1993

An Optimal Algorithm for Shortest Paths on Weighted Interval and Circular-Arc Graphs with Applications

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Danny Z. Chen

D. T. Lee

Report Number:

93-005

AN OPTIMAL ALGORITHM FOR SHORTEST PATHS ON WEIGHTED INTERVAL AND CIRCULAR-ARC GRAPHS, WITH APPLICATIONS

Mikhail J. Atallah
Danny Z. Chen
D. T. Lee

CSD-TR-93-005
January 1993
(Revised 2/93)
(Revised 3/93)

An Optimal Algorithm for Shortest Paths on Weighted Interval and Circular-Arc Graphs, with Applications

Mikhail J. Atallah*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
mja@cs.purdue.edu

Danny Z. Chen†
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
chen@cse.nd.edu

D. T. Lee‡
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208
dtlee@eecs.nwu.edu

Abstract

We give the first linear-time algorithm for computing single-source shortest paths in a weighted interval or circular-arc graph, when we are given the model of that graph, i.e., the actual weighted intervals or circular-arcs *and* the sorted list of the interval endpoints. Our algorithm solves this problem optimally in $O(n)$ time, where n is the number of intervals or circular-arcs in a graph. An immediate consequence of our result is an $O(qn+n\log n)$ time algorithm for the minimum-weight circle-cover problem, where q is the minimum number of arcs crossing any point on the circle; the $n\log n$ term in this time complexity is from a preprocessing sorting step when the sorted list of endpoints is not given as part of the input. The previous best time bounds were $O(n\log n)$ for this shortest paths problem, and $O(qn\log n)$ for the minimum-weight circle-cover problem. Thus we improve the bounds of both problems. More importantly, the techniques we give hold the promise of achieving similar $\log n$-factor improvements in other problems on such graphs.

*Research supported in part by the Leonardo Fibonacci Institute in Trento, Italy, by the Air Force Office of Scientific Research under Contract AFOSR-90-0107, and by the National Science Foundation under Grant CCR-9202807.

†Research supported in part by the Leonardo Fibonacci Institute in Trento, Italy.

‡Research supported in part by the Leonardo Fibonacci Institute in Trento, Italy and by the National Science Foundation under Grant CCR-8901815.

1 Introduction

Given a weighted set S of n intervals on a line, a *path* from interval $I \in S$ to interval $J \in S$ is a sequence $\sigma = (J_1, J_2, \ldots, J_k)$ of intervals in S such that $J_1 = I$, $J_k = J$, and J_i and J_{i+1} overlap for every $i \in \{1, \ldots, k-1\}$. The *length* of σ is the sum of the weights of its intervals, and σ is a *shortest path* from I to J if it has the smallest length among all possible I-to-J paths in S. The single-source shortest paths problem is that of computing a shortest path from a given "source" interval to all the other intervals. Our algorithm solves this shortest paths problem on interval and circular-arc graphs optimally in $O(n)$ time, when we are given the model of such a graph, i.e., the actual weighted intervals or circular-arcs *and* the sorted list of the interval endpoints. A node of an interval (resp., circular-arc) graph corresponds to an interval (resp., circular-arc) and an edge is between two nodes in the graph iff the two intervals (resp., circular-arcs) corresponding to these nodes intersect each other. Note that an interval or circular-arc graph with n nodes can have $O(n^2)$ edges. Our algorithm achieves the optimal $O(n)$ time bound by exploiting several geometric properties of this problem and by making use of the special UNION-FIND structure of [5].

One of the main applications of this shortest paths problem is to the minimum-weight circle-cover problem [9, 3, 2, 8], whose definition we briefly review: Given a set of weighted circular-arcs on a circle, choose a minimum-weight subset of the circular-arcs whose union covers the circle. It is known [3] that the minimum-weight circle-cover problem can be solved by solving q instances of the previously mentioned single-source shortest paths problem, where q is the minimum number of arcs crossing any point on the circle (in [3], a minimum-weight circle-cover is found in $O(qn^2)$ time). It is the circle-cover problem that has the main practical applications, and the study of this shortest-paths problem has mainly been for the purpose of solving the circle-cover problem. However, interval graphs and circular-arc graphs do arise in VLSI design, scheduling, biology, traffic control, and other application areas [4, 6, 7], so that our shortest paths result may be useful in other optimization problems. More importantly, our approach holds the promise of shaving a $\log n$ factor from the time complexity of other problems on such graphs.

We henceforth assume that the intervals are given sorted by their left endpoints, and also sorted by their right endpoints. This is not a limiting assumption in the case of the main application of the shortest paths problem, which is the minimum-weight circle-cover problem. In the latter problem, an $O(n \log n)$ preprocessing sorting step is cheap compared to the best previous bound for solving that problem, which was $O(qn \log n)$ [8] (by using q times the subroutine for solving the shortest paths problem, at a cost of $O(n \log n)$ time each). Using our shortest paths algorithm, the minimum-weight circle-cover problem is solved in $O(qn + n \log n)$ time, where the $n \log n$ term is from the preprocessing sorting step when the sorted list of endpoints is not given as part of the input. Therefore, in order to establish the bound we claim for the minimum-weight circle-cover problem, it suffices to give a linear-time algorithm for the shortest paths problem on interval graphs. The linear-time solution to the shortest paths problem on circular-arc graphs makes use of the solution to the shortest paths problem on interval graphs. Therefore, we mainly focus on the problem of solving, in linear time, the shortest paths problem on interval graphs.

We also henceforth assume, without loss of generality, that we are computing the shortest paths from the source interval to only those intervals whose right endpoints are to the right of the right endpoint of the source; the same algorithm that solves this case can, of course, be used to solve the case for the shortest paths to intervals whose left endpoints are to the left of the left endpoint of the source. Clearly we need not worry about paths to intervals whose right endpoints are covered by the source since the problem is trivial for those intervals – the length of the shortest path is simply the sum of the weight of the source plus the weight of the destination.

We consider the shortest paths problem on interval (resp., circular-arc) graphs in which the weights of the intervals (resp., circular-arcs) are nonnegative. The minimum-weight circle-cover problem [3], however, does allow circular-arcs to have negative weights. Bertossi [3] has already given a reduction of any minimum-weight circle-cover problem with both negative and nonnegative weights to one with only nonnegative weights (to which the algorithm for computing shortest paths in interval graphs with nonnegative weights is applicable). Therefore it suffices to solve the shortest paths problem on interval graphs for the case of nonnegative weights. Bertossi's reduction introduces zero-weight intervals, so it is important to be able to handle problems with zero-weight intervals.

We only show how to compute the lengths of shortest paths. Our algorithm can be easily modified to handle the computation for actual shortest paths and shortest path trees, in $O(n)$ time and $O(n)$ space.

In the next section, we introduce some terminology needed in the rest of the paper. Sections 3 and 4 consider the special case of the shortest paths problem on interval graphs with only positive weights. In particular, Section 3 presents a preliminary suboptimal algorithm which illustrates our main idea and observations, and Section 4 shows how to implement various computation steps of the preliminary algorithm so that it runs optimally in linear time. Section 5 gives a linear-time reduction that reduces the nonnegative weight case to the positive weight case, and it shows how to use the solution to the shortest paths problem on interval graphs to obtain the solution to that on circular-arc graphs.

2 Terminology

In this section, we introduce some additional terminology.

We say that an interval I *contains* another interval J iff $I \cap J = J$. We say that I *overlaps* with J iff their intersection is not empty, and that I *properly overlaps* with J iff they overlap but neither one contains the other.

An interval I is typically defined by its two endpoints, i.e., $I = [a, b]$ where $a \leq b$ and a (resp., b) is called the *left* (resp., *right*) endpoint of I. A point x is *to the left* (resp., *right*) of interval $I = [a, b]$ iff $x < a$ (resp., $b < x$).

We assume that the input set S consists of intervals $I_1, \ldots, I_n$, where $I_i = [a_i, b_i]$, $b_1 \leq b_2 \leq \cdots \leq b_n$, and that the weight of each interval I_i is $w_i \geq 0$. To avoid unnecessarily cluttering the exposition, we assume that the intervals have distinct endpoints, that is, $i \neq j$ implies $a_i \neq a_j$, $b_i \neq b_j$, $a_i \neq b_j$, and $b_i \neq a_j$ (the algorithm for nondistinct endpoints is a trivial modification of the one we give).

Definition 1 *We use S_i to denote the subset of S that consists of intervals $I_1, I_2, \ldots, I_i$. We assume, without loss of generality, that the union of all the I_i's in S covers the portion of the line from a_1 to b_n. We also assume, without loss of generality, that the source interval is I_1.*

Observe that for a set S^* of intervals, the union of all the intervals in S^* may form more than one connected component. If for two intervals I' and I'' in S^*, I' and I'' respectively belong to two different connected components of the union of the intervals in S^*, then there is no path between I' and I'' that uses only the intervals in S^*.

3 A Preliminary Algorithm

This section gives a preliminary, $O(n \log \log n)$ time (hence suboptimal) algorithm for the special case of the shortest paths problem on intervals with positive weights. This should be

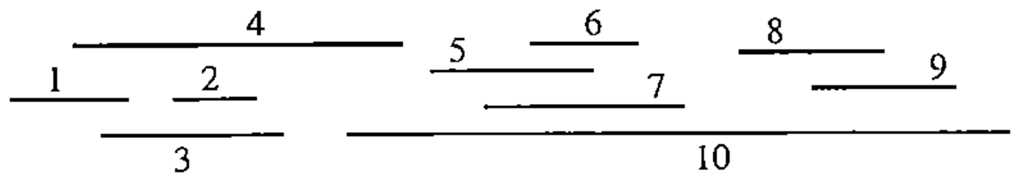

Figure 1: For $i = 1, 2, \ldots, 10$, $w_i = 15, 12, 13, 17, 17, 19, 21, 13, 15, 18$, respectively.

viewed as a "warm-up" for the next section, which will give an efficient implementation of some of the steps of this preliminary algorithm, resulting in the claimed linear-time bound. In Section 5, we point out how the algorithm for positive-weight intervals can also be used to solve problems with nonnegative-weight intervals.

We begin by introducing definitions that lead to the concept of an *inactive* interval in a subset S_i, then proving lemmas about it that are the foundation of the preliminary algorithm.

Definition 2 *An* extension *of S_i is a set S' that consists of S_i and one or more intervals (not necessarily in S) whose right endpoints are larger than b_i. (There are, of course, infinitely many choices for such an S'.)*

Definition 3 *An interval I_k in S_i ($k \leq i$) is* inactive *in S_i iff for* every *extension S' of S_i, the following holds: Every $J \in S' - S_i$ for which there is an I_1-to-J path in S' has no shortest I_1-to-J path in S' that uses I_k. An interval of S_i which is not inactive in S_i is said to be* active *in S_i.*

Intuitively, I_k is inactive in S_i if the other intervals in S_i are such that, as far as any interval J with right endpoint larger than b_i is concerned, I_k is "useless" for computing a shortest I_1-to-J path (in particular, this is true for $J \in \{I_{i+1}, \ldots, I_n\}$). In Figure 1, I_2 is inactive in S_4, I_3 is active in S_4, I_5 is inactive in S_5, I_9 is inactive in S_{10}, and I_{10} is active in S_{10}.

Observe that an interval I_k that is active in S_i, $k \leq i$, may be inactive for an S_j with $j > i$, but is certainly active for any S_j with $k \leq j \leq i$. On the other hand, an interval I_k which is inactive for S_i, $k \leq i$, is also inactive for every S_j with $j > i$.

Note that I_i is active in S_i iff there is an I_1-to-I_i path in S_i (i.e., if $\cup_{1 \leq k \leq i} I_k$ covers the portion of the line from a_1 to b_i).

Lemma 1 *The union of all the active intervals in S_i covers a contiguous portion of the line from a_1 to some b_j, where b_j is the rightmost endpoint of any active interval in S_i.*

Proof. An immediate consequence of the fact that if I_k, $k \leq i$, is active in S_i, then there is an I_1-to-I_k path in S_i. This is because if there is an I_1-to-I_k path in S_i, then there is a shortest I_1-to-I_k path in S_i, implying that every constituent interval of such a shortest I_1-to-I_k path is active in S_i. □

Definition 4 *Let $label_j(i)$, $j \geq i$, denote the length of a shortest I_1-to-I_i path in S that does not use any I_k for which $k > j$. By convention, if $j < i$, then $label_j(i) = +\infty$.*

Observe that for all i, $label_1(i) \leq label_2(i) \leq \cdots \leq label_n(i)$. For an $I_k \in S_i$, if there is no I_1-to-I_k path in S_i, then obviously $label_i(j) = +\infty$, for every $j = k, k+1, \ldots, i$. In Figure 1, $label_9(7) = +\infty$, but $label_{10}(7) = 71$.

Our algorithm is based on the following lemmas.

Lemma 2 *If $i > k$ and $label_i(i) < label_i(k)$, then I_k is inactive in S_i.*

Proof. Since $label_i(i) < label_i(k)$, $label_i(i)$ is not $+\infty$. Hence there is an I_1-to-I_i path in S_i, and there is an I_1-to-I_k path in S_i. Because $label_i(i) < label_i(k)$, it follows that there is a shortest I_1-to-I_i path in S_i that does not use I_k: The union of the intervals on that I_1-to-I_i path contains I_k (because $i > k$), and hence I_k is "useless" for any $J \in S' - S_i$ where S' is an extension of S_i. □

The following are immediate consequences of Lemma 2.

Corollary 1 *Let $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$ be the active intervals in S_i, $j_1 < j_2 < \cdots < j_k$. Then $label_i(j_1) \leq label_i(j_2) \leq \cdots \leq label_i(j_k)$.*

Figure 2 illustrates Corollary 1. Note that the right endpoints of the active intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$ in S_i are in the same sorted order as that of their labels $label_i(j_1)$, $label_i(j_2)$, $\ldots$, $label_i(j_k)$. Their left endpoints, however, are not necessarily in such a sorted order (in Figure 2, the left endpoints of the intervals are omitted, indicated by marks "...").

Corollary 2 *If I_i contains I_k (hence $i > k$) and $label_i(k) > label_i(i)$, then I_k is inactive in S_i.*

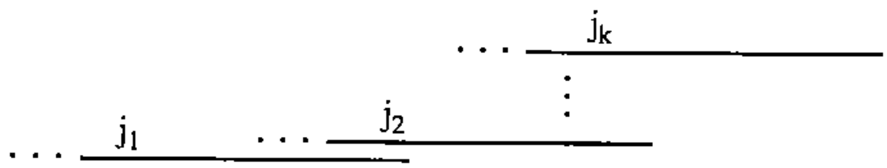

Figure 2: Illustrating Corollary 1: $label_i(j_1) \leq label_i(j_2) \leq \cdots \leq label_i(j_k)$.

Lemma 3 *If $i > k$ and $label_i(i) < label_{i-1}(k)$, then I_k is inactive in S_i.*

Proof. That $label_i(i) < label_{i-1}(k)$ implies that $label_i(i)$ is not $+\infty$. Hence there is an I_1-to-I_i path in S_i, and there is an I_1-to-I_k path in S_i. There are two cases to consider. (i) The shortest I_1-to-I_k path in S_i does not need to use I_i. Then $label_{i-1}(k) = label_i(k)$, and hence $label_i(i) < label_i(k)$. By Lemma 2, I_k is inactive in S_i. (ii) The shortest I_1-to-I_k path in S_i does use I_i. Then $label_i(k) \geq label_i(i) + w_k > label_i(i)$ (since $w_k > 0$). Again by Lemma 2, I_k is inactive in S_i. □

Lemma 4 *If interval I_k, $k > 1$, does not contain any b_j ($j < k$) such that I_j is active in S_{k-1}, then I_k is inactive in S_i for every $i \geq k$.*

Proof. It suffices to prove that I_k is inactive in S_k. Suppose that I_k is active in S_k. Then by Lemma 1, the union of all the active intervals in S_k covers the contiguous portion of the line from a_1 to b_k (note that b_k is the rightmost endpoint of any interval in S_k). This implies that I_k contains the right endpoint of at least one active interval in S_k other than I_k. But all the intervals in S_{k-1} ($= S_k - \{I_k\}$) that I_k intersects are inactive in S_{k-1}, and hence they remain inactive in S_k, contradicting to that I_k intersects some active intervals in S_k other than I_k. □

We first give an overview of the algorithm. The algorithm scans the intervals in the order $I_1, I_2, \ldots, I_n$ (i.e., the scan is based on the increasing order of the sorted right endpoints of the intervals in S). When the scan reaches I_i, the following must hold before the scan can proceed to I_{i+1}:

(1) All the active intervals in S_i are stored in a tree T.

(2) All the inactive intervals in S_i have been marked as such (possibly at an earlier stage, when the scan was at some $I_{i'}$ with $i' < i$).

(3) If I_k ($k \leq i$) is active in S_i, then the correct $label_i(k)$ is known.

If we can maintain the above invariants, then clearly when the scan terminates at I_n, we already know the desired $label_n(i)$'s for all I_i's which are active in S_n. A postprocessing step will then compute, in linear time, the correct $label_n(i)$'s of the inactive I_i's in S_n (more on this later).

The details of the preliminary algorithm follow next. In this algorithm, the *right* endpoints of the active intervals are maintained in the leaves of the tree structure T, one endpoint per leaf, in sorted order.

1. Initialize T to contain I_1.

2. For $i = 2, 3, \ldots, n$, do the following. Perform a search in T for a_i. This gives the smallest b_j in T that is $> a_i$. If no such b_j exists, then (by Lemma 4) mark I_i as being inactive and proceed to $i + 1$. So suppose such a b_j exists. Set $label_i(i) = label_{i-1}(j) + w_i$, and note that this implies that I_j remains active in S_i and has the same label as in S_{i-1}, i.e., $label_i(j) = label_{i-1}(j)$. Next, insert I_i in T (of course b_i is then in the rightmost leaf of T). Then repeatedly check the leaf for I_k which is immediately to the left of the leaf for I_i in T, to see whether I_k is inactive in S_i (by Lemma 3, i.e., check whether $label_{i-1}(k) < label_i(i)$), and, if I_k is inactive, then mark it as such, delete it from T, and repeat with the leaf made adjacent to I_i by the deletion of I_k. Note that more than one leaf of T may be deleted in this fashion, but that the deletion process stops short of deleting I_j itself, because it is I_j that gave I_i its current label (i.e., $label_i(i) = label_{i-1}(j) + w_i \geq label_{i-1}(j)$). Of course any I_ℓ whose leaf in T is *not* deleted is in fact active in S_i and already has the correct value of $label_i(\ell)$: It is simply the same as $label_{i-1}(\ell)$ and we need not explicitly update it (the fact that this updating is implicit is important, as we cannot afford to go through all the leaves of T at the iteration for each i).

   When Step 2 terminates (at $i = n$), we have the values of the $label_n(\ell)$'s for all the active I_ℓ in S_n. The next step obtains the values of the $label_n(\ell)$'s for the other intervals (those that are inactive in S_n).

3. For every inactive I_i in S_n, find the smallest right endpoint $b_j > a_i$ such that I_j is active in S_n, and set $label_n(i) = label_n(j) + w_i$. Note that by Lemma 1, such an I_j exists and it intersects I_i. This step can be easily implemented by a right-to-left scan of the sorted list of all the endpoints.

The correctness of this algorithm easily follows from the definitions, lemmas, and corollaries preceding it. Note that although a particular iteration in Step 2 may result in many deletions from T, overall there are less than n such deletions. The time complexity of this algorithm is $O(n \log n)$ if we implement T as a 2-3 tree [1], but $O(n \log\log n)$ if we use the data structure of Van Emde Boas [11] (the latter would require normalizing all the $2n$ sorted endpoints so that they are integers between 1 and $2n$). The next section gives an $O(n)$ time implementation of the above algorithm. Note that the main bottleneck is Step 2, since the scan needed for Step 3 obviously takes linear time.

4 A Linear Time Implementation

As observed earlier, the main bottleneck is Step 2 of the preliminary algorithm given in the previous section. We shall implement essentially the same algorithm, but without using the tree T. Instead, we use a UNION-FIND structure [5] where the elements of the sets are integers in $\{1, \ldots, n\}$, with integer i corresponding to interval I_i. Initially, each element i is in a singleton set also named i, that is, initially set i is $\{i\}$. (We often call a set whose name is integer i as set i, with the understanding that set i may contain other elements than i.) During the execution of Step 2, we maintain the following invariants (assume we are at index i in Step 2):

(1) To each currently active interval I_j corresponds a set named j. If $I_{i_1}, I_{i_2}, \ldots, I_{i_k}$ are the active intervals in S_i, $i_1 < i_2 < \cdots < i_k$, then for every $i_j \in \{i_1, i_2, \ldots, i_{k-1}\}$, the indices of the inactive intervals $\{I_\ell \mid i_j < \ell < i_{j+1}\}$ are all in the set whose name is i_{j+1}. Set i_{j+1} consists of the indices of the above-mentioned inactive intervals, and also of the index i_{j+1} of the active interval $I_{i_{j+1}}$. Note that since I_1 is always active, $i_1 = 1$ in the above discussion, and the set whose name is 1 is a singleton (recall that a preprocessing step has eliminated intervals whose right endpoints are contained in interval I_1). The next invariant is about intervals that are inactive and do not overlap with any active interval.

(2) Let $Loose(S_i)$ denote the subset of the inactive intervals in S_i that do not overlap with any active interval in S_i. In Figure 1, the active intervals in S_9 are I_1, I_3, I_4, and $Loose(S_9)$ consists of intervals $I_5, I_6, \ldots, I_9$. Observe that, based on Lemma 1, every interval in $Loose(S_i)$ is to the right of the union of the active intervals in S_i; furthermore, $Loose(S_i)$ is nonempty iff $I_i \in Loose(S_i)$. If $Loose(S_i)$ is not empty, then let $CC_1, CC_2, \ldots, CC_t$ be the connected components of $Loose(S_i)$: There is a set named j_l for every such CC_l, where I_{j_l} is the *rightmost* interval in CC_l (I_{j_l} is the interval in CC_l having the largest right endpoint); we say that such an inactive I_{j_l} is *special inactive*. The (say) μ elements in set j_l correspond to the μ intervals in CC_l; more specifically, they are the contiguous subset of indices $\{j_l - \mu + 1, j_l - \mu + 2, \ldots, j_l - 1, j_l\}$. Note that $j_l - \mu$ is the set named j_{l-1} if $1 < l \leq t$, and that $j_t = i$.

   In Figure 1, for $i = 9$, $CC_1 = \{I_5, I_6, I_7\}$, $CC_2 = \{I_8, I_9\}$, and the special inactive intervals are I_7 and I_9.

(3) An auxiliary stack contains the active intervals $I_{i_1}, I_{i_2}, \ldots, I_{i_k}$ mentioned in item (1) above, with I_{i_k} at the top of the stack. We call it the *active* stack.

   In Figure 1, for $i = 9$, the active stack contains I_1, I_3, I_4 (with I_4 at the top of the stack).

(4) Another auxiliary stack contains the special inactive intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_t}$ mentioned in item (2) above, with I_{j_t} at the top of the stack. We call it the *special inactive* stack.

   In Figure 1, for $i = 9$, the special inactive stack contains I_7, I_9 (with I_9 at the top of the stack).

A crucial point is how to implement, in Step 2, the search for b_j using a_i as the key for the search. This is closely tied to the way that the above invariants (1)–(4) are maintained. It makes use of some preprocessing information that is described next.

Definition 5 *For every I_i, let $Succ(I_i)$ be the smallest index ℓ such that $a_i < b_\ell$, i.e., $b_\ell = \text{Min}\{b_r \mid I_r \in S, a_i < b_r\}$.*

In Figure 1, $Succ(I_5) = 5$, $Succ(I_9) = 8$, and $Succ(I_{10}) = 4$.

Note that $\ell \leq i$, and that $\ell = i$ occurs when I_i does not contain any b_r other than b_i. Also, observe that the definition of the $Succ$ function is static (it does not depend on which intervals are active). The *Succ* function can easily be precomputed in linear time by scanning right-to-left the sorted list of all the $2n$ interval endpoints.

The significance of the *Succ* function is that, in Step 2, instead of searching for b_j using a_i as the key for the search, we simply do a FIND($Succ(I_i)$): Let j be the set name returned by this FIND operation. We distinguish 3 cases.

1. If $j = i$, then surely I_i does not overlap with any interval in S_{i-1} and it is inactive in S_i (by Lemma 4). We simply mark I_i as being special inactive, push I_i on the special inactive stack, and move the scan of Step 2 to index $i + 1$.

   In Figure 1, this happens for $i = 2$, $i = 5$, and $i = 8$.

2. If $j < i$ and I_j is active in S_{i-1}, we set $label_i(i) = label_{i-1}(j) + w_i$. Then do the following updates on the two stacks:

   (a) We pop *all* the special inactive intervals I_{i_l} from their stack and, for each such I_{i_l}, we do UNION(i_l, i), which results in the disappearance of set i_l and the merging of its elements with set i; set i retains its old name.

   In Figure 1, for $i = 10$, this results in the disappearance of sets 7 and 9, and the merging of their contents with set 10.

   (b) We repeatedly check whether the top of the active stack, I_{i_k}, is going to become inactive in S_i because of I_i (that is, because $label_i(i) < label_{i-1}(i_k)$). If the outcome of the test is that I_{i_k} becomes inactive, then we do UNION(i_k,i), pop I_{i_k} from the active stack, and continue with $I_{i_{k-1}}$, etc. If the outcome of the test is that I_{i_k} is active in S_i, then we keep it on the active stack, push I_i on the active stack, and move the scan of Step 2 to index $i + 1$.

   In Figure 2, if I_i is active in S_i, $j = j_1$, and $label_i(i) < label_{i-1}(j_2)$, then the sets $j_2, j_3, \ldots, j_k$ disappear and their contents get merged with set i.

3. If $j < i$ and I_j is special inactive in S_{i-1}, then I_i does not overlap with any active interval in S_{i-1} and it is inactive in S_i (by Lemma 4). But, I_i does overlap with one or more inactive intervals in S_{i-1}, including the special inactive interval I_j; more precisely, I_i overlaps with some connected components of $Loose(S_{i-1})$ whose rightmost intervals are contiguously stored in the stack of special inactive intervals. Let these connected components with that I_i overlaps be called, in left to right order, $C_1, C_2, \ldots, C_h$. The

rightmost interval of C_1 is I_j. Let $I_{r_2}, I_{r_3}, \ldots, I_{r_h}$ be the rightmost intervals of (respectively) $C_2, C_3, \ldots, C_h$ (of course $I_{r_h} = I_{i-1}$). Observe that the top h intervals in the stack of special inactive intervals are $I_j, I_{r_2}, \ldots, I_{r_h}$, with I_{r_h} $(= I_{i-1})$ on top. Because of I_i, all of these h intervals will become inactive in S_i (whereas they were special inactive in S_{i-1}). Their h sets (corresponding to $C_1, C_2, \ldots, C_h$) must be merged into a new, single set having I_i as its rightmost interval. I_i is special inactive in S_i. This is achieved by:

(a) Popping $I_{r_h}, \ldots, I_{r_2}, I_j$ from the stack of special inactive intervals,

(b) performing UNION(r_h, i), UNION(r_{h-1}, i), ..., UNION(r_2, i), UNION(j, i), and

(c) pushing I_i on the special inactive stack.

Observe that the total number of the UNION and FIND operations performed by our algorithm is $O(n)$. It is well-known that a sequence of m UNION and FIND operations on n elements can be performed in $O(m\alpha(m+n,n)+n)$ time [10], where $\alpha(m+n,n)$ is the (very slow-growing) functional inverse of Ackermann's function. Therefore, our algorithm runs within the same time bound. However, it is possible to achieve $O(n)$ time performance for our algorithm, by the following observations.

In our algorithm, every UNION operation involves two set names that are *adjacent* in the sorted order of the currently existing set names. That is, if L is the sorted list of the set names (initially L consists of all the integers from 1 to n), then a UNION operation always involves two adjacent elements of L. Thus the underlying UNION-FIND structure we use satisfies the requirements of the *static tree set union* in [5], in order to result in linear-time performance: It is the *linked list* $LL = (1, 2, \ldots, n)$, where the element in LL that follows element ℓ is $next(\ell) = \ell + 1$, for every $\ell = 1, 2, \ldots, n-1$ (the requirement in [5] is that the structure be a static tree). Note that the *next* function is static throughout our algorithm. The UNION operation in our algorithm is always of the form $unite(next(\ell), \ell)$, as defined in [5], that is, it concatenates two disjoint but *consecutive* sublists of LL into one contiguous sublist of LL. On this kind of structures, a sequence of m UNION and FIND operations on n elements can be performed in $O(m+n)$ time [5]. Therefore, the time complexity of our algorithm is $O(n)$.

5 Further Extensions

This section sketches how the shortest paths algorithm of the previous sections can be used to solve problems where intervals can have zero weight, and how it can be used to solve the version of the problem where we have circular-arcs rather than intervals on a line.

5.1 Zero-Weight Intervals

The astute reader will have observed that the definitions and the shortest paths algorithm of the previous sections can be modified to handle zero-weight intervals as well. However, doing so would unnecessarily clutter the exposition. Instead, we show in what follows that the shortest paths problem in which some intervals have zero weight can be reduced in linear time to one in which all the weights are positive. Not only does this simplify the exposition, but the reduction used is of independent interest.

Let $P1$ be the version of the problem that has zero-weight intervals, and let Z be the nonempty subset of S that contains all the zero-weight intervals of S. First, observe that in order to solve $P1$, it suffices to solve the problem $P2$ obtained from $P1$ by replacing every connected component CC of Z by a new zero-weight interval that is the union of the zero-weight intervals in CC (because the label of $I \in Z$ in $P1$ is the same as the label of $J = \cup_{I \in CC} I$ in $P2$). Hence it suffices to show how to solve $P2$. In what follows assume that we have already created, in $O(n)$ time, $P2$ from $P1$.

We next show how to obtain, from $P2$, a problem $P3$ such that (i) every interval in $P3$ has a positive weight (and therefore $P3$ can be solved by the algorithm of the previous sections), and (ii) the solution to $P3$ can be used to obtain a solution to $P2$.

Recall that, by the definition of $P2$, two zero-weight intervals in it cannot overlap. $P3$ is obtained from $P2$ by doing the following for each zero-weight interval $J = [a, b]$: "cut out" the portion of the problem in between a and b, that is, first erase, for every interval I of $P2$, the portion of I in between a and b, and then "pull" a and b together so they coincide in $P3$. This means that in $P3$, J has disappeared, and so has every interval J' that was contained in J. An interval J'' in $P2$ that contained J, or that properly overlapped with J, gets shrunk by the disappearance of its portion that used to overlap with J. For example, if we imagine that the situation in Figure 1 describes problem $P2$, and that J is (say) interval I_4 in Figure 1 (so I_4 has zero weight), then "cutting" I_4 results in the disappearance of I_2 and I_3 and the "bringing together" of I_1 and I_{10} so that, in the new situation, the right endpoint of I_1 coincides with the left endpoint of I_{10}.

Implementation Note: The above-described cutting-out process of the zero-weight intervals can be implemented in linear time by using a linked list to do the cutting and pasting. In particular, if in $P2$ an interval I of positive weight contains many zero-weight intervals $J_1, \ldots, J_k$, the cutting-out of these zero-weight intervals does *not* affect the representation we use for I (although in a geometric sense I is "shorter" afterwards, as far as the linked list representation is concerned, it is unchanged). This is an important point, since it implies that only the endpoints contained in a J_k are affected by the cutting-out of that J_k, and such an endpoint gets updated only once because it is not contained in any other zero-weight interval of $P2$ (recall that the zero-weight intervals of $P2$ are pairwise non-overlapping).

By definition, $P3$ has no zero-weight intervals. So suppose $P3$ has been solved by using the algorithm we gave in the earlier sections. The solution to $P3$ yields a solution to $P2$ in the following way.

- If an interval I is in $P3$ (i.e., I has not been cut out when $P3$ was obtained from $P2$), then its label in $P2$ is exactly the same as its label in $P3$.
- Let $J = [a, b]$ be a zero-weight interval which was cut out from $P2$ when $P3$ was created. (In $P3$, a and b coincide, so in what follows when we refer to "a in $P3$" we are also referring to b in $P3$.) For each such $J = [a, b]$, compute in $P3$ the smallest label of any interval of $P3$ that contains a: This is the label of J in $P2$. This computation can be done for all such J's by one linear-time scan of the endpoints of the active intervals for $P3$.
- Suppose I is a positive-weight interval of $P2$ that was cut out when $P3$ was created, because it was contained in a zero-weight interval J of $P2$. Then the label of I in $P2$ is equal to: (weight of I) + (label of J in $P2$).

5.2 Circular-Arcs

The version of the shortest paths problem where we have circular-arcs on a circle C instead of intervals on a straight line can be solved by two applications of the shortest paths algorithm for intervals: Suppose $I_1 = [a, b]$ is the "source" circular-arc, where a and b are now positions on circle C. (We use the convention of writing a circular-arc as a pair of positions on the circle such that, when going from the first position to the second position along the arc, we travel in the clockwise direction.)

It is not hard to see that the following linear-time procedure solves the shortest paths problem on circular-arc graphs.

- Create a problem on a straight line by "opening" circle C at a. That is, create an n-interval problem by starting at a and traveling clockwise along C, putting the intervals encountered during this trip on a straight line, until the trip is back at a. Intervals that contain a are not included twice in the straight-line problem: Only their first appearance on the clockwise trip is used, and they are "truncated" at a (so that on the line, they appear to begin at a, just like the source I_1). Then solve the straight-line problem so created, by using the algorithm for the interval case. The computation of this step gives each circular-arc a label.

- Repeat the above step with a playing the role of b, and "counterclockwise" playing the role of "clockwise".

- The correct label for a circular-arc is the smaller of the two labels, computed above, for the intervals corresponding to that arc.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[2] M. J. Atallah and D. Z. Chen. "An optimal parallel algorithm for the minimum circle-cover problem," *Information Processing Letters*, 32 (1989), pp. 159–165.

[3] A. A. Bertossi. "Parallel circle-cover algorithms," *Information Processing Letters*, 27 (1988), pp. 133–139.

[4] K. S. Booth and G. S. Lukeher. "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *Journal of Computer and System Sciences*, 13 (1976), pp. 335–379.

[5] H. N. Gabow and R. E. Tarjan. "A linear-time algorithm for a special case of disjoint set union," *Journal of Computer and System Sciences*, 30 (1985), pp. 209–221.

[6] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[7] U. I. Gupta, D.T. Lee, and J. Y.-T. Leung. "Efficient algorithms for interval graphs and circular-arc graphs," *Networks*, Vol. 12 (1982), pp. 459–467.

[8] O. H. Ibarra, H. Wang, and Q. Zheng. "Minimum cover and single source shortest path problems for weighted interval graphs and circular-arc graphs," *Proc. of the Thirtieth Annual Allerton Conference on Communication, Control, and Computing*, 1992, University of Illinois, Urbana, pp. 575–584.

[9] C. C. Lee and D. T. Lee. "On a circle-cover minimization problem," *Information Processing Letters*, 18 (1984), pp. 109–115.

[10] R. E. Tarjan. "A class of algorithms which require nonlinear time to maintain disjoint sets," set union algorithm," *Journal of Computer and System Sciences*, 18 (2) (1979), pp. 110–127.

[11] P. Van Emde Boas. "Preserving order in a forest in less than logarithmic time and linear space," *Information Processing Letters*, 6 (3) (1977), pp. 80–82.