νₜ, νₜʰ);

  // Check if νₜ = ν₀. If so, only want to cross-reference each timeline on which νₜʰ
  // is ordered with each other such timeline, not with all the timelines in the graph (ν₀
  // 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 xtopaths.
  //
  if νₜ ≠ ν₀ then
    // Find all vertices which are now < νₜʰ. This is νₜ and those vertices < νₜ.
    //
    tidₜʰ := V[νₜ].on(0)→tid; // Find any timeline on which νₜ is ordered. The
    ver₀ := V[νₜ].on(0)→ver; // first such timeline is used because we must
    t := T[tidₜʰ]; // scan V[νₜ].on from the beginning, anyway.
    verₜ := ver₀;
    for xt ∈ t→xtopaths do // scanned in increasing org_tid sequence
      if t→id ≠ tidₜʰ then
        // find the latest ν₀ < νₜ on xt’s origin timeline
        //
        ver₀ := xt→path(≤, verₜ)→org;
        if ver₀ ≠ 0 then // everything follows ν₀; ignore it
          origins &:= (xt→org_tid, ver₀);
        endif;
      else
        // We want νₜ’s version itself, not that of the vertex before νₜ
        //
        origins &:= (tidₜʰ, ver₀);
        ord := V[νₜ].on.next(0); // next timeline on which νₜ is ordered
        if ord ≠ null then
          tidₜʰ := ord→tid;
          ver₀ := ord→ver;
        else
          tidₜʰ := id_null;
        endif;
      endif;
    endfor;
else
    for ord ∈ \( V[ν_h] \).on do
        origins &\( = \langle \text{ord} \rightarrow \text{tid}, 0 \rangle \); 
    endfor;
endif;

// Update \( ν_h \) to follow origins
//
for ord ∈ \( V[ν_h] \).on do
    update_tξ(\( T[\text{ord} \rightarrow \text{tid}] \), ν_h, origins, ord→ver);
endfor;

// Update all vertices which follow \( ν_h \) to follow origins
//
νh := \( V[ν_h] \).on(0→ver); \hspace{1cm} // a reference point for comparisons against \( ν_h \)
tid_h := \( \text{on}(0 \rightarrow \text{tid}) \); \hspace{1cm} // \( T[\text{tid}_h] \) is a reference point
for \( t \in T \), \( t \rightarrow \text{id} \neq \text{tid}_h \) do
    xt := \( T[\text{xtaths}(\text{tid}_h)] \); \hspace{1cm} // is any vertex on \( t \) ordered with \( T[\text{tid}_h] \)?
    if xt ≠ null and xt→path.org(\( \geq \), νh) ≠ null then
        // find the earliest vertex on \( t \) which follows \( ν_h \)
        //
        ver_term := xt→path.org(\( \geq \), νh)→term;
        update_tξ(\( t, \text{id}_{null}, \text{origins}, \text{ver}_\text{term} \)); \hspace{1cm} // \( ν_{\text{term}} \) unnecessary here
    endif;
endfor;
return;
end add_edge:
function list_interval (v_e, v_s : vertex_id) : set of vertex_id;
    I : set of vertex_id := ∅; // avoid duplicates
    I_terms : list of ordering := [ ]; // termini of all spans of vertices making up I
    I_term : ^ordering;
    ver_e, ver_s, ver_I_term : version_index;
    p, p_I_term : ^x_til_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
            // 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 and if 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;
        endif;
    endfor;
end
else
    if \( v_{g} < I_{\text{term}} \rightarrow \text{ver} \) then
        for each \( p \in x_{t} \rightarrow \text{path} \)
            with \( p \rightarrow \text{term} \in [v_{g}, I_{\text{term}} \rightarrow \text{ver}] \) do
                \( I += p \rightarrow \text{vid} \);
        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
\begin{align*}
v\_limit, e\_limit & : \text{integer} := \text{some large positive number} \quad \text{// greatest # of elements} \\
id\_null & : \text{integer} := -1; \quad \text{// "no such object"}
\end{align*}

\textbf{types}
\begin{align*}
natural & = \quad \text{range} \ [0..] \ \text{of integer}; \\
vertex\_id & = \quad \text{range} \ [id\_null..v\_limit] \ \text{of integer}; \\
edge\_id & = \quad \text{range} \ [id\_null..e\_limit] \ \text{of integer}; \\
timeline\_id & = \quad \text{range} \ [id\_null..] \ \text{of integer}; \\
version\_index & = \natural;
\end{align*}
\begin{align*}
\text{ordering} & = \text{record} \quad \text{// version (order) of a vertex on a timeline} \\
& \quad \text{tid} : \ timeline\_id; \\
& \quad \text{ver} : \ version\_index; \\
\text{end \ ordering;} \\
\text{ordering\_set} & = \text{srt\_set of ordering key tid;}
\end{align*}
\begin{align*}
\text{wv\_ordering} & = \text{record} \quad \text{// edge tail or head, interval start or end} \\
& \quad \text{vid} : \ vertex\_id; \\
& \quad \text{tid} : \ timeline\_id; \\
& \quad \text{ver} : \ version\_index; \\
\text{end \ wv\_ordering;} \\
\text{candidacy} & = (t, h, s, e); \\
\text{wv\_vertex} & = \text{record} \\
& \quad \text{on} : \ \text{list of ordering;} \quad \text{// though a list, this is sorted by tid} \\
& \quad \text{out} : \ \text{edge\_id;} \\
& \quad \text{// ... and whatever an implementation needs to keep track of} \\
\text{end \ wv\_vertex;} \\
\text{next\_edge} & = (\text{edge\_link}, \ \text{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;
sel : ~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
v : natural := 0;                                // current number of vertices
E : array [0..e_limit] of wv_edge;              // any O(1) access time structure
e : 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);
    v += 1;
    v_q := v;
    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;
    endwhile;
end;
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 and (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_t : edge_id);
  t : ^timeline;
  tid_t, 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_t := ε;
  if V[v_t].out = id_null then
    E[e_t] := (vertex_link, v_t, v_h);
  else
    E[e_t] := (edge_link, V[v_t].out, v_h);
  endif;
  V[v_t].out := e_t;
  // 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 xpaths.
  //
  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;
    ver_on := V[v_t].on(0)->ver;
    t := T[tid_on];
    ver_t := ver_on;
  endif;
end add_edge;
for $xt \in t\to xpaths$ do // scanned in increasing org_tid sequence
  if $t\to id \neq tid_{on}$ then
    // find the latest $v_{org} < v_t$ on $xt$'s origin timeline
    //
    $ver_{org} := xt\to path(\leq, ver)\to org$;
    if $ver_{org} \neq 0$ then // everything follows $v_0$; ignore it
      origins &= (xt$\to$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 $\neq$ null then
      tid$_{on}$ := ord$\to$tid;
      ver$_{on}$ := ord$\to$ver;
    else
      tid$_{on}$ := id_null;
    endif;
  endif;
endfor;
else
  for ord $\in V[v_h].on$ do
    origins &= (ord$\to$tid, 0);
  endfor;
endif;

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

// Update all vertices which follow $v_h$ to follow origins
//
tid$_{h} := V[v_h].on$(0)$\to$tid; // a reference point for comparisons against $v_h$
ver$_{h} := V[v_h].on$(0)$\to$ver;
for $t \in T$ do
  if $t \neq$ null and $t\to$ path.org($\geq$, ver$_h$) $\neq$ null then
    // find the earliest vertex on $t$ which follows $v_h$
    //
    ver$_{term} := t\to$ path.org($\geq$, ver$_h$)$\to$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_o, ver' : 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 \in not_candidate_for then
      \(A_t := \{v_c\}\);
   endif;
   if h \in not_candidate_for then
      \(A_h := \{v_c\}\);
   endif;
   if s \in not_candidate_for then
      \(B_s := \{v_c\}\);
   endif;
   if e \in 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 \(v_c \in B_e\) and \(v_c \notin A_t\) then
      for ord \in V(\{v_c\}) do
         \(t := T[ord\rightarrow id];\)
         ver_o := ord\rightarrow ver;
         ver' := t\rightarrow self\rightarrow path(>, \{ver\}\rightarrow term); // next vertex on i 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 \in t\rightarrow xtlpaths do
            \(p := xt\rightarrow path[ver_o];\) // find a path \(p\) with \(v_c\) as terminus
            if \(p \neq null\) then
               xt\rightarrow 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\rightarrow path[ver'] = null then
                  xt\rightarrow path += (p\rightarrow org, ver_o);
               endif;
            endif;
         endfor;
      endfor;
   endif;
end disable_candidate;
function list_interval (νₚ, νₑ : vertex_id) : set of vertex_id;
    l : srt_set of vertex_id := ∅;              // avoid duplicates
    νₚ : vertex_id;
    verₚ, verₑ, verₚₚ, verₑₚ : version_index;
    tidₚ, tidₑ : 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 νₑₚ < νₑ on each timeline with which νₑ is
  // ordered.
  //
  t := T[V[νₑ].on(0)→tid];                   // find some timeline on which νₑ is ordered
  verₑ := V[νₑ].on(0)→ver;
  for xt ∈ t→xpaths do
    if xt→org_tid ≠ t→id then
      verₑₚ := xt→path(≤, verₑ)→org;        // latest vertex < νₑ on xt’s origin timeline
    else
      verₑₚ := verₑ;                        // this will lead to putting νₑ in l
    endif;
    if verₑₚ ≠ 0 then
      // Add the vertex on T[xt→org_tid] just after νₑₚ to done_set.
      //
      done_set += (xt→org_tid, verₑₚ + 1);
    endif;
  endfor;

  // Add all vertices between νₚ and νₑ to l, doing one span of a timeline’s vertices
  // at a time.
  //
  tidₚ := V[νₚ].on(0)→tid;                   // find some timeline on which νₚ is ordered
  verₚ := V[νₚ].on(0)→ver;
  if done_set[tidₚ] ≠ null and if done_set[tidₚ]→ver > verₚ then      // if νₚ ≤ νₑ then
    todo_set += (νₚ, tidₚ, verₚ);                  // start todo_set with νₚ
  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 := done_set[doing->tid, doing->ver];
/
while doing->ver < ver_I_term do
/
  l := 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 := (ν_h, tid_h, ver_h); // just keep going along t
    
    else
      / If ν_h should be in I, is not already in I, and we have not
      / already recorded that it should be in I, record ν_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 := todo_set[ν_h, tid_h, ver_h];
        endif;
      endif;
    endif;
  endfor;
endfor;
/
doing := next;
endwhile;
endwhile;
end;
/
return make_set(I); // convert from srt_set to set
end list_interval;
8. Bibliography


