Discrete Mathematics and Theoretical Computer Science **4**, 2001, 323–350

An Efficient Algorithm for the Uniform Maximum Distance Problem on a Chain

Gabrielle Assunta Grün[†]

School of Computing Science, Simon Fraser University, Burnaby BC V5A 1SA

received Oct 10, 2000, revised Apr 27, 2001, accepted Jul 31, 2001.

Efficient algorithms for temporal reasoning are essential in knowledge-based systems. This is central in many areas of Artificial Intelligence including scheduling, planning, plan recognition, and natural language understanding. As such, scalability is a crucial consideration in temporal reasoning. While reasoning in the interval algebra is NP-complete, reasoning in the less expressive point algebra is tractable. In this paper, we explore an extension to the work of Gerevini and Schubert which is based on the point algebra. In their seminal framework, temporal relations are expressed as a directed acyclic graph partitioned into chains and supported by a *metagraph* data structure, where time points or events are represented by vertices, and directed edges are labelled with $<$ or $\leq$. They are interested in fast algorithms for determining the strongest relation between two events. They begin by developing fast algorithms for the case where all points lie on a chain. In this paper, we are interested in a generalization of this, namely we consider the problem of finding the maximum "distance" between two vertices in a *chain*; this problem arises in real world applications such as in process control and crew scheduling. We describe an O(n) time preprocessing algorithm for the maximum distance problem on chains. It allows queries for the maximum number of $<$ edges between two vertices to be answered in O(1) time. This matches the performance of the algorithm of Gerevini and Schubert for determining the strongest relation holding between two vertices in a chain.

Keywords: graph theory, maximum distance problem, temporal reasoning, analysis of algorithms and data structures

1 Introduction

Temporal reasoning plays a vital role in many domains of Artificial Intelligence including planning, plan recognition, natural language understanding, scheduling, and diagnosis of technical systems. However, even when an algorithm for temporal reasoning has reasonable complexity such as linear or quadratic time, it may still be inadequate for large databases. In addition, if all the temporal precedence information is stored in a matrix having O(n^2) space and requiring $\Omega(n^2)$ preprocessing, both the storage and processing are still excessive for large-scale applications. The reality that some temporal reasoning tasks need a large amount of time and space is noted; for example, the best known algorithm for computing closure in the

[†]The work of this author was supported by the Natural Science and Engineering Research Council of Canada. It was supervised by Arvind Gupta and Jim Delgrande.

point algebra takes O(n^2) space and O(n^4) time [GS95]. Thus, research has focused on particular domains for which extremely efficient algorithms might be developed.

This paper has, as a foundation, the development of the work of Gerevini and Schubert [GS95], the latest version of which is the *TimeGraphII* system. The starting point of their technique is on chains, i.e. sets of linearly ordered time points. After O(n) preprocessing, queries on chains can be answered in O(1) time. On the other hand, determining the strongest relation between vertices in different chains is dependent on a *metagraph* that ideally should be much smaller than the original graph. In this method, an arbitrary set of assertions regarding points in time is processed into a directed acyclic graph (DAG) where time points or events are represented by vertices and directed edges are labelled with assertions among time points. The DAG is then decomposed into *chains* of such assertions which are separated from the DAG. If the original graph is dominated by chains, the resulting reasoner will be efficient.

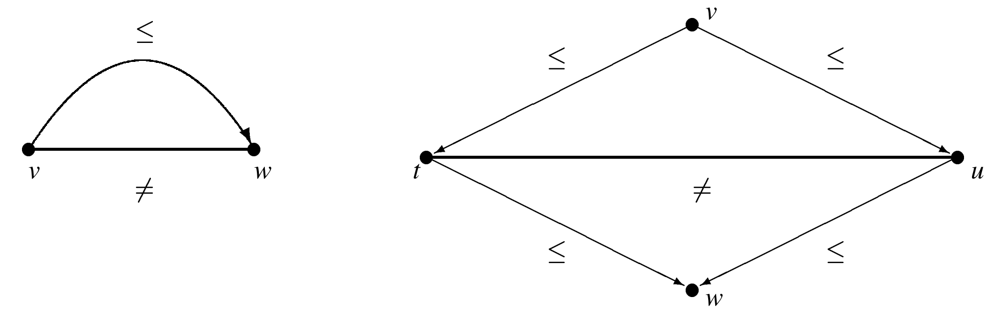
There are times in which it is not merely enough to know that an event precedes another but when it is also useful to bound the number of events (or the amount of time) lying between particular events. We call this the *maximum time separation problem*, or in other words, the *maximum distance problem*. Formally, the problem is to find the longest weighted path between two vertices in a graph. This problem is the same as the LONGEST PATH problem [GJ79] and is NP-complete for general graphs. With regard to directed acyclic graphs or DAGs, the time complexity is $O(|V|+|E|)$ [CLR90]. For our purposes, we restrict ourselves to chains and edges with weights of 0 or 1. The parameters of this problem are that $\leq$ edges have weight 0 and $<$ edges have weight 1, where weights on edges are summed to get distances. Our technique is based on partitioning the chain into discrete regions called *proper edge regions* and checking where the events being queried lie in relation to these regions. We show that after O(n) preprocessing time, queries about the maximum distance between two vertices in a chain can be answered in O(1) time. This is the same performance as the algorithm of Gerevini and Schubert for reasoning within a chain.

A simple real-world example of an application of the maximum distance problem on chains can be found in the education domain. The vertices represent courses, the $<$ edges represent the relation of course prerequisites, and the $\leq$ edges represent the relation of course prerequisites/corequisites. Then, the maximum distance between two vertices denotes the (maximum) number of courses in sequence required before a specific course can be taken. This can be valuable in course planning (in realistic university course requirements, the chains are often quite short though). As well, it is easy to imagine a similar example in the realm of sports or game competitions. Crew scheduling is a further example of where this can be useful. A group of workers is denoted by a vertex. The constraint that one group of workers must start before another group is represented by $<$ edges, and the constraint that one group of workers must start before or at the same time as another group is conveyed by the $\leq$ edges. Another widely applicable example is that of the manufacturing or production of goods and other materials. Here, the vertices represent processes, and the $<$ edges represent the constraint that a process must precede another process. The $\leq$ edges signify the constraint that a process must precede or occur simultaneously to another process. The information obtained from the maximum distance between two vertices can be used in the optimization of resource allocation. While some of these applications may have short chains, it is quite possible that further applications could be found, say from computational biology. As well, an extension of our work may well he incorporated in the more general TimeGraphII framework.

Section 2 on the following page describes related work, and Section 3 on page 327 introduces the formal definitions and problem descriptions. The algorithm for the maximum distance problem is explained in Section 4 on page 329. The formal query algorithm is detailed in Section 5 on page 342 and Section 6 on page 348 describes some extensions to this work. Section 7 on page 349 gives the conclusion and future directions.

2 Related Work

Since the results of this paper are largely based on the work of Gerevini and Schubert [GS95], a summary of their approach is given here. Beginning with arbitrary assertions in the *point algebra*, these assertions are processed to yield a temporally labelled (TL) graph. The vertices of the TL graph represent time points with each vertex having its own identifier. The directed edges are labelled with $<$ and $\leq$, and the undirected edges are labelled with $\neq$ or =. Through the method described below, we can convert the TL graph into a directed graph with only $<$ and $\leq$ edges, none of which are redundant such that there are no explicit $<$ and $\leq$ relations implied by a transitive path.



(a) (b)

Fig. 1: The two kinds of the implicit $<$ relation. Thin lines indicate paths, and thick lines represent $\neq$ edges. In both of the graphs, there is an implicit $<$ relation between v and w.

First, the "=" relations are eliminated by extracting the strongly connected components from the TL graph through an algorithm adapted from depth first search. Each strongly connected component is collapsed into a single vertex and all the identifiers of the vertices that make up the strongly connected component are alternate identifiers for this new vertex. If an edge in a strongly connected component is labelled with $<$ or $\neq$, the graph is inconsistent and the process is halted.

A further concern is that of *implicit* $<$ relations. An *implicit* $<$ relation is present between a pair of vertices when the strongest relation implied by the graph among the pair of vertices is $<$, and no path with at least one $<$ edge exists between the vertices. These relations occur in two forms, one with a $\neq$ edge as well as a path containing only $\leq$ edges between the pair of vertices (see Figure 1 (a)). The other form, a $\neq$ diamond, has two separate paths containing only $\leq$ edges between the pair of vertices through two different intermediate vertices that are connected by a $\neq$ edge (see Figure 1 (b)). The implicit relations are efficiently identified and made *explicit* by adding $<$ edges between the pairs of vertices involved. As well, the redundant $\neq$ relations from the implicit $<$ relations are removed. This step can be the most expensive of the whole preprocessing, time-wise. However, the time taken is minimized by using the metagraph structure (below) and for the second form of implicit $<$ relations, by only looking for the smallest $\neq$ diamonds. To state it differently, given a $\neq$ edge involving a pair of vertices, a search is made for their nearest common ancestor and nearest common descendant. The resulting ($<$, $\leq$)-graph is then further processed into structures designed for efficient reasoning.

In the latest form of the system of Gerevini and Schubert, the focus of temporal reasoning is on *chains* of events, where a chain is a path of $\leq$ edges with possible transitive edges linking pairs of vertices on the $\leq$ path. From the $(<, \leq)$-graph, a timegraph is created, which is the $(<, \leq)$-graph partitioned into a set of time chains such that each vertex is on precisely one chain. The timegraph has a unique source or start time, and unique sink or end time. This allows each vertex v of the timegraph to be given a *pseudotime* consisting of the length of the longest path from the source to v, i.e. the $\leq$ rank of v, multiplied by an increment. The pseudotimes are computed by a slight adaptation of the DAG longest path algorithm. Vertices within a chain can have a *nextgreater* link, an edge connecting a vertex to the closest vertex known to be strictly greater than the specified vertex based on the edge labels. It takes linear time to compute pseudotimes, and to compute the nextgreater links within a chain.

The supporting metagraph is composed of *cross-edges* that join different chains, the endpoints of which are called *metavertices*. As well, each metavertex has two extra edges associated with it, namely, the *nextin* edge that connects the metavertex to the closest vertex on the same chain with an incoming cross-edge, and the *nextout* edge that connects the metavertex to the closest vertex on the same chain with an outgoing cross-edge. The metagraph, which includes the nextin, nextout and nextgreater edges, can be computed in linear time. It expresses information represented in the original graph not related by the chains.

If it is assumed that the timegraph is dominated by chains of events, the metagraph is anticipated to be much smaller than the original. This leads to efficient reasoning algorithms, given that reasoning within a chain takes constant time. The five cases in which computing the strongest relation entailed by the timegraph between two time points takes constant time are now described. If the identifiers of two points are alternate names of the same vertex, the relationship between them is equality. The $<$ relations are identified by checking if the pseudotime of the head of the nextgreater link of the smaller vertex (with respect to pseudotimes) is less than or equal to the pseudotime of the larger vertex. Otherwise, if the pseudotime of one vertex is less than the pseudotime of another, a $\leq$ relation exists between these vertices. If two points having the same pseudotime are on different chains and there is no $\neq$ edge between them, the relation between them is $\{=, <, >\}$. If there is a $\neq$ edge between the vertices, the relation between the vertices is $\neq$, provided that all implicit relations have been made *explicit*. However, to reason about points in different chains, a standard search of the metagraph that takes $O(\hat{e})$ time is needed, where $\hat{e}$ is the number of edges in the metagraph. Gerevini and Schubert [GS95] also discusses point algebra disjunctions, which is independent of the timegraph and not of interest here. We will also not be further concerned with the metagraph in this paper.

Other structures and methods for temporal reasoning have been tried as well. Notably, Ghallab and Mounir Alaoui [MA89] use a lattice of time points undergirded by a maximum spanning tree to attain an efficient indexing. The system is claimed to be both sound and complete in dealing with the SIA (a restricted form of the interval algebra comparable to the point algebra) by Ghallab and Mounir Alaoui [MA89]. However, it was later shown to be incomplete for $\neq$ relations. Its performance for updating and retrieving a set of temporal relations is linear with a small constant on average.

Dorn [Dor92] uses sequence graphs to reduce the time and space required by a variable but significant amount for temporal reasoning in technical processes such as monitoring, diagnosis, planning and scheduling in expert systems. A sequence graph is made up of at least one sequence chain and other intervals that are only loosely attached to chains. Sequence chains are based on the observation that events in technical domains frequently occur one after another. In addition, execution of the processes is often uninterrupted for a long period of time. Only "intermediate" relations are stored, yet the techniques that are used allow no loss of information. The approach of Dorn [Dor92] is interval based.

Furthermore, Delgrande and Gupta [DG96] give an O(n) preprocessing algorithm that permits arbitrary $<$ and $\leq$ queries about events in the point algebra to be answered in O(1) time in the class of series parallel graphs. Series parallel graphs have been used to model process execution in addition to varied planning and scheduling applications. The work of Van Allen et al. [VADG98] is an extension of the technique of Delgrande and Gupta [DG96] for series-parallel graphs embedded in general graphs. It achieves a similar performance to the methods of Gerevini and Schubert in which chains are the main components of consideration as opposed to series parallel graphs.

3 Preliminaries

A *directed graph* G is a pair (V,E), where V is a set of vertices and E is a set of edges, $E \subseteq V \times V$. The graphs that we use are all finite and simple, that is, there are no self loops. For a directed edge e=(v_i, v_j), v_i is the tail of e and v_j is the head of e. A path from v_i to v_j of length k in a graph G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$, such that $v_i = v_1$, $v_j = v_k$ and $(v_{\ell-1}, v_\ell) \in E$ for ℓ=2 to k. The path contains the vertices $(v_1, v_2, \ldots, v_k)$ and the edges (v_1, v_2), $(v_2, v_3), \ldots,$ (v_{k-1}, v_k). A path from v_i to v_j is a cycle if v_i $=v_j$. A *directed acyclic graph* (DAG) is a directed graph with no directed cycles. In a DAG, if there is a path from v_i to v_j, we say that v_i is an ancestor of v_j and v_j is a descendant of v_i. See Cormen et al. [CLR90] for more details.

A timegraph that is based on a single chain is a DAG $\mathcal{G}$= ($\mathcal{V}$, $\mathcal{E}$), where the vertex set $\mathcal{V}$ represents a set of time events $(v_1, v_2, \ldots, v_n)$ occurring along a time line, and the edge set $\mathcal{E}$ denotes $<$ and $\leq$ relations among time events [DG98, GS95]. For a vertex v_j, j is referred to as the rank of v_j and v_j is identified and referred to as j. The time that the event denoted by v_i happened is represented by $t(v_i)$. There are edges (v_i, v_{i+1}) labelled with $\leq$, for i=1 to $n-1$. The set of edges labelled with $\leq$ is called $E_{\leq}$; $E_{\leq}$ together with $\mathcal{V}$ corresponds to a chain within the timegraph. As well, there are two distinguished events, namely the source, v_1, and the sink, v_n, such that $t(v_1) \leq t(v_i) \leq t(v_n)$, for all $v_i \in \mathcal{V}$. All vertices except the source and the sink have one outgoing $\leq$ edge and one incoming $\leq$ edge. The set of edges that is labelled with $<$ is referred to as $E_<$, such that if (v_i, v_j) $\in E_<$, then $i < j$. This is interpreted to mean that event v_i happens strictly before event v_j, i.e. $t(v_i) < t(v_j)$. $\mathcal{E}$ is the union of $E_{\leq}$ and $E_<$.

We assume that there are no $<$ redundant edges. As such, the *covering assumption* states that a distinct $<$ edge (a,b) cannot exist when there is a $<$ edge (c,d) that subsumes (a,b) such that $a \leq c < d \leq b$ (refer to Figure 2). From this, it can be seen that all vertices except the source and the sink have at most one outgoing $<$ edge and at most one incoming $<$ edge. The reason for this is that if two different edges terminate at the same vertex, they must start at different vertices and so one completely encloses the other, and vice versa.

Formally, the central problem of interest is the following:

Name: MAX DIST

Instance: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a time graph based on a single chain, which is a DAG and satisfies the covering assumption.

Problem: Find a representation of $\mathcal{G}$ that allows the length of the longest path (maximum distance) between any two vertices a, $b \in \mathcal{V}$ ($a < b$) to be computed by a constant time procedure. Here, maximum distance is measured by assigning $\leq$ edges weight 0 and $<$ edges weight 1.

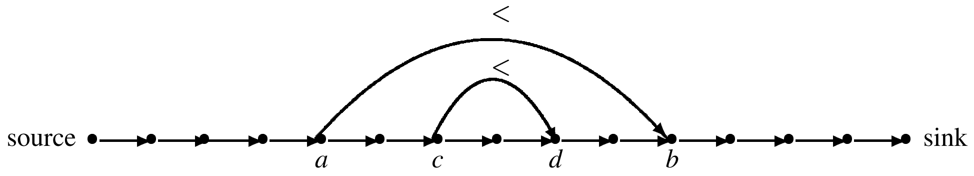


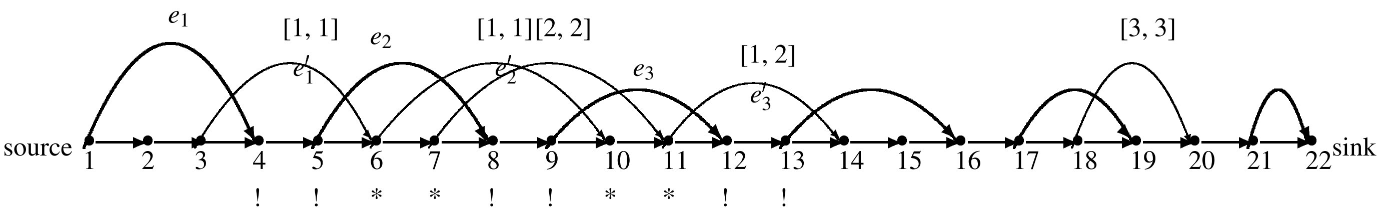
Fig. 2: Edge (c,d) subsumes (a,b) making (a,b) redundant.

The maximum distance between two vertices a and b, is interpreted to be the maximum number of $<$ edges in a path between the vertices and is referred to as $\mathit{distance}(b,a)$. The statement that a vertex a is of distance d from another vertex b is interpreted to mean that time event b occurs at least d time units after time event a. If time is discretized, it can also be interpreted to mean that at least $d-1$ time events occur between a and b. For every vertex $v \in \mathcal{V}$, the maximum distance from the source to v is known as $\mathit{sourceDistance}(v)$. The expression $\mathit{sourceDistance}(b,a)$ is $\mathit{sourceDistance}(b) - \mathit{sourceDistance}(a)$ and is known as the difference distance between a and b.

3.1 Definitions associated with the Actual Algorithm

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a chain and let v be a vertex in $\mathcal{V}$. Then, the closest vertex on $\mathcal{G}$ that is known to be strictly greater than v is referred to as $\mathit{nextGreater}(v)$ [GS95]. If such a vertex does not exist, $\mathit{nextGreater}(v)$ is ∞. As well, the closest vertex on $\mathcal{G}$ that is known to be strictly less than v is called $\mathit{previousLesser}(v)$. If such a vertex does not exist, $\mathit{previousLesser}(v)$ is $-\infty$. The first $<$ edge in the *nextGreater traversal* from vertex a to vertex b is the $<$-edge that determines the *nextGreater* of vertex a which we will call (v_i, v_{i+1}). The next $<$ edge in the traversal which is denoted by (v_{i+2}, v_{i+3}) is the edge that determines the *nextGreater* vertex of v_{i+1}, i.e. $\mathit{nextGreater}(v_{i+1}) = v_{i+3}$. The last $<$ edge in the sequence of $<$ edges is (v_{j-1}, v_j), the edge that determines the *previousLesser* of vertex b. Zero or more intervening $\leq$ edges may lie in between the $<$-edges, and as such, v_i could be a and v_j could be b. In Figure 3, the *nextGreater* traversal from vertex 2 to vertex 20 includes the $<$ edges $(3,6)$, $(6,10)$, $(11,14)$ and $(17,19)$. Formally, the *nextGreater* traversal from vertex a to vertex b, $a < b$, $\mathit{ngTraversal}(a,b)$ is a path from a to b in $\mathcal{G}$ that contains the $<$ edges $(v_i, v_{i+1}), (v_{i+2}, v_{i+3}), \ldots, (v_{j-1}, v_j)$ where $v_{i+1} = \mathit{nextGreater}(a)$, $v_{i+2k+1} = \mathit{nextGreater}(v_{i+2k-1})$, $1 \leq k \leq \frac{j-i-1}{2}$ and $v_{j-1} = \mathit{previousLesser}(b)$. Then, the number of $<$ edges in the *nextGreater* traversal from a to b is denoted by $\|\mathit{ngTraversal}(a,b)\|$. Observe that the difference distance between a and b can be expressed as $\|\mathit{ngTraversal}(\mathit{source},b)\| - \|\mathit{ngTraversal}(\mathit{source},a)\|$. The *path induced by the difference distance* can be interpreted as $\mathit{ngTraversal}(\mathit{source},b) - \mathit{ngTraversal}(\mathit{source},a)$ in which "$-$" refers to the set difference of sets of edges. The number of $<$ edges in this path is equal to $\mathit{sourceDistance}(b,a)$.

A $<$ edge (u,v) is a *proper edge* if $(u,v) \in \mathit{ngTraversal}(\mathit{source},\mathit{sink})$. A $<$ edge that is not a proper edge is a *non-proper edge*. For a $<$ edge (v_i, v_j), the associated *region* is the subset $\mathcal{V}' \subseteq \mathcal{V}$ connected with the chain $\mathcal{G}$ where $\mathcal{V}' = \{v_{i+1}, v_{i+2}, \ldots, v_{j-1}\}$. We can speak of the outgoing $<$ edges or incoming $<$ edges of a region if their tail or head is one of the vertices of the region, respectively, and the other vertex is outside the region. The region associated with a proper edge is known as a *proper edge region*. The chain is divided into p disjoint proper edge regions, $R_1, R_2, \ldots, R_p$ in ascending order of vertex indices, where p is the number of proper edges.



The proper edges are bold and have a label of [0,0]. The $<$ edges of *nonProperPath*(a) are marked as e'_1, e'_2, e'_3 when a is 2 or 3. The corresponding proper edges are e_1, e_2, e_3. Now, the vertices v that satisfy $head(e_j) \leq v \leq head(e'_j) - 1$, for some j are marked with a "!". The vertices v that satisfy $head(e'_j) \leq v \leq head(e_{j+1}) - 1$, for some j are marked with a "*". The vertex denoted by $terminal(a)$ is 14.

Fig. 3: The different cases for the query algorithm.

4 An O(1) Time Solution for the MAX DIST Problem

In this section, we present an O(1) time query algorithm and an O(n) time preprocessing algorithm for the MAX DIST Problem.

4.1 An Overview of the Algorithm

In this section, we describe the main ideas of our algorithm. Suppose we want to compute the maximum distance between two vertices a and b, $a < b$ in a chain $\mathcal{G}$ without preprocessing. The maximum distance from a to b in $\mathcal{G}$ is the number of $<$ edges in the *nextGreater* traversal from a to b, i.e. $||ngTraversal(a,b)||$. This follows from the covering assumption which implies that $<$ edges cannot be contained in each other. It can be seen that simplistically, it would take O(n) time to compute this distance.

The obvious way to preprocess the chain to allow for constant time queries would be to use the difference distance as the maximum distance. However, this only serves as an estimate for $||ngTraversal(a,b)||$, since the maximum distance may be one less than the difference distance. An example of this can be seen in Figure 3 when a is 15 and b is 19. Difference distances can be easily computed using an adaptation of the Gereveni and Schubert algorithm for chains. The fact that the maximum distance is either equal to the difference distance or one less than the difference distance is proven in Corollary 2 in Section 5 on page 342.

1. If *a is outside a proper edge region*, the maximum distance is equal to the difference distance no matter where b is after a. This is because the path induced by the difference distance is exactly $ngTraversal(a,b)$. For an illustration of this case, see Figure 3 with a as 8 and b as 22.
2. If *a and b are in the same proper edge region*, the maximum distance between a and b is zero by the covering assumption. For an example of this case, consider Figure 3 with a as 10 and b as 11.
3. If *a is inside a proper edge region, and b is not in the proper edge region that a is in*, then we must consider two other cases:
   (a) If *there is no $<$ edge e leaving the proper edge region that a is in such that* $tail(e) \geq a$, the maximum distance is equal to one less than the difference distance. This is because the proper

edge of the proper edge region that a is in, i.e. the first $<$ edge in the path induced by the difference distance, is counted in the difference distance when it is not on a path from a to b. Consider Figure 3 with a as 15 and b as 19, for an example of this case.

(b) If *there is a* $<$ *edge e leaving the proper edge region that a is in such that* $\mathit{tail}(e) \geq a$, a short discussion follows before the actual subcases are described. The rest of this section is devoted to characterizing the intricacies of this case.

A *non-proper path* is a maximal *nextGreater* traversal that begins with the tail of a $<$ edge that is inside a proper edge region. It terminates with a $<$ edge (u,v) such that the edge that determines $\mathit{nextGreater}(v)$, $\mathit{nextGreaterEdge}(v)$ is a proper edge or $\mathit{nextGreater}(v) = \infty$. The non-proper path starting immediately at or after a is known as the $\mathit{nonProperPath}(a)$. The vertex where the $\mathit{nonProperPath}(a)$ terminates is known as $\mathit{terminal}(a)$. The path $\mathit{nonProperPath}(a)$ is equivalent to $\mathit{ngTraversal}(a, \mathit{terminal}(a))$. When b is before $\mathit{terminal}(a)$, there are two possible cases. To see these cases, first let the $<$ edges of the $\mathit{nonProperPath}(a)$ be $e'_1, e'_2, \ldots, e'_\ell$. Let $e_1, e_2, \ldots, e_\ell$ be proper edges such that $\mathit{tail}(e_i) < \mathit{tail}(e'_i) < \mathit{head}(e_i)$, for i=1 to ℓ. Refer to Figure 3 for a clear picture of this. The difference distance counts $e_1, e_2, \ldots, e_\ell$ but the maximum distance must count $e'_1, e'_2, \ldots, e'_\ell$. Now, if $\mathit{head}(e_j) \leq b \leq \mathit{head}(e'_j) - 1$ for some j, the maximum distance corresponds to one less than the difference distance. This is because e_j is counted in the difference distance when e'_j has not terminated yet. Otherwise, $\mathit{head}(e'_j) \leq b \leq \mathit{head}(e_{j+1}) - 1$ for some j, and the maximum distance is equal to the difference distance. This is because e_j has been already counted in the difference distance and now the corresponding e'_j has terminated as well.

To differentiate between these two cases, we introduce a labelling scheme on non-proper paths. In particular, an (essentially) unique numeric label is assigned to each distinct non-proper path. One problem with this is that non-proper paths can merge, i.e. the next non-proper edge of at least two distinct non-proper paths is the same (in Figure x3, $\mathit{terminal}(6) = \mathit{terminal}(7) = 14$). When this occurs, we assign all the labels of the paths being merged to be the label of the merged path. In this way, we keep track of where the $\mathit{nonProperPath}(a)$ terminates. This will allow us to differentiate between Case 3(b)i and 3(b)ii of this algorithm, i.e. to determine if b is less than $\mathit{terminal}(a)$. We assign the labels using integers so that the merged non-proper paths are labelled with a contiguous range of numbers. For uniformity, we extend this and write all the labels as a range of integers. The label $[a,b]$ includes all numbers in the range from a to b inclusive. In practice, all the $<$ edges of a non-proper path will be labelled with the label of the path.

While the actual algorithm is detailed in Section 4.3 on page 334, some features of our labelling algorithm are noted here. A very important attribute of the labelling scheme is that for each proper edge region, the incoming and outgoing $<$ edges of the region are labelled in ascending order of heads and tails, respectively. This is known as the *ordering condition*. To see a very basic example of this, consider the proper edge region $\{6, 7\}$ in Figure 3. The tail of the edge (6, 10) with label [1, 1] which is 6 is less than the tail of the edge (7,11) with label [2,2] which is 7. In addition, the proper edges themselves are all labelled with [0,0]. Finally, labels of non-proper paths are sometimes reused when this creates no danger (see Figure 4‡) and thus are not entirely unique.

(continuing 3(b))

i. If $b < \mathit{terminal}(a)$,

A. If *the low endpoint of the label of* $\mathit{nextGreaterEdge}(a)$ *is less than or equal to the high endpoint of the label of the edge that determines* $\mathit{previousLesser}(b)$, $\mathit{previousLesserEdge}(b)$, then the

‡ Note that in the color diagrams, the right-hand side of the boxes should have arrows.

maximum distance is equivalent to the difference distance. This is due to the fact that when this condition holds, $\mathsf{head}(e'_j) \leq b \leq \mathsf{head}(e_{j+1}) - 1$ for some j, by the ordering condition (and the covering assumption). Observe Figure 3 with a as 3 and b as 10 to see an example of this.

B. If *the low endpoint of the label of* $\mathsf{nextGreaterEdge}(a)$ *is more than the high endpoint of the label of* $\mathsf{previousLesserEdge}(b)$, then the maximum distance is equal to one less than the difference distance. This is because when this condition holds, $\mathsf{head}(e_j) \leq b \leq \mathsf{head}(e'_j) - 1$ for some j, again due to the ordering condition. Observe Figure 3 with a as 2 and b as 8 to see an instance of this.

ii. If $b \geq \mathsf{terminal}(a)$,

A. If *the* $\mathsf{nonProperPath}(a)$ *terminates outside a proper edge region*, the maximum distance is equal to the difference distance. This is similar to Case 1. Notice Figure 3 with a as 18 and b as 22 to see an example of this.

B. If *the* $\mathsf{nonProperPath}(a)$ *terminates inside a proper edge region*,

- If $\mathsf{terminal}(a)$ *and b are in the same proper edge region*, the maximum distance corresponds to the difference distance. This is similar to Case 2. Notice Figure 3 with a as 3 and b as 15 to see an instance of this; $\mathsf{terminal}(a)$ is 14.
- If *b is not in the proper edge region that* $\mathsf{terminal}(a)$ *is in*, the maximum distance is equivalent to one less than the difference distance. This is analogous to Case 3(a). Notice Figure 3 with a as 2 and b as 19 to see an instance of this.

For the formal description of the querying algorithm, see Section 5 on page 342.

4.2 Some Simple Heuristics

Before we describe the actual algorithm that solves the MAX DIST problem in more detail, we will point out the inaccuracies of one of the many simple heuristics which we have tried unsuccessfully. This will provide a deeper appreciation of our more complex solution.

Suppose a and b are vertices of a time chain $\mathcal{G}$, $a < b$. Then, as previously noted, $\mathsf{sourceDistance}(b,a)$ can be either $\mathsf{distance}(b,a)$ or $\mathsf{distance}(b,a)+1$. We can define $\mathsf{sinkDistance}(b,a)$ in a similar way, noting that it can also be at most $\mathsf{distance}(b,a)+1$. Define the *outgoing interval* of a vertex v as the interval from v to the tail of $\mathsf{nextGreaterEdge}(v)$ inclusive. Likewise, define the *incoming interval* of v as the interval from the head of $\mathsf{previousLesserEdge}(v)$ to v inclusive. Now, it is less intuitive that if we take the minimum of both the source and sink distances over the outgoing interval of a and the incoming interval of b, the result can still be off by one.

Consider Figure 5 when b is 27. This induces an incoming interval of [25, 27]. We consider two possible instantiations for a for this example, namely, a' which is 2 inducing an outgoing interval of [2, 3] and a'' which is 4 inducing an outgoing interval of [4, 4]. The incoming interval has a uniform $\mathsf{sourceDistance}$ of 3 and $\mathsf{sinkDistance}$ of 1. The two outgoing intervals have a uniform $\mathsf{sourceDistance}$ of 0 and $\mathsf{sinkDistance}$ of 4. Based on the heuristic, the maximum distance between both a' and b and a'' and b should be 3. However, this is only true of the maximum distance between a' and b; the maximum distance between a'' and b is 2.

The problem is that we cannot tell which of $\mathsf{head}(e'_j) \leq b \leq \mathsf{head}(e_{j+1}) - 1$ or $\mathsf{head}(e_j) \leq b \leq \mathsf{head}(e'_j) - 1$ (assuming the same definitions from the previous section) holds. To put it differently, there is no way to

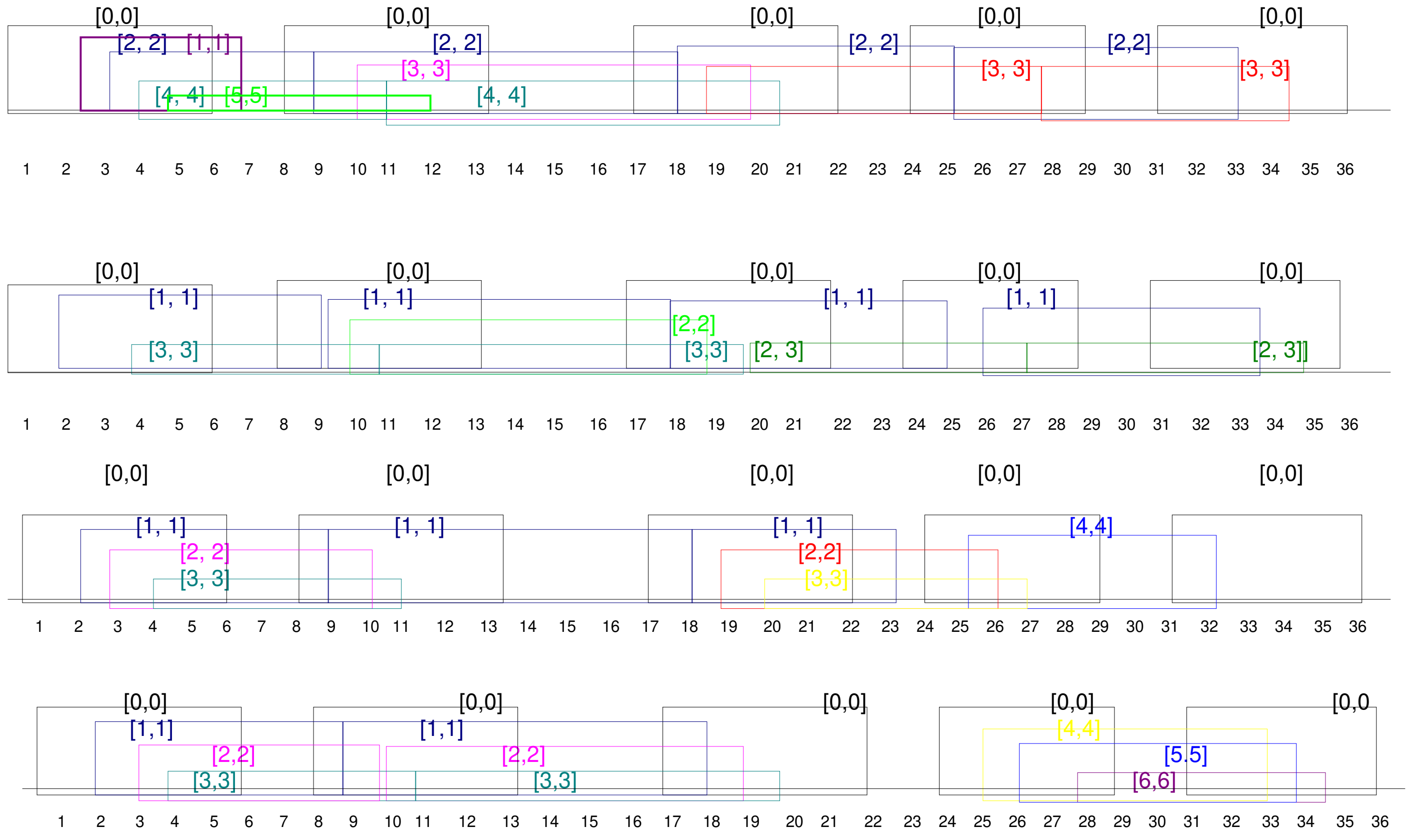


Fig. 4: Examples for labels produced by the labelling algorithm in Section 4.3.

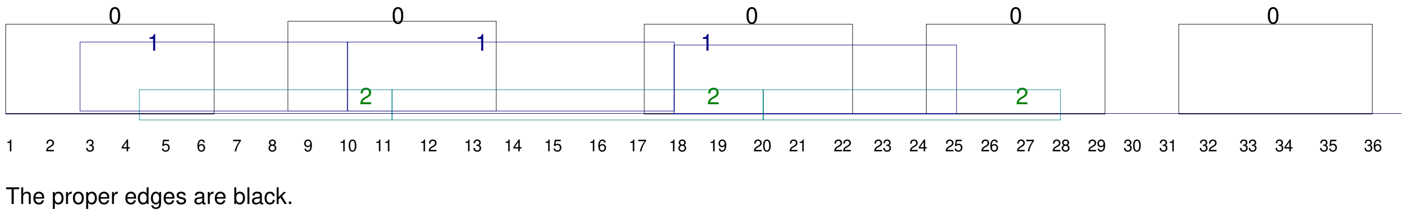


Fig. 5: Example where simple technique does not work.

know that *previousLesserEdge*(b) is not on *nonProperPath*(a'') and it actually ends before the corresponding non-proper edge of *nonProperPath*(a'') ends. In addition, *terminal*(a) must be taken into account in the computation of maximum distances.

4.3 The Labelling Algorithm

4.3.1 General Description

Now is a good opportunity to define the following fields that the formal querying algorithm in Section 5 on page 342 uses in addition to *nextGreater*(v), *previousLesser*(v) and *sourceDistance*(v) for all vertices v of $\mathcal{G}$:

startProperEdge(v) is the tail of the proper edge of the proper edge region containing v, if v is inside a proper edge region. If v is outside a proper edge region, *startProperEdge*$(v) = \infty$.

labelNextGreater(v) is the label attached to *nextGreaterEdge*(v) and it is computed by the labelling algorithm described in this section. If *nextGreater*$(v) = \infty$, it is undefined.

labelPreviousLesser$((v)$ is the label attached to *previousLesserEdge*(v) and it is computed by the labelling algorithm described in this section. If *previousLesser*$(v) = -\infty$, it is undefined.

terminal(v) is the last vertex in *nonProperPath*(v). It is used to differentiate between the case when there is *no* non-proper edge e leaving the proper edge region that v is in such that *tail*$(e) \geq v$ and the instance when there is such an edge. It is undefined in the former instance; an example of this in Figure 3 is that *terminal*(15) is undefined. When v is outside a proper edge region and before the head of the last proper edge in the chain, then *terminal*(v) will correspond to the head of the last proper edge in the chain (in Figure 3, *terminal*$(16) = 22$). If v is at or after the head of the last proper edge in the chain, *terminal*(v) is undefined (in Figure 3, *terminal*(22) is undefined).

Our preprocessing algorithm labels the proper edges and assigns the terminal fields that are associated with proper edges, while the *startProperEdge* and *sourceDistance* fields are being calculated. The *nextGreater* field is computed by the mechanisms detailed in [GS95] prior to this. Subsequently, the labelling algorithm assigns labels to all the non-proper edges in a chain. In addition, it also assigns the *terminal* fields associated with all the non-proper paths. Care must be taken to ensure that the resulting labels obey the ordering condition as it is a vital part of the correctness of the query algorithm. Note that the labels are assigned in two passes through the chain. The purpose of the first pass of preprocessing for the label assignment of the second pass is twofold: to calculate the count of distinct numbers used in the entire labelling and to compute the size of the range of the label of each individual non-proper edge [§]. The count of distinct labels needed is basically equivalent to the number of distinct non-proper paths in the chain, since each non-proper edge must have its own label. However, when a non-proper path has terminated, its label can be reused. When non-proper paths merge, the merged path is not included in the count. In addition, the size of the range of a label is always one except where several non-proper paths merge into one path. Then, the size of the range of the label is the sum of the sizes of the ranges of the labels of the "last" $<$ edges of the non-proper paths merging together.

The first pass considers each proper edge region one by one from the source to the sink (see Fig. 6). Computing the number of distinct labels needed and calculating the size of the range of a label is done in

§ If we let a label assigned in this fashion be $[\ell, h]$ and the size of its range be s, $\ell = h - s + 1$.

the same way. To compute these values, the pattern of the heads of the incoming $<$ edges and the tails of the outgoing $<$ edges for each proper edge region is examined (refer to the following section for a more complete description of this).

The second pass involves assigning labels that form a contiguous interval of the positive integers using the information gathered in the first pass. In addition, it assigns the *terminal* fields associated with non-proper paths. This pass scans each proper edge region in turn from the *sink to the source*, the reverse of the first pass. The first proper edge region considered is the nearest proper edge region from the sink having outgoing edges. It is interesting to observe that if the second pass would scan from source to sink, fractional labels could be required and this would result in significant complications. Thus, to label the outgoing edges in a proper edge region R_i, we do the following:

1. If R_{i+1} has no outgoing edges or $R_i = R_p$ (defined in Section 3.1 on page 328, i.e. the last proper edge region of the chain), the outgoing edges are processed in decreasing tail order. These edges are then given labels, the high endpoint of which is the value of the next number to be used in a label. The low endpoint of the labels is based on the size of the range of the label of the edge together with the high endpoint of the labels . After each label is assigned, the next number to be used is decremented by the size of the range of the label. Note that the next number to be used in a label is initially set to be the number of labels needed, which was determined in the first pass. Some examples of this case are the labelling of (28,35) and (25,33) in the first chain and all the $<$ edges of the last chain except (2, 9), (3, 10), and (4, 11) in Figure 7.

2. Otherwise, the outgoing edges of R_i that are part of a non-proper path for which at least one edge has been labelled are identified (see next section for more details). This is important since the labels of all the $<$ edges of a particular non-proper path are the same up to a partition of a range (when the path is considered in reverse). This is done by examining the pattern of the heads of the incoming edges (outgoing edges of R_i) and the tails of the outgoing edges of R_{i+1}.

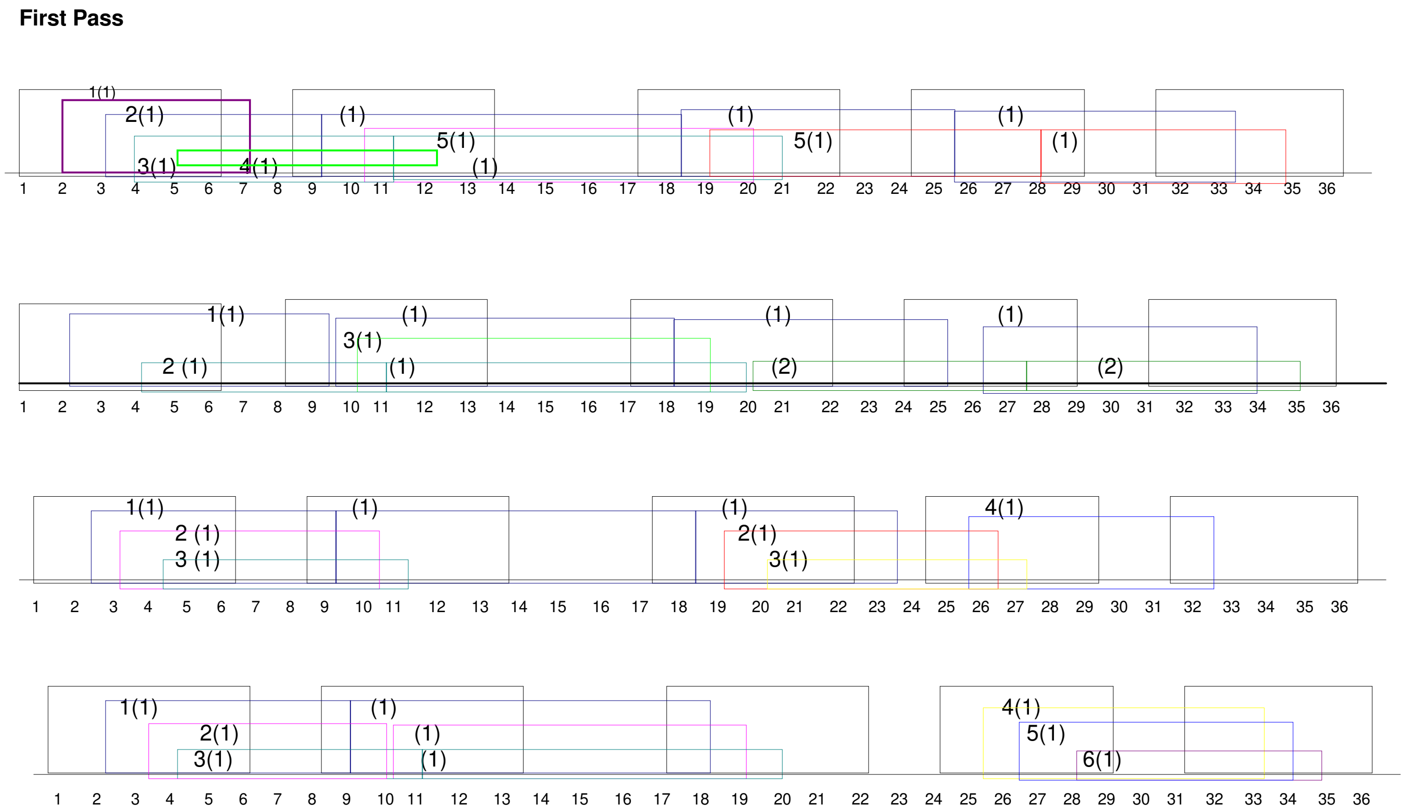
   (a) If there is no such edge, Case 1 is applied. An instance of this case us the labelling of (18,23), (19, 26) and (20,27) in the third chain in Figure 7.

   (b) If there is at least one such edge, each edge is labelled appropriately. Let the edges labelled in this way be $e''_1, e''_2, \ldots, e''_y$ in increasing tail order. Let the label of e''_1 be $[\ell_1, h_1]$ and let the label of e''_y be $[\ell_y, h_y]$. Some illustrations of this case are the labelling of all the $<$ edges in the second chain except (26, 34) and (27, 35) and the $<$ edges (2, 9), (3, 10), and (4, 11) in the last chain in Figure 7.

   (c) Any remaining outgoing edges of R_i with lower tails than e''_1 are labelled in decreasing tail order. The high endpoint of the first edge to be labelled this way is $\ell_1 - 1$. After each label is assigned in this fashion, the next number to be used is decremented by the size of the range of the label. An example of this is the labelling of (2,7) in the first chain in Figure 7,

   (d) As well, any remaining outgoing edges with higher tails than e''_y are labelled in increasing tail order. The low endpoint of the first edge to be labelled this way is $h_y + 1$. Two examples of this are the labelling of (3,10) and (4, 11) in the third chain in Figure 7,

See Figures 4 and 7 for clarification.



c(d) on an edge e means that on encountering e, the number of distinct labels is incremented to c and the size of the range of the label of e is d. c is omitted when e is not the first < edge of a non-proper path. Note that in the first chain after R_3, the number of distinct labels needed is 3.

Fig. 6: Examples for the working of the first pass of the labelling algorithm.

As a guide, the marker of (3,9) in the first chain is 2(1) since it is the second outgoing edge encoutered, and the size of the range of the label is 1, as there are no heads of incoming edges immediately before the tail of (3, 9). For (9, 18), the marker is (1), since it is part of the non-proper path of which (3,9) is the first < edge. The marker of (20, 27) in the second chain is (2), since the sum of size of the range of the labels of (10, 19) and (11, 20) is 2.

When the *terminal* of a non-proper path is encountered during this pass of the labelling algorithm, it is indexed under all the numbers included in the range of the label of the last $<$ edge of the non-proper path. Then, to fill the *terminal* fields of the vertices in a proper edge region immediately preceding a $<$ edge of a particular non-proper path, the information indexed under the lower endpoint of the label of the edge is retrieved. This is a default; the upper endpoint of the label could be used instead for the same effect. Naturally, this is done after the outgoing edges of the proper edge region have been assigned labels. As an example, let us reflect on the second chain of Figure 4. We register 35 as the *terminal* under indices 2 and 3 and then 34 as the *terminal* under index 1. Consider the *terminal* field assignments in R_4, i.e. $\{25, \ldots, 28\}$ (the rest are similar). Vertices 25 and 26 get the *terminal* field indexed under 1, i.e. 34, and vertex 27 gets the *terminal* field indexed under 2, i.e. 35.

In other words, for any vertex v, $terminal(v)$ is determined by the value indexed under the lower endpoint of $labelNextGreater(v)$, given that $nextGreaterEdge(v)$ is a non-proper edge. This is a default; the upper endpoint of the label could be used instead for the same effect. This is because of some properties of merging process; see Observations 3 and 5 in Section 4.3.3 on page 339. In addition, observe that the reuse of labels causes no problems, due to Observation 4.

4.3.2 Some More Details

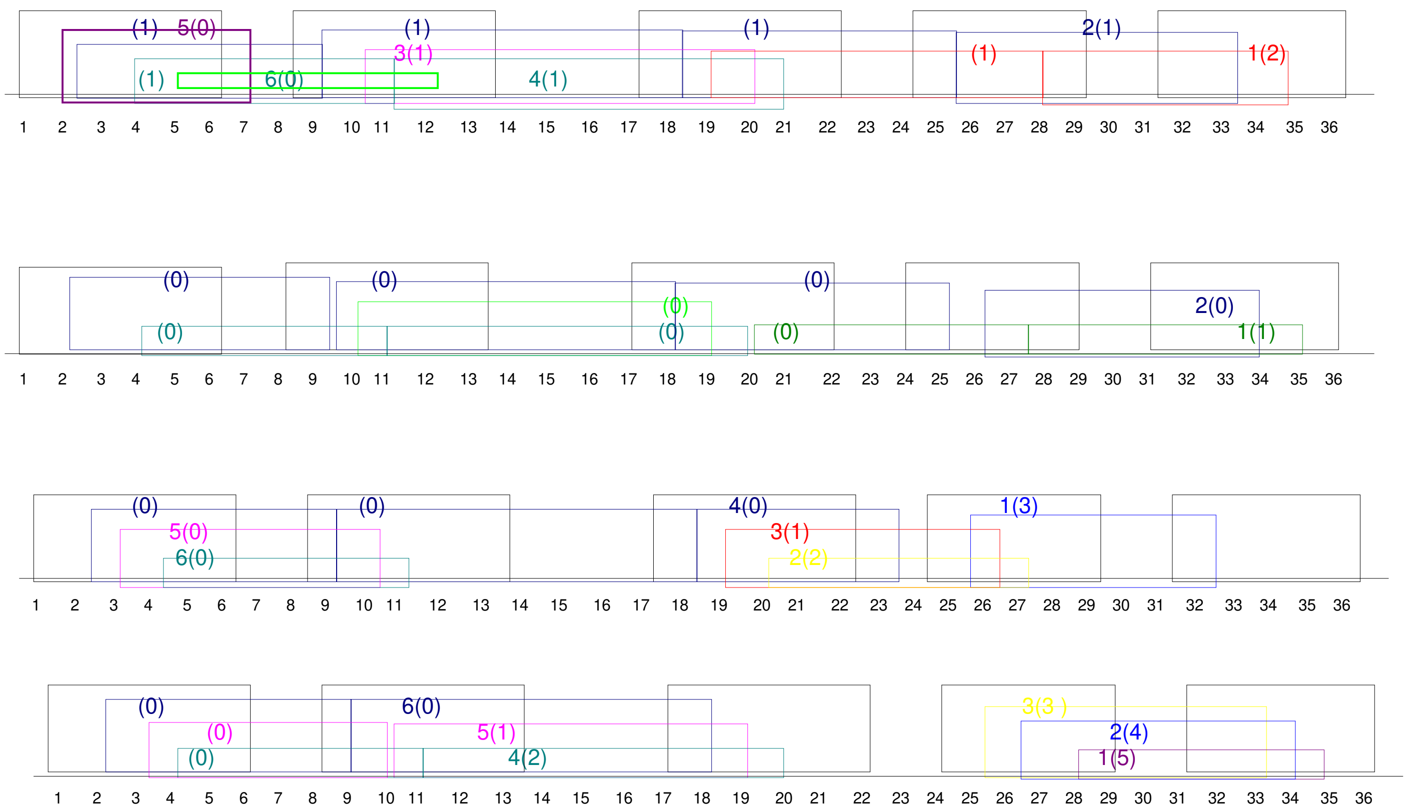
Here, we explain in further detail the way in which the number of distinct labels and the size of the range of the label for each non-proper edge are computed in the first pass of the algorithm.

We will focus on a single proper edge region, R_i. We inspect the sequence of heads of incoming edges and the tails of outgoing edges of R_i from left to right. If there are no heads of incoming edges immediately before the tail of a particular outgoing edge $e_{o,k}$, the count of distinct labels required is incremented by one. As well, the size of the range of the label of $e_{o,k}$ is one. This is due to the fact that $e_{o,k}$ is the first $<$ edge of a non-proper path. Otherwise when there is at least one head of an incoming edge immediately before a particular outgoing edge $e_{o,k}$, the count of distinct labels needed is not incremented. The cause of this is that under these conditions, $e_{o,k}$ is part of a non-proper path which has been already encountered. In other words, each incoming edge $e_{i,j}$ immediately before the tail of a particular outgoing edge $e_{o,k}$ satisfies $nextGreater(head(e_{i,j})) = head(e_{o,k})$; this is referred to as the *nextGreater condition*. Thus, the size of the range of the label of $e_{o,k}$ is the sum of the sizes of the ranges of the labels of each of the incoming edges, the heads of which are immediately before $e_{o,k}$. For every successive head of an incoming edge encountered, this sum is accumulated. As an example of this, consider the second chain of Figure 6. The size of the range of the labels of (10, 19) and (11,20) is 1, but the size of the range of the label of (20, 27) is the sum of these ranges which is 2.

When there is no outgoing edge following a sequence of at least one incoming edge, we subtract the accumulated sum from the number of distinct labels required, provided that the *nextGreater* condition has been satisfied at least once for R_i. This is because in that situation, Case 2(d) applies for the second pass. When we do the subtraction, the range of contiguous numbers used in the entire labelling will always have a lower endpoint of 1. An example of this case is found in the third chain of Figure 6. Upon encountering the heads of (3,10) and (4, 11) in R_2, the count of distinct labels needed is decremented to 1 from 3. Then, after we reach the tail of (19, 26) in R_3, the count of distinct labels is 2. If the subtraction would not be done, the label for (2,9) would be 3 and not 1.

For the second pass of the labelling algorithm, the process of identifying the $<$ edges belonging to particular non-proper paths that have been previously encountered is similar to the above method. To recognize the outgoing edges of R_i that are part of a non-proper path for which at least one edge has been

Second Pass



c(d) on an edge e means that the non-proper path of e is the "c"th in line to be assigned a label and the next number to be used for a label on encountering e is d. c is omitted when e is not the last < edge of a non-proper path.

Fig. 7: Examples for the working of the second pass of the labelling algorithm.

labelled, we examine the sequence of heads of incoming edges and the tails of outgoing edges of R_{i+1} from left to right. Once the outgoing edge following a sequence of incoming edges (we keep track of the first and last edges of this sequence) is reached, we go back and label the incoming edges according to the label of the outgoing edge and the size of the ranges of the labels of the incoming edges. To assign labels to the incoming edges of R_{i+1} (outgoing edges of R_i) for which the *nextGreater* condition can be met, the following rules are applied:

> First, assume that for an interval $I = [\ell, h]$, $\mathit{first}(I) = \ell$, $\mathit{last}(I) = h$. Let the size of the range of the label of a non-proper edge e be $|\mathit{label}(e)|$, $\mathit{label}(e)$ being the label of e.
>
> Now, if $\mathit{nextGreater}(\mathit{head}(e_{i,j})) = \mathit{head}(e_{o,k})$, for $j = g$ to f[¶] in ascending head order,
>
> $$\mathit{label}(e_{i,g}) = [\mathit{first}(\mathit{label}(e_{o,k})), \mathit{first}(\mathit{label}(e_{o,k})) + |\mathit{label}(e_{i,g})| - 1].$$
>
> For $j = g+1$ to f,
>
> $$\mathit{label}(e_{i,j}) = \left[\mathit{last}(\mathit{label}(e_{i,j-1})) + 1, \mathit{last}(\mathit{label}(e_{i,j-1})) + \left|\mathit{label}(e_{i,j})\right|\right].$$

Note that we must keep track of the first incoming edge of R_{i+1} for which the *nextGreater* condition cannot be met, so as to carry out Case 2(d) of the second pass when it is necessary.

As an example, let us consider the labelling of the outgoing edges of R_2 of the second chain of Figure 4. It is determined that $\mathit{nextGreater}(\mathit{head}((9,18))) = \mathit{head}((18,25))$, so $\mathit{label}((9,18)) = [1, 1+1-1] = [1,1]$. Next, it is determined that $\mathit{nextGreater}(\mathit{head}((10,19))) = \mathit{nextGreater}(\mathit{head}((11,20))) = \mathit{head}((20,27))$, so $\mathit{label}((10,19)) = [2, 2+1-1] = [2,2]$ and $\mathit{label}((11,20)) = [2+1, 2+1] = [3,3]$.

4.3.3 Properties of the Labelling Algorithm

Rational numbers present problems in terms of storage and access. This is especially significant in terms of computing and assigning the *terminal* fields, and rational number labels would cause undue complications in the labelling scheme. Consequently, the following three observations are important.

Lemma 1 *All the outgoing edges of a proper edge region for which the* nextGreater *condition can be met are labelled.*

Proof: Suppose that there is a gap between the outgoing edges of a proper edge region for which the *nextGreater* condition can be met that are labelled. Then, there has to be at least one unlabelled edge e_y between two edges e_x and e_z labelled arbitrarily with $[\ell_x, h_x]$ and $[\ell_z, h_z]$ respectively, among the outgoing edges of the proper edge region R_i. This case must hold after all the outgoing edges of a proper edge region for which the *nextGreater* condition can be met have been identified and assigned labels. In addition, assume that e_x, e_y and e_z are in ascending tail order. For this to be true, outgoing edges labelled with $[\ell_x, h_x]$ and $[\ell_z, h_z]$ respectively must be consecutive in R_{i+1}. There are two possible ways that there is no edge with a label that corresponds to e_y in R_{i+1}. One way is when the head of e_y is outside a proper edge region and between R_i and R_{i+1}. However, to obey the covering assumption, the head of e_x must also be outside a proper edge region between R_i and R_{i+1}. Thus, the premise that edges labelled $[\ell_x, h_x]$ and $[\ell_z, h_z]$ respectively are consecutive in R_{i+1} and e_y has no label is violated. This is because then e_x would not get

[¶] The variables f and g are merely used for "indexing".

its label from an edge labelled $[\ell_x, h_x]$ in R_{i+1} (see Figure 8(a)). The other way is that the heads of e_x, e_y and e_z are in R_{i+1} and are all in ascending order. Again, a contradiction is reached since now e_y should get its label from the edge in R_{i+1} that is part of the same non-proper path which e_z is a part of; e_y and e_z merge into one path (refer to Figure 8(b)). The statement of the lemma follows from this. □

Lemma 2 *The labelling algorithm labels every non-proper edge.*

Proof: For the proper edge regions in which Case 1 of the second pass of the labelling algorithm applies, it is fairly obvious that all the outgoing edges are labelled since the labelling proceeds through all the outgoing edges one by one in decreasing tail order.

For the proper edge regions in which Case 2 of the second pass of the labelling algorithm applies, it suffices to know that the edges labelled in Case 2(b) form a contiguous "block" of labelled edges. By Lemma 1 and the property of *nextGreater* fields that all the vertices immediately before a $<$ edge have the head of the edge as their *nextGreater* field value, this holds. As an aside, it is interesting to note that the outgoing edges with lower tails than those in the block (when they exist) all end outside a proper edge region. All the outgoing edges with tails lower than those of the block are labelled in descending tail order in Case 2(c). In addition, all the outgoing edges with tails higher than those of the block are labelled in ascending tail order in Case 2(d). Since every non-proper edge is a outgoing edge of some proper edge region by the covering assumption, the lemma follows from this. □

Claim 1 *Fractional labels are not needed when using the labelling scheme described.*

Proof: This follows from Lemmas 1 and 2. □

The following claim expresses an essential attribute for the labelling scheme to enable the query algorithm to function correctly.

Claim 2 *The ordering condition holds.*

Proof: The proof is by construction of the labels. □

It is important that the labels of non-proper paths form a contiguous interval of the positive integers so that when non-proper paths merge, the label ranges are consistent and the *terminal* fields are retrieved properly.

Claim 3 *The numbers used in the labels of non-proper paths form a contiguous interval of the positive integers.*

Proof: The proof is by construction of the labels. □

A couple more characteristics of the labelling algorithm follow.

Claim 4 *The label of a non-proper path can be reused after the path has terminated or before the path has started, and this is the only time that labels are reused by the labelling method.*

Proof: As long as there is a $<$ edge of a particular non-proper path in a proper edge region, the label of that non-proper path is in a sense "reserved". This is because the label of all the $<$ edges of a particular non-proper path is the same up to a partition of a range (when the path is considered in the sink to source direction). Since a label of a non-proper path only needed for the span of the path, it can be safely reused in proper edge regions outside this span. The only situation when labels may be reused is in Case 2(d) of the labelling algorithm. Observe Figure 4 which shows the label assignment for the same chains shown in the first pass and second pass illustrations of Figures 6 and 7 for examples of label reuse; for example, the $<$ edges (3, 10) and (19, 26) as well as (4, 11) and (20, 27) in the third chain have the same labels. □

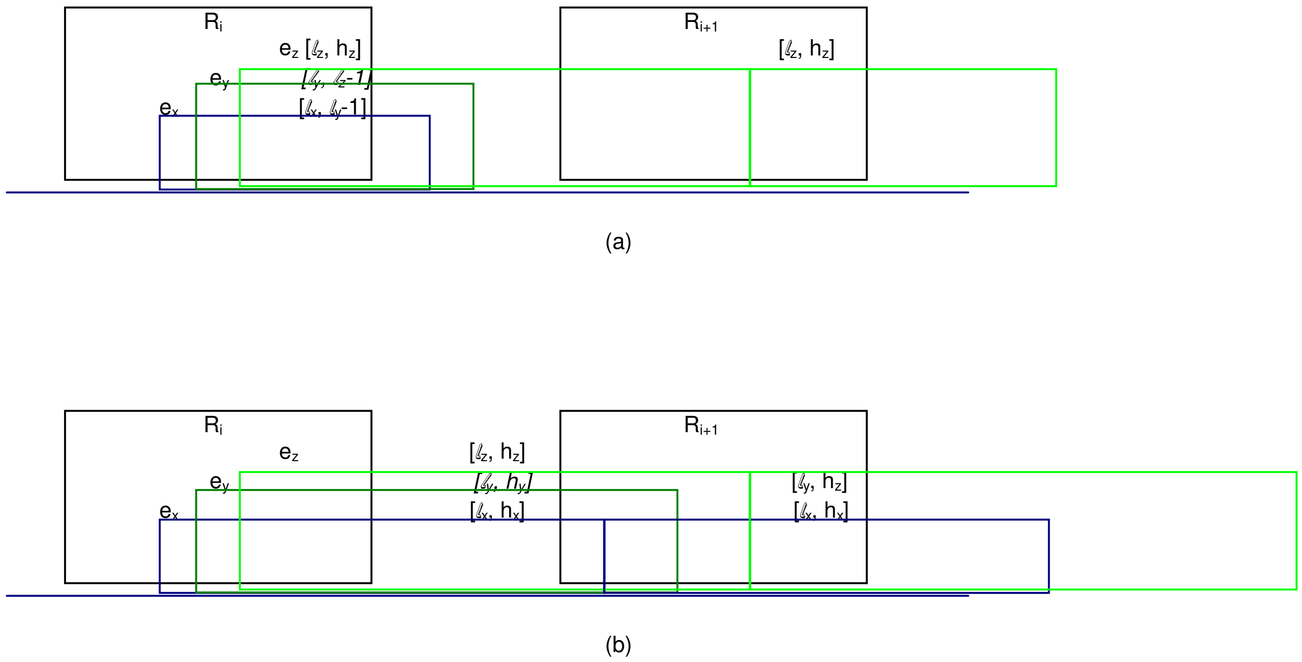


Fig. 8: Scenarios of proof of Observation 1.

There can be any number of < edges, the tails of which are before the tail of e_x or after the tail of e_z in R_i. The < edges shown are the bare minimum.

Claim 5 *Distinct label ranges associated with a particular proper edge region are not overlapping, i.e. if a $<$ edge e_i with a label of $[c,d]$ and a $<$ edge e_j with a label of $[e,f]$ are both entering or leaving the same proper edge region, either $d < e$ or $f < c$.*

Proof: The proof is by construction of the labels. Once several different non-proper paths merge into one path, they cannot become separate again. □

4.4 The Complexity Results

Theorem 1 *The running time of the labelling algorithm is O(n), where $n = |\mathcal{V}|$ for a chain $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.*

Proof: The first pass scans each proper edge region from the source to the sink. The sequence of heads of incoming edges and the tails of outgoing edges of each proper edge region is examined from left to right. Carrying out the first pass means passing over each non-proper edge twice. Thus, the work done in the first pass takes O(n) time. This is because the maximum number of $<$ edges possible in a chain is $n-1$ due to the covering assumption.

The second pass scans each proper edge region from the sink to the source. The sequence of heads of incoming edges and the tails of outgoing edges of each proper edge region (after the first to be considered for the labelling and except where the previously considered proper edge region has no outgoing edges) is inspected from left to right. Carrying out the second pass means passing over each non-proper edge at most 3 times. Thus, the work done to assign the labels takes O(n) time.

As well, the time needed to assign the *terminal* fields associated with non-proper edges is O(n) as the terminal of each non-proper path is discovered once and the number of non-proper paths is bounded above by $n-1$. Also, the size of the indexed storage that keeps track of the *terminal* fields associated with edges having certain labels is $n-1$. Thus, the time taken by the second pass is in the order of n. The fact that the maximum number of $<$ edges possible in a chain is $n-1$ under the covering assumption really underlies this bound. Therefore, the total time taken by the labelling algorithm is O(n). □

5 The Formal Querying Algorithm

5.1 Preliminaries

Theorem 2 *$\mathit{distance}(b,a) = \|\mathit{ngTraversal}(a,b)\|$, where a and $b \in \mathcal{V}$ and $a < b$ for some chain $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.*

Proof: First, once the number of $<$ edges of a path from a to b is known, we know that $\mathit{distance}(b,a)$ can be no less. Thus, $\mathit{distance}(b,a)$ must be at least $\|\mathit{ngTraversal}(a,b)\|$. Now, we must prove that $\mathit{distance}(b,a)$ can be no more than $\|\mathit{ngTraversal}(a,b)\|$. If we imagine adding another distinct $<$ edge, the head and tail of which are both outside the region of any $<$ edge contained in $\mathit{ngTraversal}(a,b)$, the added edge would be a $<$ edge of $\mathit{ngTraversal}(a,b)$ and $\|\mathit{ngTraversal}(a,b)\|$ would be increased by 1. This does not make $\mathit{distance}(b,a)$ more than $\|\mathit{ngTraversal}(a,b)\|$ (i), and if the added edge is inside a proper edge region, the covering assumption is violated as well. The only way that $\mathit{distance}(b,a)$ could be more than $\|\mathit{ngTraversal}(a,b)\|$ would be to have at least two $<$ edges contained (and at least one edge must be completely enclosed) in the same $<$ edge that is a part of $\mathit{ngTraversal}(a,b)$ (iia) and (iib). However, this is a contradiction by the covering assumption. Thus, $\mathit{distance}(b,a) = \|\mathit{ngTraversal}(a,b)\|$. See Fig. 9. □

Corollary 1 *$\mathit{distance}(b,a) = \mathit{distance}(b',a) + \mathit{distance}(b,b')$, provided that b' is outside the region of any $<$ edge contained in $\mathit{ngTraversal}(a,b)$.*

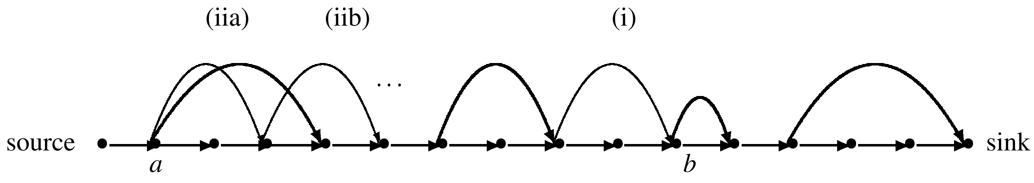


Fig. 9: Proof of Theorem 2.

Proof: From the definition of the *nextGreater* traversal, we have $||ngTraversal(a,b)|| = ||ngTraversal(a,b')|| + ||ngTraversal(b',b)||$. From this and Theorem 2, the corollary follows. □

Now, the way in which the path induced by $sourceDistance(b,a)$ consisting of only proper edges among the $<$ edges compares to $ngTraversal(a,b)$ will be analyzed.

Henceforth, assume that a is inside a proper edge region and b' is not in the proper edge region that a is in. In addition, there is a non-proper edge e leaving the proper edge region that a is in such that $tail(e) \geq a$.

Theorem 3
Under these assumptions, if $head(e'_j) \leq b' \leq head(e_{j+1}) - 1$[‖] *and* $b' \leq terminal(a)$, $distance(b',a) = sourceDistance(b',a) = j$ *(i). In addition, if* $head(e'_j) = b' = terminal(a)$, $startProperEdge(b') = \infty$, $distance(b',a) = sourceDistance(b',a) = j$ *(ii).*

Proof: We proceed by induction on j.

Basis Case: For j=1, there is one non-proper edge between a and b', namely, e'_1. As well, $a < head(e_1) < b'$ and the head of a proper edge is the only place where the distance from the source increases. Thus, $distance(b',a) = sourceDistance(b',a) = 1$ (see Fig. 10 and Fig. 11 for cases (i) and (ii), respectively). The basis case is established.

Inductive Case: Assume the inductive hypothesis holds when $i < j$, for some j. Now, we prove that it holds for j. Assume that $head(e'_j) \leq b' \leq head(e_{j+1}) - 1$ and $b' \leq terminal(a)$ or $head(e'_j) = b' = terminal(a)$, $startProperEdge(b) = \infty$. Since the inductive hypothesis holds when $i < j$, if $head(e'_{j-1}) \leq b'' \leq head(e_j) - 1$, $distance(b'',a) = sourceDistance(b'',a) = j-1$. Now, there is an additional non-proper edge e'_j in $ngTraversal(a,b')$ compared to $ngTraversal(a,b'')$. Thus, $distance(b',a) = distance(head(e'_{j-1}),a) + 1 = j-1+1 = j$ and $sourceDistance(b',a) = sourceDistance(head(e'_{j-1}),a) + 1 = j-1+1 = j$ by Corollary 1 (see Fig. 12 and Fig. 13 for cases (i) and (ii), respectively). Again, the fact that the head of a proper edge is the only place where the distance from the source increases has been used. The inductive case is established. □

We maintain our assumption that a is inside a proper edge region and b' is not in the proper edge region that a is in. As well, there is a non-proper edge e leaving the proper edge region that a is in such that $tail(e) \geq a$.

Theorem 4 *Under these assumptions, if* $head(e_j) \leq b \leq head(e'_j) - 1$ *and* $b < terminal(a)$, $distance(b',a) = sourceDistance(b',a) - 1 = j - 1$.

Proof: The proof is by induction on j and is similar to the proof of Theorem 3. □

‖ We assume the same notation for the proper edges and the $<$ edges. of the non-proper path as in Section 4.1 on page 329.

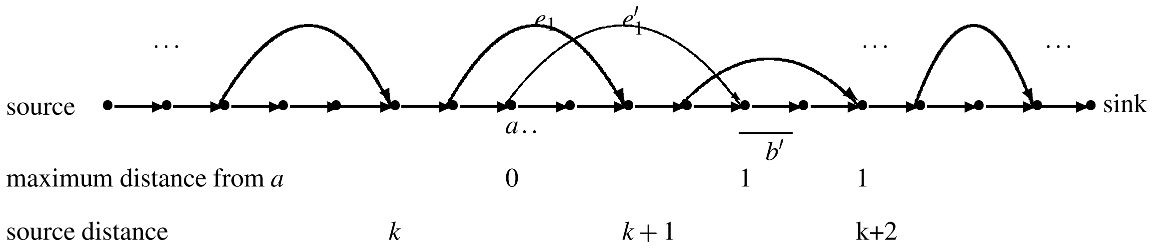


Fig. 10: Basis Case of proof of Theorem 3(i).

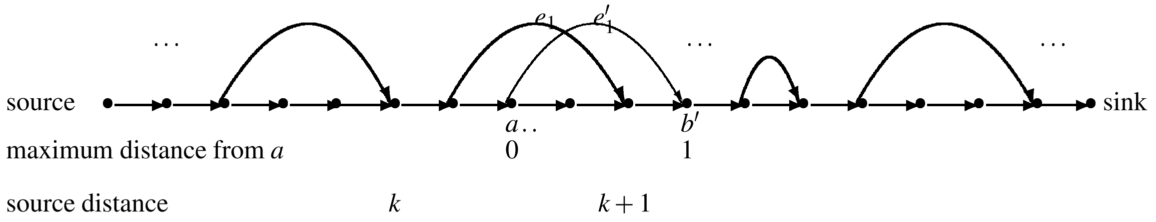


Fig. 11: Basis Case of proof of Theorem 3(ii).

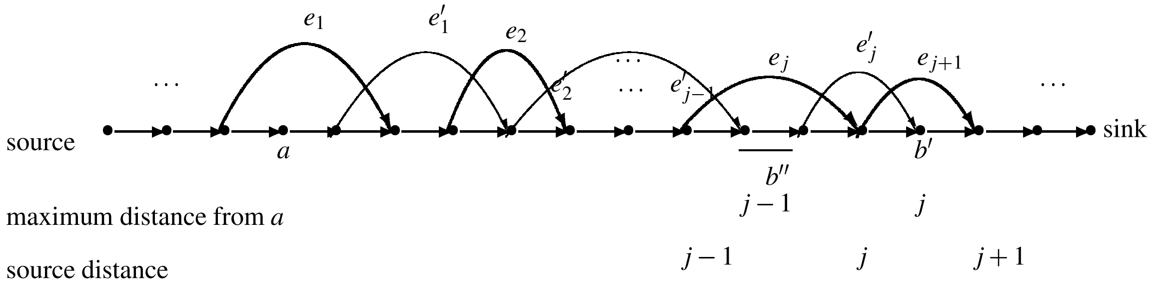


Fig. 12: Inductive Case of proof of Theorem 3(i).

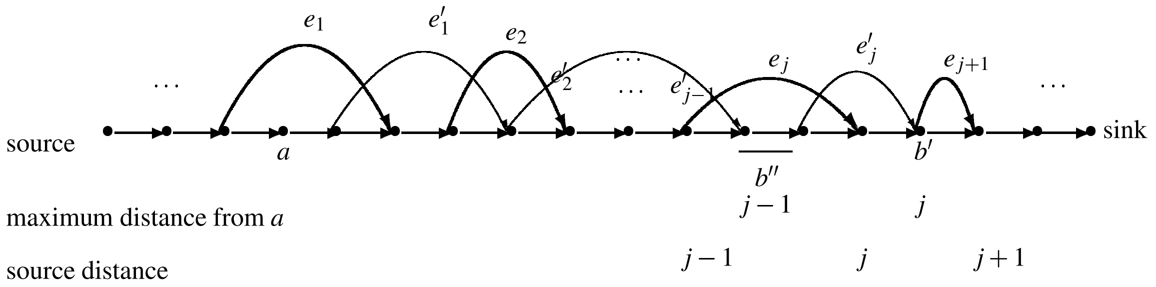


Fig. 13: Inductive Case of proof of Theorem 3(ii).

5.2 The Actual Query Algorithm and its Proof

Again, note that for an interval $I = [\ell,h], first(I) = \ell, last(I) = h$. As such, the query algorithm is as follows:

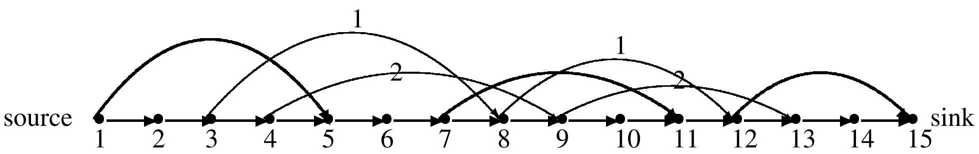
1. If $startProperEdge(a) = \infty$, then $distance(b,a) = sourceDistance(b,a)$.
2. If $startProperEdge(a) \neq \infty$ and $b < nextGreater(startProperEdge(a))$, then $distance(b,a) = 0$.
3. If $startProperEdge(a) \neq \infty$ and $b \geq nextGreater(startProperEdge(a))$
   (a) If $terminal(a)$ = "undefined", then $distance(b,a) = sourceDistance(b,a) - 1$.
   (b) If $terminal(a) \neq$ "undefined"
      i. If $b < terminal(a)$
         A. If $first(labelNextGreater(a)) \leq last(labelPreviousLesser(b))$, then $distance(b,a) = sourceDistance(b,a)$.
         B. If $first(labelNextGreater(a)) > last(labelPreviousLesser(b))$, then $distance(b,a) = sourceDistance(b,a) - 1$.
      ii. If $b \geq terminal(a)$,
         A. If $startProperEdge(terminal(a)) = \infty$, $distance(b,a) = sourceDistance(b,a)$.
         B. If $startProperEdge(terminal(a)) \neq \infty$,
            - If $b < nextGreater(startProperEdge(terminal(a)))$, then $distance(b,a) = sourceDistance(b,a)$.
            - If $b \geq nextGreater(startProperEdge(terminal(a)))$, then $distance(b,a) = sourceDistance(b,a) - 1$.

Figure 14 on the next page gives an example for each case of the query algorithm. According to the illustrated chain we have:

1. $distance(15,5) = sourceDistance(15,5) = 2$.
2. $distance(10,8) = 0$.
3. (a) $distance(15,10) = sourceDistance(15,10) - 1 = 1$.
   (b) i. A. $distance(9,3) = sourceDistance(9,3) = 1$.
         B. $distance(8,4) = sourceDistance(8,4) - 1 = 0$.
      ii. A. $distance(15,3) = sourceDistance(15,3) = 3$.
         B. • $distance(14,4) = sourceDistance(14,4) = 2$.
            • $distance(15,4) = sourceDistance(15,4) - 1 = 2$.

Theorem 5 *The query algorithm is correct with respect to computing the maximum distance between any two vertices a and $b \in \mathcal{V}$ such that $a < b$.*

Proof: The cases in the following proof correspond exactly to those in the query algorithm.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	vertex
0	0	0	0	1	1	1	1	1	1	2	2	2	2	3	sourceDistance
∞	1	1	1	∞	∞	∞	7	7	7	∞	∞	12	12	∞	startProperEdge
5	8	8	9	11	11	11	12	13	15	15	15	∞	∞	∞	nextGreater
$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	1	1	3	4	4	7	8	9	9	12	previousLesser
0	1	1	2	0	0	0	1	2	0	0	0	-	-	-	labelNextGreater
-	-	-	-	0	0	0	1	2	2	0	1	2	2	0	labelPreviousLesser
15	12	12	13	15	15	15	12	13	-	15	15	-	-	-	terminal

(a) Fields of Example Time Chain

Fig. 14: Example Time Chain. The proper edges are bold.

1. By Theorem 2, $\mathit{distance}(b,a) = ||\mathit{ngTraversal}(a,b)||$. Since a is outside a proper edge region, $\mathit{ngTraversal}(a,b)$ is exactly the path induced by $\mathit{sourceDistance}(b,a)$. This is because the source is also outside a proper edge region and so there is a discrete number of proper edges between the source and a. So in this case the number of $<$ edges in $\mathit{ngTraversal}(a,b)$ is equal to $\mathit{sourceDistance}(b,a)$. Thus, $\mathit{distance}(b,a)$= $\mathit{sourceDistance}(b,a)$.

2. If $b < \mathit{nextGreater}(\mathit{startProperEdge}(a))$, there can be no $<$ edge (u,v) such that $a \leq u < v \leq b$ by the covering assumption; otherwise (u,v) would subsume the $<$ edge $(\mathit{startProperEdge}(a), \mathit{nextGreater}(\mathit{startProperEdge}(a)))$. So, $\mathit{distance}(b,a)$ =0.

3. (a) By Theorem 2, $\mathit{distance}(b,a) = ||\mathit{ngTraversal}(a,b)||$. In addition, the path induced by $\mathit{sourceDistance}(b,a)$ has one $<$ edge that is not present in $\mathit{ngTraversal}(a,b)$ (the other $<$ edges are all in common between the two paths), namely the $<$ edge $(\mathit{startProperEdge}(a), \mathit{nextGreater}(\mathit{startProperEdge}(a)))$. This is because this edge is on $\mathit{ngTraversal}(\mathit{source},b)$ and it is not on $\mathit{ngTraversal}(\mathit{source},a)$ nor is it on any path beginning at a. Thus, $\mathit{distance}(b,a) = \mathit{sourceDistance}(b,a) - 1$.

   (b) i. A. Assume the antecedent holds, Theorem 3 (i) applies.
   Let $\mathit{first}(\mathit{labelNextGreater}(a)) = \ell_a$ and let $\mathit{last}(\mathit{labelPreviousLesser}(b)) = \ell_b$. Essentially what must be shown is that if $\ell_a \leq \ell_b$, then $\mathit{head}(e'_j) \leq b \leq \mathit{head}(e_{j+1}) - 1$ for some j, and from this, $\mathit{distance}(b,a) = \mathit{sourceDistance}(b,a)$=j. Assume that $\ell_a \leq \ell_b$.

   b **is inside a proper edge region:** Assume that b is outside a proper edge region. Now, b cannot be directly after the head of a proper edge. This is because if b were directly after the head of a proper edge, then ℓ_b would be 0; so only when ℓ_a is 0, is it possible that

$\ell_a \leq \ell_b$. But $\ell_a \neq 0$ as there is a non-proper edge e leaving the same proper edge region that a is in such that $tail(e) \geq a$. As well, if we assume that b is directly after the head of a non-proper edge such that $\ell_a \leq \ell_b$ and (u,v) is a $<$ edge where $u = previousLesser(b)$, then $head(e'_j) \leq v \leq b$ **. This is because the outgoing edges of each proper edge region are labelled in ascending tail order by Claim (Lemma) 2. But since b is outside a proper edge region, $head(e'_j)$ is also outside a proper edge region by the covering assumption ($tail(e'_j)$ is in the proper edge region of e_j so $tail(e'_j) < head(e'_j) \leq b$). So $b \geq terminal(a)$ contradicting our assumption that $b < terminal(a)$. Therefore b is inside a proper edge region, i.e. $tail(e_{j+1}) + 1 \leq b \leq head(e_{j+1}) - 1$ for some j.

$head(e'_j) \leq b$ for some j: Assume that b is inside a proper edge region and $head(e'_j) > b$. However, the incoming edges of each proper edge region are labelled in ascending head order by Claim (Lemma) 2. For this to be the case and for $head(e'_j) > b$, $\ell_a > \ell_b$ ‡ but this contradicts the assumption that $\ell_a \leq \ell_b$. Therefore, $head(e'_j) \leq b$.

Thus, $head(e'_j) \leq b \leq head(e_{j+1}) - 1$ and $distance(b,a)= sourceDistance(b,a) = j$.

B. Given the antecedent, Theorem 4 applies. Essentially what must be shown is that if $\ell_a > \ell_b$, then $head(e_j) \leq b \leq head(e'_j) - 1$ for some j and from this $distance(b,a) = sourceDistance(b,a) - 1 = j - 1$. Since $head(e'_j) \leq b \leq head(e_{j+1}) - 1$ and $head(e_j) \leq b \leq head(e'_j) - 1$ for some j completely define the places that b can be under the assumptions of the antecedent of this case, it suffices to prove that if $\ell_a > \ell_b$, it is not the case that $head(e'_j) \leq b \leq head(e_{j+1}) - 1$. Suppose that $head(e'_j) \leq b \leq head(e_{j+1}) - 1$. But then taking (u,v) to be a $<$ edge where $u = previousLesser(b)$, we have $head(e'_j) > v$‡ through the assumption that $\ell_a > \ell_b$ and the sorted ascending order of incoming edges expressed in Claim (Lemma) 2. Note that we must also have $head(e'_j) > b$ through the definition of *previousLesser*; otherwise $head(e'_j)$ would be v. So, there is a contradiction of the assumption that $head(e'_j) \leq b \leq head(e_{j+1}) - 1$. Thus, $head(e_j) \leq b \leq head(e'_j) - 1$ for some j and $distance(b,a)= sourceDistance(b,a) - 1 = j - 1$.

ii. A. Assume the antecedent holds. Also, to compute $distance(terminal(a),a)$, $terminal(a)$ is used as b'. Thus, Theorem 3 (ii) applies. As a result, $distance(terminal(a),a) = sourceDistance(terminal(a),a)$. Since $terminal(a)$ is outside a proper edge region as $startProperEdge(terminal(a)) = \infty$, $distance(b, terminal(a))= sourceDistance(b, terminal(a))$ by Case 1 of this theorem. Since $terminal(a)$ is not in a region of a $<$ edge contained in $ngTraversal(a,b)$ by the definition of $terminal(a)$ ($terminal(a)$ is a head of the last $<$ edge in the $nonProperPath(a)$), Corollary 1 applies. Thus, the distances are summed to get $distance(b,a)= distance(terminal(a),a) + distance(b, terminal(a)) = sourceDistance(terminal(a),a) + sourceDistance(b, terminal(a)) = sourceDistance(b,a)$.

B. • Assume the antecedent holds. As before, Theorem 3(ii) applies and $distance(terminal(a),a) = sourceDistance(terminal(a),a)$. Since $terminal(a)$ is in the same proper edge region that b is in through the fact that $startProperEdge(terminal(a)) \neq \infty$ and $b < nextGreater(startProperEdge(terminal(a)))$, $distance(b, terminal(a)) = sourceDistance(b, terminal(a)) = 0$ by Case 2 of this theorem. Again, Corollary 1 holds. As such, the distances are summed to

** The label of e'_j includes ℓ_a and unless $e'_j = (u,v)$, the label of (u,v) does not include ℓ_a by Claim 5.

get $distance(b,a) = distance(terminal(a),a) = sourceDistance(terminal(a),a) = sourceDistance(b,a)$.

- Assume the antecedent holds. As before, Theorem 3(ii) applies and $distance(terminal(a),a)= sourceDistance(terminal(a),a)$. b is not in the proper edge region that $terminal(a)$ is in and there is no non-proper edge e leaving the proper edge region that $terminal(a)$ is in such that $tail(e) \geq terminal(a)$. This is due to the fact that $startProperEdge(terminal(a)) \neq \infty$ and $b \geq nextGreater(startProperEdge(terminal(a)))$. Thus, $distance(b, terminal(a)) = sourceDistance(b, terminal(a))-1$ by Case 3(a) of this theorem. Again, Corollary 1 applies. Hence, the distances are summed to get $distance(b,a) = distance(terminal(a),a)+distance(b, terminal(a)) = sourceDistance(terminal(a),a)+sourceDistance(b, terminal(a))-1= sourceDistance(b,a)-1$. □

Note that the cases in the formal version of the query algorithm correspond exactly to those cases described in the overview. We are aware that Case 1 is actually a special case of Case 3(b)(i)A and 3(b)(ii)A when the opening condition of Case 3(b) that $startProperEdge(a) \neq \infty$ and $b \geq nextGreater(startProperEdge(a))$ is omitted. This is true because proper edges have a label of [0,0]. Since the numbers in the labels are all positive, it is always the case that $first(labelNextGreater(a)) \leq last(labelPreviousLesser(b))$, if a is outside a proper edge region. Thus, $distance(b,a) = sourceDistance(b,a)$ no matter where b is before $terminal(a)$. Since a is outside a proper edge region so is $terminal(a)$, and consequently $startProperEdge(terminal(a)) = \infty$. If b is at or after the head of the last proper edge which is $terminal(a)$ in this instance, it is still the case that $distance(b,a) = sourceDistance(b,a)$. However, it serves an illustrative purpose to keep these cases separate.

Observe that it is not essential that $first(labelNextGreater(a))$ is used as opposed to some other number in the range of $labelNextGreater(a)$ as long as $last(labelPreviousLesser(b))$ is utilized. Label ranges of different non-proper paths present in the same proper edge region are non-overlapping by Claim 5 and since $b < terminal(a)$, a non-proper edge of the $nonProperPath(a)$ is present in the proper edge region that b is in. In addition, the sizes of ranges of $<$ edges of the $nonProperPath(a)$ may increase but not decrease in the direction from the source to the sink. As a result, if $first(labelNextGreater(a)) \leq last(labelPreviousLesser(b))$, then we also have $last(labelNextGreater(a)) \leq last(labelPreviousLesser(b))$. However, $first(labelNextGreater(a))$ is used for consistency reasons.

Corollary 2 *Either* $distance(b,a) = sourceDistance(b,a)$ *or* $distance(b,a)= sourceDistance(b,a) - 1$.

Proof: This is a direct consequence of Theorem 5. □

Theorem 6 *The querying algorithm expressed in Theorem 5 runs in O(1) time.*

Proof: Each of the three steps of the query algorithm involves the look up and comparison of a constant number of fields, and thus the query algorithm also takes constant time. □

6 Extensions to the MAX DIST Problem

Two basic extensions to the MAX DIST problem have been considered, namely the 2-value MAX DIST problem and the ramifications of updates to the chain on the structures of the problem solved in the previous section. It is interesting to note that the solution of the MAX DIST problem applies to the variant of the problem where $<$ edges have a weight of a positive number w_1. Naturally, every occurrence of "1" in the query algorithm must be changed to w_1. This is called the 1-value MAX DIST problem. Extending further to the 2-value MAX DIST problem, we allow two possible weights for $<$ edges, w_1 and w_2. We have developed an O(1) query algorithm given an O(n) preprocessing step for some restricted cases of the 2-value MAX DIST problem. Updates to the chain that include adding a vertex, along with the insertion and deletion of certain restricted edges have been provided for with a cost in terms of time of O($lg\ n$) per operation. Then, querying is also degraded to O($lg\ n$) time. See [Grü99] for details of this and an elaboration of the allowed updates.

7 Conclusion and Open Problems

In this paper, we have seen how an O(1) time solution to the MAX DIST problem can be achieved after O(n) preprocessing. The query algorithm has been explained in detail. In addition, the necessary preprocessing that includes the labelling algorithm has also been explicated in some depth. These findings are significant, since the MAX DIST problem may be relevant to important applications ranging from crew scheduling to production optimization in manufacturing and product synthesis.

Here is a list of some areas of future research concerning matters mentioned in this paper:

- What other real world applications of the MAX DIST problem exist?
- Can the labelling scheme characterized in this paper be applied to other problems?
- Is there also a way to solve the 2-value MAX DIST problem for every case and with no error bound in O(1) time after O(n) preprocessing time? Can this be done for the k-value MAX DIST problem as well?
- Is it possible to increase the scope of allowed updates while maintaining O($lg\ n$) time complexity?
- Is it possible to answer queries about the maximum distance or to put it differently, the longest weighted path between two vertices, in a series-parallel graph or even a local graph in O(1) time after O(n) preprocessing time?

References

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communication of the ACM*, 26(1):832–843, 1983.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[Cuk94] Diana Cukierman. Formalizing the temporal domain with hierarchical structures of time units. Master's thesis, School of Computing Science, Simon Fraser University, 1994.

[DG96] James P. Delgrande and Arvind Gupta. A representation for efficient temporal reasoning. In *American Association for Artificial Intelligence Conference*, Portland, Oregon, August 1996. AAAI Press.

[DG98] Jim P. Delgrande and Arvind Gupta. Revising timegraph II. In *Twelfth Canadian Conference on Artificial Intelligence*, Vancouver, June 1998.

[Dor92] Jürgen Dorn. Temporal reasoning in sequence graphs. In *Proceedings of the Tenth National Conference of the American Association for Artificial Intelligence (AAAI-92)*, pages 735–740. AAAI Press, 1992.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[Grü99] Gabrielle Assunta Grün. The maximum distance problem. Master's thesis, School of Computing Science, Simon Fraser University, 1999.

[GS93] Alfonso Gerevini and Lenhart Schubert. Efficient temporal reasoning through timegraphs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 648–654, Chamb'ery, Savoie, France, 1993.

[GS95] Alfonso Gereveni and Lenhart Schubert. Efficient algorithms for qualitative reasoning about time. *Artificial Intelligence*, pages 207–248, 1995.

[GSS94] Alfonso Gerevini, Lenhart Schubert, and Stephanie Schaeffer. The temporal reasoning systems timegraph I-II. Technical Report 93-4, Computer Science Department, University of Rochester, Rochester, NY 14627, USA, 1994.

[MA89] Ghallab M. and A. Mounir Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 1297– 1303, 1989.

[MS90] Stephanie Miller and Lenhart K. Schubert. Time revisited. *Computational Intelligence*, 6:108–118, 1990.

[SPT87] Lenhart K. Schubert, Mary Angela Papalaskaris, and Jay Taugher. Accelerating deductive inference: Special methods for taxonomies, colours, and times. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 187–220. Springer-Verlag, 1987.

[VADG98] Tim Van Allen, James P. Delgrande, and Arvind Gupta. Point-based approaches to qualitative temporal reasoning. Technical Report TR 1998-16, School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada, 1998.

[VKvB90] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufman, San Mateo, CA, 1990.

[YA93] Ed Yampratoom and James Allen. Performance of temporal reasoning systems. Technical Report 93-1, Computer Science Department, University of Rochester, Rochester, NY 14627, USA, 1993.