Solving the All-Pair Shortest Path Query Problem on Interval and Circular-Arc Graphs

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 and Computer Engineering
Northwestern University
Evanston, IL 60208
dtlee@ece.nwu.edu

R. Sridhar
School of Computer Science
University of Oklahoma
Norman, OK 73019
sridhar@giants.ecn.uoknor.edu

Chandra N. Sekharan
Department of Mathematical and Computer Sciences
Loyola University of Chicago
Chicago, IL 60626
chandra@math.luc.edu

Abstract

In this paper, we study the following all-pair shortest path query problem: Given the interval model of an unweighted interval graph of n vertices, build a data structure such that each query on the shortest path (or its length) between any pair of vertices of the graph can be processed efficiently (both sequentially and in parallel). We show that, after sorting the input intervals by their endpoints, a data structure can be constructed sequentially in $O(n)$ time and $O(n)$ space; using this data structure, each query on the length of the shortest path between any two intervals can be answered in $O(1)$ time, and each query on the actual shortest path can be answered in $O(k)$ time, where k is the number of intervals on that path. Furthermore, this data structure can be constructed optimally in parallel, in $O(\log n)$ time using $O(n/\log n)$ CREW PRAM processors; each query on the actual shortest path can be answered in $O(1)$ time using k processors. Our techniques can be extended to solving the all-pair shortest path query problem on circular-arc graphs, both sequentially and in parallel, in the same complexity bounds. As an immediate consequence of our results, we improve by a factor of n the space complexity of the previously best known sequential all-pair shortest path algorithm for unweighted interval graphs.

*Supported in part by the National Science Foundation under Grant CCR-9623585.
†Supported in part by the National Science Foundation under Grants CCR-8901815 and CCR-9309743.

1 Introduction

A graph $G = (V, E)$ is an interval graph if there exists a set S of intervals on the real line such that there is a one-to-one correspondence between the vertices $v_i \in V$ and the intervals $I_i \in S$ in such a way that an edge $(v_i, v_j) \in E$ if and only if $I_i \cap I_j \neq \emptyset$. Such an interval set S is called a *model* of the interval graph G. Interval graphs arise in many application areas, such as VLSI design, scheduling, biology, traffic control, archeology, etc. See [10] for an extensive discussion on interval graphs.

It is already known that the problem of recognizing interval graphs can be solved sequentially in $O(|V| + |E|)$ time [6, 13]. In this paper, we assume that a model $S = \{I_1, I_2, \ldots, I_n\}$ of an n-vertex interval graph has been given as input to the problems that we consider.

Given the interval set S, 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 number of intervals on σ, 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. Let $d(I, J)$ denote the *distance* between intervals I and J, which is defined to be the length of a shortest I-to-J path minus 1. The all-pair shortest path problem on S is that of computing the shortest path between every pair of intervals in S. Note that on an arbitrary n-vertex graph, a sequential algorithm by Seidel [19] for the all-pair shortest path problem takes $O(M(n) \log n)$ time where $M(n)$ denotes the time-complexity for matrix multiplication involving small integers only. $M(n)$ is currently known to be $O(n^{2.376})$[9].

The above-mentioned all-pair shortest path problem on *unweighted* interval graphs was recently considered by Ravi *et al.* [18], and they gave a sequential algorithm that runs in $O(n^2)$ time and uses $O(n^2)$ space to store the lengths of the all-pair shortest paths. In this paper, we show that, given the interval model, the information for the all-pair shortest paths can be made available (both sequentially and in parallel) much more efficiently than [18], by solving the following *all-pair shortest path query* problem: Given the interval model of an unweighted interval graph of n vertices, build a data structure such that each query on the shortest path (or its length) between any pair of vertices of the graph can be processed efficiently. Our main results for the all-pair shortest path query problem are as follows: A data structure can be constructed sequentially in $O(n \log n)$ time if S is not given sorted and in $O(n)$ time if S is already sorted, and in $O(n)$ space; using this data structure, each query on the length of the shortest path between two intervals in S can be answered in $O(1)$ time, and each query on the actual shortest path can be answered in $O(k)$ time, where k is the number of intervals on that path. Furthermore, this data structure can be constructed optimally in parallel, in $O(\log n)$ time using $O(n)$ processors if S is not sorted and using $O(n/\log n)$ processors otherwise; each query on the actual shortest path can be answered in $O(1)$ time using k processors. This data structure also enables us to answer very efficiently the *geometric* all-pair shortest path queries, which are defined as follows: Given two arbitrary query points p and q on the real line, find the shortest path $SP(p,q)$, such that $SP(p,q)$ uses only the intervals in S and that the union of the intervals on $SP(p,q)$ contains both p and q. For two arbitrary query points, the length query can be answered in $O(\log n)$ time using one processor (both sequentially and in parallel), and the path query can be answered in $O(\log n + k)$ time sequentially and in $O(\log n)$ time using $O(\lceil k/\log n \rceil)$ processors in parallel. Our techniques can be extended to circular-arc graphs; hence all our results on interval graphs can also be obtained on circular-arc graphs, in the same complexity bounds. Note that our sequential algorithm, when applied to the all-pair shortest path problem on unweighted interval graphs, improves by a factor of n the space complexity of the algorithm by Ravi *et al.* [18].

There has been some interesting work on shortest path problems on interval and circular-arc graphs. On interval and circular-arc graphs in which weights are associated with the intervals/circular-arcs, Ibarra *et al.* [12] gave an $O(n \log n)$ time algorithm for solving sequentially the *single-source* shortest path problem. Using the algorithm in [12], the all-pair shortest path problem on these weighted interval and circular-arc graphs can be solved in $O(n^2 \log n)$ time. Very recently, Atallah *et al.* [3] solved the single-source shortest path problem on these weighted interval and circular-arc graphs in $O(n)$ time if the intervals/circular-arcs are given sorted, and hence also solved the weighted all-pair shortest path problem in the optimal $O(n^2)$ time and $O(n^2)$ space. Parallel algorithms for the single-source shortest path problem on unweighted interval graphs have been studied. Atallah and Chen [2] gave parallel techniques for solving the minimum circle-cover problem [16]; their techniques are also useful for optimally solving the single-source shortest path problem on unweighted interval graphs. A parallel algorithm for computing a shortest path between two specified intervals on unweighted interval graphs was published recently [17]; this algorithm runs in $O(\log n)$ time using $O(n)$ EREW PRAM processors.

Our solutions to the all-pair shortest path query problem generalize the techniques in [2, 17] and make use of other parallel algorithmic techniques. It is interesting to note that our results imply that, on unweighted interval and circular-arc graphs, the all-pair shortest path query problem can be solved (both sequentially and in parallel) in the same complexity bounds as those of the optimal single-source shortest path algorithms. On arbitrary unweighted graphs, however, the best known algorithm for the all-pair shortest path query problem is, to our knowledge, no better than the all-pair shortest path algorithm [19]. Using the data structure for the all-pair shortest path query problem, we can solve the problem of computing (both sequentially and in parallel) the lengths of shortest paths between m pairs of intervals in unweighted interval and circular-arc graphs. Using the previously best known sequential algorithms to solve this problem would take $O(n \log n + \min\{mn, n^2\})$ time, but our results provide a sequential solution of $O(n \log n + m)$ time and a parallel solution of $O(\log n)$ time and $O(n + m/\log n)$ CREW PRAM processors, which is a significant improvement when m is larger than $O(\log n)$.

In the rest of this paper, we assume that the endpoints of the intervals in S are already sorted. If this is not the case of the input, then we sort the endpoints of S in $O(n \log n)$ time sequentially, and in $O(\log n)$ time using n processors in parallel by using [7]. We will only present the parallel versions of our algorithms, because the sequential bounds that we claim follow immediately from the parallel ones. Henceforth, we only show the construction of the data structure in $O(\log n)$ time using $O(n/\log n)$ processors, and the processing of the shortest path queries in the desired parallel complexity bounds.

The parallel computational model that we use is the CREW PRAM. Recall that the PRAM is a synchronous parallel computational model in which all the processors share a common memory and each processor can access any memory location in constant time. The CREW PRAM allows simultaneous accesses to the same memory location by multiple processors only if all such concurrent accesses are for reading data only. We would like to mention that except for the component for processing shortest path queries, our data structure can be built in the same complexity bounds on the EREW PRAM, hence matching the optimal bounds and model in [2, 17].

In the next section, we introduce some useful notation and observations. Section 3 presents several structures and properties needed by our solutions. Section 4 discusses our parallel algorithm on interval graphs. Section 5 extends our techniques to circular-arc graphs.

2 Preliminaries

Every interval $I_i \in S$ is defined by its left endpoint a_i and right endpoint b_i, that is, $I_i = [a_i, b_i]$, with $a_i \leq b_i$. We also denote a_i (resp., b_i) by $lp(I_i)$ (resp., $rp(I_i)$). Without loss of generality, we assume that no two distinct intervals in S have the same endpoint and that $a_i < b_i$ for every $i = 1, 2, \ldots, n$. Let L be the sorted array containing the endpoints of the intervals in S. Then $|L| = 2n$. Let $L = (p_1, p_2, \ldots, p_{2n})$, where every p_i is either a_j or b_j for some j. Note that from L, we can easily obtain the list of all the intervals in S that are sorted by the a_i's (resp., b_i's). Hence we also assume that the intervals in S have been labeled in such a way that $b_i < b_j$ if and only if $i < j$. Such a labeling is easily obtained from L in $O(\log n)$ time using $n/\log n$ processors by doing parallel prefix [14, 15].

Let s (resp., t) be the smallest left endpoint (resp., largest right endpoint) among those of S. Then $s = a_l$ for some $l \in \{1, 2, \ldots, n\}$ and $t = b_n$. Without loss of generality, we assume that $\cup_{i=1}^{n} I_i$ is equal to the interval $[s, t]$, that is, the union of the intervals in S is one connected component $[s, t]$ on the real line. This is because if it is not the case, then the length of the shortest path between any two intervals belonging to different connected components is $+\infty$, and such shortest paths can be easily processed by using some simple data structures. Note that $\cup_{i=1}^{n} I_i$ can be computed in $O(\log n)$ time and $n/\log n$ processors by using [2].

We generalize the idea of *successor function* that was used in [16, 2], by defining the successor function SUC as follows: For every point p on the real line, $SUC(p) = I_i \in S$ if and only if $b_i = \max\{b_j \mid I_j \text{ contains } p\}$. Note that for every point $q \notin [s, t]$, $SUC(q)$ is not equal to any interval in S, and we define $SUC(q) = \emptyset$ for such a point q.

We also define the successor function on the intervals of S. In particular, we let $SUC(I_i) = SUC(b_i)$ for every i. Note that, because $\cup_{i=1}^{n} I_i$ is a connected component, $SUC(I_i) = I_i$ implies that $i = n$. For example, in Figure 1, $SUC(I_1) = I_4$, $SUC(I_4) = I_9$, and $SUC(I_{10}) = I_{10}$. We also define on S the *inverse* of SUC, SUC^{-1}, by $SUC^{-1}(I_j) = \{I_i \in S \mid SUC(I_i) = I_j\}$. Obviously, we can have $|SUC^{-1}(I_j)| > 1$, and hence in general SUC^{-1} is not a function from S to S. In Figure 1, for example, $SUC^{-1}(I_{10}) = \{I_7, I_8, I_9, I_{10}\}$, $SUC^{-1}(I_5) = \{I_3\}$, and $SUC^{-1}(I_8) = SUC^{-1}(I_2) = \emptyset$. Both SUC and SUC^{-1} play important roles in our data structure for computing shortest paths.

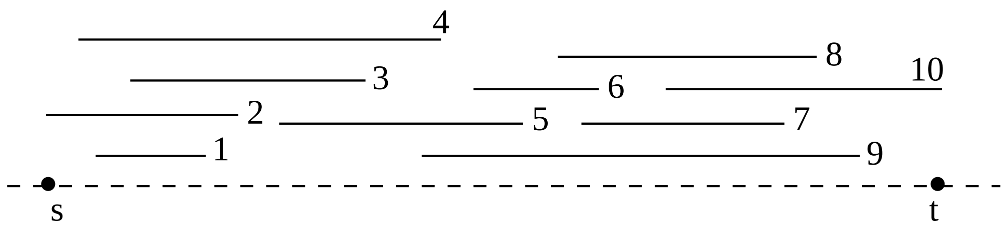

Figure 1: Illustrating the definitions of SUC and SUC^{-1}.

The SUC function has the following property:

Lemma 1 *Let p and q be two points on the real line such that $p < q$ and that $SUC(p) = SUC(q)$. Then for every point $r \in [p, q]$, $SUC(r) = SUC(p)$.*

Proof. Suppose that $SUC(p) = SUC(q) = I_i \neq SUC(r) = I_j$. If $i < j$, then it follows from $b_i < b_j$ that I_j contains q and hence $SUC(q)$ should instead be I_j, a contradiction. If $i > j$, then from the facts that $b_i > b_j$ and that I_i contains r, it follows that $SUC(r)$ should instead be I_i, again a contradiction. □

The corollaries below follow immediately from Lemma 1.

Corollary 1 *Suppose that the real line is partitioned into m contiguous regions such that two distinct points p and q are in the same region of the partition if and only if $SUC(p) = SUC(q)$. Then $3 \leq m \leq n + 2$.* □

Corollary 2 *Let I_i and I_j be such that $i < j$ and that $SUC(I_i) = SUC(I_j)$. Then for every k such that $i < k < j$, $SUC(I_k) = SUC(I_i)$.* □

Corollary 2 implies that, for every I_i, the intervals in $SUC^{-1}(I_i)$ occur in S consecutively. This property is useful in computing SUC^{-1} efficiently.

3 Properties of a Set of Intervals and Its SUC-TREE

Observe that the SUC function actually gives rise to a tree structure T (SUC-TREE) whose nodes correspond to the intervals in S. We define for every $I_i, i = 1, 2, ..., n-1$, that $SUC(I_i)$ is the parent node in T corresponding to node I_i. We will assume that the *Level* of the root of T is 0 and the level of any node is one more than that of its parent. In this section, we present certain properties of a set of intervals relevant for our technique and present a method to calculate the distance between any two intervals based on the Level numbers of the corresponding nodes in T.

We denote two overlapping intervals I_i and I_j by $I_i \sim I_j$ and nonoverlapping intervals by $I_i \not\sim I_j$. Also, $SUC^r(I_i)$ is the rth ancestor of I_i in SUC-TREE. First, we have the following property which can be easily verified.

Property 1 *If $rp(I_i) < rp(I_j)$ then $Level(I_i) \geq Level(I_j)$ and if $Level(I_i) > Level(I_j)$ then $rp(I_i) < rp(I_j)$* □

Lemma 2 *An interval I_i does not overlap with an interval I_j if $|Level(I_j) - Level(I_i)| > 1$.*

Proof: Without loss of generality, let $Level(I_j) < Level(I_i) - 1$. Let $I_k = SUC(I_i)$. Clearly, $I_k \neq I_j$. Since $Level(I_j) < Level(SUC(I_i))$, we have $rp(I_j) > rp(SUC(I_i))$ by Property 1. Now, I_i and I_j cannot overlap for otherwise I_j would become a candidate for the successor of I_i and I_k cannot be the successor of I_i. □

In the following lemmas we determine the distance relations between any two intervals depending upon the levels of their corresponding nodes in SUC-TREE.

Lemma 3 *For any two intervals I_x and I_y, if $Level(I_y) = Level(I_x)$ then*

a. If $I_x \sim I_y$ then $d(I_x, I_y) = 1$.

b. otherwise, $d(I_x, I_y) = 2$.

Proof: Part (a) of the lemma is obvious. For part (b), (i.e., when $I_x \not\sim I_y$) let us assume without loss of generality that $rp(I_y) > rp(I_x)$. Then we have to prove that there is a path $I_x \sim SUC(I_x) \sim I_y$. i.e., we have to prove that $SUC(I_x) \sim I_y$. It is clear that $rp(I_y) > rp(I_x) > lp(SUC(I_x))$. Since $Level(SUC(I_x)) > Level(I_y)$, by Property 1, we have $rp(SUC(I_x)) > rp(I_y)$. Hence we have $lp(SUC(I_x)) < rp(I_y) < rp(SUC(I_x))$ and therefore $SUC(I_x) \sim I_y$. □

Lemma 4 *For any two intervals I_x and I_y, if $Level(I_y) = Level(I_x) - 1$ then*

a. If $I_x \sim I_y$ then $d(I_x, I_y) = 1$.

b. otherwise, if $SUC(I_x) \sim I_y$ then $d(I_x, I_y) = 2$.

c. otherwise, $d(I_x, I_y) = 3$.

Proof: The proof of parts (a) and (b) are obvious. We will prove part (c). From Property 1, we have $rp(I_y) > rp(I_x)$. Also, we know that $I_y \not\sim I_x$ and $I_y \not\sim SUC(I_x)$. Since $Level(SUC(I_x)) = Level(I_y)$, $d(SUC(I_x), I_y) = 2$ by Lemma 3. Hence $d(I_x, I_y) \leq 3$. If possible, suppose that $d(I_x, I_y) = 2$. Then there exists an interval I_z such that $I_z \sim I_x$ and $I_z \sim I_y$. This would imply that $I_z \sim SUC(I_x)$. Since $I_y \not\sim SUC(I_x)$ and $rp(I_y) > rp(I_x)$, we have $lp(I_y) > rp(SUC(I_x))$. This implies that $rp(I_z) > rp(SUC(I_x))$ raising a contradiction to our assumption that $SUC(I_x)$ is the successor of I_x. Thus, $I_z \not\sim I_x$ and hence $d(I_x, I_y) = 3$. □

The following lemma is easy to prove.

Lemma 5 *Given two intervals* I_x *and* I_y *such that* $rp(I_y) > rp(I_x)$ *and* $d(I_x, I_y) \geq 2$, *then there exists a shortest path from* I_x *to* I_y *passing through interval* $SUC(I_x)$.

Lemma 6 *If* $Level(I_y) < Level(I_x) - 1$ *then* $d(I_x, I_y) = Level(I_x) - Level(I_y) - 1 + d(I_z, y)$, *where* $I_z = SUC^{Level(I_x) - Level(I_y) - 1}(I_x)$.

Proof: The proof is straightforward based on Lemmas 2 and 5. □

4 The Algorithm on Interval Graphs

This section discusses the data structure for reporting the shortest path queries on interval graphs. We first describe the data structure and its construction, and then show the process of answering the shortest path queries.

4.1 Constructing the Data Structure

The shortest path data structure consists of SUC-TREE T discussed previously preprocessed for solving shortest path queries. To preprocess T, we need to compute both *SUC* and SUC^{-1}.

The *SUC* function is computed by performing parallel prefix [14, 15] on array L, as follows:

Procedure *SUC*-Function.

(a) For every p_i in L, associate it with the value of b_j if $p_i = a_j$ for some j, and associate it with $-\infty$ if $p_i = b_j$ for some j.

(b) Perform a parallel prefix on L to compute, for every p_i, $i = 1, 2, \ldots, 2n$, the maximum value among the associated values of $p_1, p_2, \ldots, p_i$.

(c) For every p_i in L that represents b_j for some j, suppose that the value for p_i computed by Step (b) is b_k for some k (it is possible that $j = k$). Let $SUC(I_j) = I_k$.

That the above procedure correctly computes the SUC function follows immediately from the facts that (i) $SUC(I_j) = I_k$ contains b_j (and hence $a_k < b_j < b_k$ in L), and (ii) b_k is the largest right endpoint among all the intervals that contain b_j. It is easy to see that the computation of SUC takes $O(\log n)$ time using $n/\log n$ processors.

After SUC has been computed, the SUC^{-1} relation can be obtained in $O(\log n)$ time using $n/\log n$ processors, based on Corollary 2, as follows: Mark on S the indices i at which $SUC(I_i) \neq SUC(I_{i+1})$; more specifically, we compute for each index j the pair (l_j, r_j) such that $SUC^{-1}(I_j) = \{I_{l_j}, I_{l_j+1}, \ldots, I_{r_j}\}$. For example, if $SUC(I_i) = I_j \neq SUC(I_{i+1}) = I_k$, then set $r_j = i$ and $l_k = i + 1$.

Given SUC and SUC^{-1}, every child node in T knows its parent node (from SUC) and every parent node knows all its child nodes (from SUC^{-1}). Hence we can apply the Euler Tour technique [20] to T for various computations. In particular, we preprocess T so that the following type of queries on T can be quickly answered: Given a node v in T and a non-negative integer i, find the i-th node on the path in T from v to the root of T. Such queries are called *level-ancestor queries* by Berkman and Vishkin[4], who gave efficient parallel algorithms for preprocessing rooted trees so that the level-ancestor queries can be answered quickly. Let a level-ancestor query on a node v in T and an integer i be denoted by $LAQ(v, i)$. We need to be able to answer the level-ancestor queries on T efficiently, in order to handle the processor assignment problem in reporting actual shortest paths. The results of Berkman and Vishkin [4, 5] show (implicitly) that a level-ancestor query can be answered sequentially in constant time, after an $O(\log n)$ time, $n/\log n$ processor preprocessing in the CREW PRAM model. The preprocessing of tree T is done by simply using the algorithm of Berkman and Vishkin [4, 5]. From now on, we assume that such a preprocessing has been done on T. This completes the construction of the shortest path data structure.

4.2 Processing the Shortest Path Queries

The proof of the following Lemma provides both the algorithm and the complexity bounds for answering shortest path queries between any pair of intervals given the SUC-TREE preprocessed for level-ancestor queries.

Lemma 7 *Given the SUC-TREE data structure preprocessed for* level-ancestor queries*, the distance between any two intervals can be computed in $O(1)$ time.*

Proof: For intervals I_x and I_y such that $|Level(I_x) - Level(I_y)| \leq 1$ we apply Lemmas 3 and 4 to compute $d(I_x, I_y)$ and this takes $O(1)$ time. Otherwise, we first compute I_z, the $(Level(I_x) - Level(I_y) - 1)$th ancestor of node I_x in the SUC-TREE (assuming $rp(I_y) > rp(I_x)$). This can be done in constant time, since the SUC-TREE is preprocessed for level-ancestor queries. We then apply Lemmas 4 and 6 to compute $d(I_x, I_y)$.

□

We now have the main result.

Lemma 8 *A set of N intervals sorted in ascending order can be preprocessed in $O(\log N)$ time with $O(N/\log N)$ processors on the CREW-PRAM model such that the shortest distance queries can be answered in constant time using a single processor.*

Proof: The Lemma follows based on the time and processor complexity for preprocessing level-ancestor queries and Lemma 7. □

4.3 Geometric Shortest Path Queries

A geometric shortest path query is defined as follows. Given two arbitrary query points p and q on the real line, find the p-to-q shortest distance, such that the shortest path uses only intervals in S and that the union of intervals in the shortest path contains both p and q.

To solve the geometric query problems we need to find the intervals that contain points p and q. Without loss of generality assume $p < q$. In order to find these intervals we need a second data structure, the minimum interval-cover. A subset of the intervals in S whose union equals interval $[s, t]$ is called an *interval-cover* of $[s, t]$. An interval-cover of minimum size is a *minimum interval-cover* of $[s, t]$. Let $s = a_l$ for some $l \in \{1, 2, \ldots, n\}$. Then a minimum interval-cover, denoted by IC, can be obtained sequentially as follows (this algorithm is a modification of the minimum circle-cover algorithm of Lee and Lee [16]):

1. Put I_l in IC. Let $i = 1$ and $J_i = I_l$.

2. While $J_i \neq I_n$, put $J_{i+1} = SUC(J_i)$ in IC and increase i by 1.

For example, in Figure 1, $IC = \{I_2, I_4, I_9, I_{10}\}$.

Note that when both SUC and SUC^{-1} are given, such a minimum interval-cover IC can be obtained from S in $O(\log n)$ time and $n/\log n$ processors by using [2]. Let $c = |IC|$. Observe that the intervals in $IC = (J_1, J_2, \ldots, J_c)$ are listed in the sorted order according to their left (resp., right) endpoints, and that no J_i in IC is contained in another J_k in IC.

Given the sorted list of the left (resp., right) endpoints of the intervals in IC, for an arbitrary point $p \in [s, t]$, we can find, in $O(\log n)$ time and one processor, the largest (resp., smallest) left (resp., right) endpoint a_j (resp., b_j) in that list that is no bigger (resp., no smaller) than p. We call such an interval $I_j \in IC$ the *rightmost* (resp., *leftmost*) interval in IC that contains p. When IC is available, we can easily compute, for every $p_i \in L$, the rightmost and leftmost intervals in IC that contain p_i, by performing parallel prefix on L.

After having determined the intervals we can now use Lemma 8 to determine the shortest distance. The complexity of finding those intervals is $O(\log n)$ and using Lemma 8 we can report the distance in $O(1)$ time using a single processor.

After $k = |SP(p, q)|$ is known, the actual shortest path $SP(p, q)$ can be reported in $O(\log n)$ time and $\lceil k/\log n \rceil$ processors, by using the procedure below:

Procedure Path-Query.

(a) Perform, in parallel, $\lceil k/\log n \rceil - 1$ level-ancestor queries on T, by using the node $SUC(p)$ and integers $\lceil \log n \rceil$, $2\lceil \log n \rceil$, $\ldots$, $(\lceil k/\log n \rceil - 1)\lceil \log n \rceil$. (Each query is handled by one processor and in $O(1)$ time.) These queries together divide $SP(p, q)$ into $\lceil k/\log n \rceil$ subpaths of $O(\log n)$ intervals each.

(b) Report the $\lceil k/\log n \rceil$ subpaths of $SP(p, q)$ in parallel, by assigning one processor to output each subpath of $SP(p, q)$ sequentially.

5 Extension to Circular-Arc Graphs

The definition of circular-arc graphs is the same as that of interval graphs, with the exception that the set of intervals on the real line is replaced by a set of circular-arcs on a unit circle C. Circular-arc graphs arise in many applications [10]. In this section, we show how to extend our results on interval graphs to circular-arc graphs. The same notation as in the previous sections will be used, unless otherwise specified.

The input to the problem here consists of a set S of n circular-arcs on unit circle C. $S = \{I_1, I_2, \ldots, I_n\}$ is called a circular-arc model of a circular-arc graph (see Figure 2). The

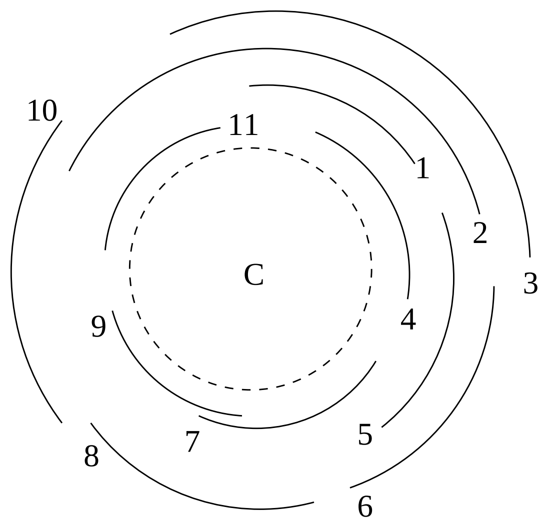

Figure 2: A circular-arc model on circle C.

left (resp., right) endpoint $lp(I_i) = a_i$ (resp., $rp(I_i) = b_i$) of arc I_i is the last point of I_i that we encounter when walking along I_i counterclockwise (resp., clockwise) until a point not on I_i is met. That is, $I_i = [a_i, b_i]$ is the arc on C from a_i clockwise to b_i.

Without loss of generality, we assume that no arc in S is of length 2π (otherwise, the shortest path problem becomes trivial). Also, we assume that $\cup_{i=1}^{n} I_i = C$ (otherwise, the problem becomes one on interval graphs). Furthermore, we assume that the endpoints of the arcs in S are already given sorted, according to the order in which they are visited during the clockwise traversal along C by starting at a_1. Let L be the array containing the $2n$ endpoints $p_1, p_2, \ldots, p_{2n}$ of the arcs in S, in such a sorted order.

5.1 The Data Structure

In the circular-arc graph case, the data structure consists of three parts: (1) a forest F based on the *SUC* function, (2) a minimum circle-cover CC of circle C, and (3) a partition P of C. A subset of the arcs in S whose union covers C is called a *circle-cover*, and a circle-cover of minimum size is a *minimum circle-cover*. The minimum circle-cover CC can be computed in $O(\log n)$ time and $n/\log n$ processors by using the algorithm of Atallah and

Chen[2].

The third part of the data structure consists of a partition P of C. First, we will show how to construct P for a set of intervals on a real-line using functions SUC and SUC^{-1}. This technique can be then easily extended to the case of set of arcs in C using the SUC function defined on C. By Corollary 1, the real line can be partitioned into at most $n+2$ contiguous regions such that the points in the same region all have the same SUC value. We denote such a partition by P. Note that the boundary points of all these regions belong to L. Hence the description of P can be obtained by comparing the SUC values of consecutive points in L, as in the computation for SUC^{-1}. We represent P by the list $(q_1, q_2, \ldots, q_d)$, where every q_i is a boundary point separating two consecutive regions of P and d is the number of such boundary points of P. Given P, we need to compute, for every $p_i \in L$, the region in P that contains p_i. Let such a region for p_i be denoted by $P(p_i)$. $P(p_i)$ can be computed, for every $i = 1, 2, \ldots, 2n$, in $O(\log n)$ time and $n/\log n$ processors, by doing parallel prefix on L.

The other things that we need to discuss are methods to determine SUC and SUC^{-1}, and the forest F that we obtain from them. For a point p on C, $SUC(p) = I_i \in S$ if and only if two conditions hold for I_i: (i) I_i contains p, and (ii) the arc on C from p clockwise to b_i (i.e., the arc $[p, b_i]$) has the longest arc length among all such arcs formed by p and the right endpoints of the arcs in S that contain p. For every $I_j \in S$, $SUC(I_j)$ and $SUC^{-1}(I_j)$ are both defined in the same way as in the interval graph case. In Figure 2, $SUC(I_1) = I_4$, $SUC(I_7) = I_9$, and $SUC(I_{11}) = I_3$; $SUC^{-1}(I_7) = \{I_5, I_6\}$. Note that in this situation, there is no $I_i \in S$ such that $SUC(I_i) = I_i$ (otherwise, the union of the arcs in S would have not covered C).

In order to compute the SUC function efficiently, we would like to obtain, from the arcs in S, a set S' of intervals on the real line. This is done as follows.

1. Map the elements of L onto the real line: This is done as if circle C is "cut" open at a_1 and then the arc of length 2π so resulted is "unrolled" onto the real line. Let the interval $[0, 2\pi]$ on the real line correspond to C, with a_1 being at 0. Every element p_i in L is mapped to a point (also denoted by p_i) on interval $[0, 2\pi]$, such that the length of the interval $[0, p_i]$ on the line is equal to the length of the arc $[a_1, p_i]$ on C. Let L_1 be the array containing the points on the line corresponding to the endpoints in L. Let $s = 0$ and $t = 2\pi$. Make t the last element of L_1.

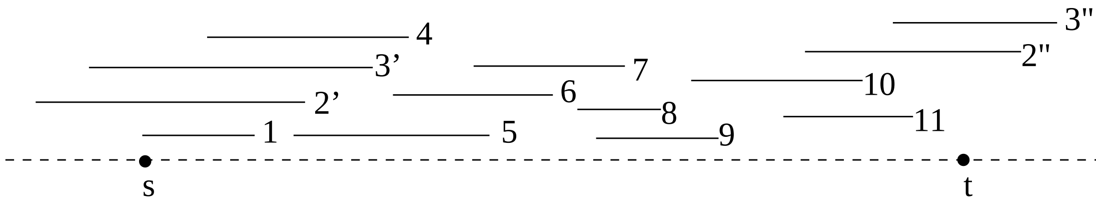

Figure 3: The resulting intervals reduced from circular-arcs.

2. Reduce the arcs in S to intervals on the line: For every arc $I_i \in S$, if the interior of I_i does not contain a_1, then arc I_i is mapped to the interval $[a_i, b_i] \subset [s, t]$, which is also denoted by I_i. If the interior of arc I_i contains a_1, then create two intervals on the line corresponding to I_i: $I'_i = [-a'_i, b_i]$ and $I''_i = [a_i, b'_i]$, where a'_i is equal to the length of the arc $[a_i, a_1]$ on C and b'_i is equal to 2π plus the length of the arc $[a_1, b_i]$ on C. For example, the set of arcs in Figure 2 is reduced to the set of intervals in Figure 3. The set of intervals so resulted is denoted by S'. Obviously, $|S'| = O(n)$. Let L_2 be the sorted array containing the $-a'_i$'s and L_3 be the sorted array containing the b'_i's. Note that L_2 and L_3 can each be easily obtained from L in $O(\log n)$ time using $n/\log n$ processors, by checking every $p_i \in L$ to see whether it is an endpoint of an arc that properly contains a_1. Let L' be the concatenation of L_2, L_1, and L_3, i.e.,

$$L' = L_2 L_1 L_3.$$

Relabel the elements in L' and denote them by p'_i.

Once L' is available, the *SUC* function for the points on C can be obtained by computing the *SUC* function for the points on interval $[s, t]$, as follows:

(a) Associate the elements in L' with appropriate values: If an arc $I_j \in S$ does not properly contain a_1, then associate element $p'_i = a_j$ with the value of b_j on the line; otherwise, associate element $p'_i = -a'_j$ with the value of b_j and associate element $p'_h = a_j$ with the value of b'_j on the line. Associate t and all the b_j's and b'_j's with $-\infty$.

(b) Perform a parallel prefix on L' to compute, for every p'_i, the maximum value among all the associated values of p'_1, p'_2, $\ldots$, p'_i.

(c) For every p'_i in L' that represents b_j for some j, suppose that the value for p'_i computed by Step (b) is either b_k or b'_k for some k. Let $SUC(I_j) = I_k$ for arc $I_j \in S$.

The above procedure obviously computes the SUC function for the arcs in S (and hence for the points on C), in $O(\log n)$ time using $n/\log n$ processors. Given the SUC function, the SUC^{-1} relation is obtained in the similar way as in the interval graph case.

Unlike the interval graph case, the SUC function on the arcs of S does not give rise to a tree or a forest. To be able to use the level-ancestor queries in reporting shortest paths, we need to construct a forest F based on the SUC function. This is done by duplicating certain arcs in S and slightly modifying the SUC function, as follows.

(a) Let W be the set of the arcs in S that properly contain a_1.

(b) For every arc $I_i \in W$, create a copy $New(I_i)$ of I_i.

(c) For every arc $I_i \in W$, if $SUC(I_i) = I_j \in W$ and $b_i < b_j$ in L, then let $SUC(New(I_i)) = New(I_j)$; otherwise, let $SUC(New(I_i)) = I_j$. For every $I_i \in S - W$, if $SUC(I_i) = I_j \in W$ and $b_i < b_j$ in L, then let $SUC(I_i) = New(I_j)$.

(d) For every arc $I_i \in W$, let $SUC(I_i) = \emptyset$.

(e) The values of the SUC function on other arcs in S remain unchanged.

The above procedure can be easily implemented in $O(\log n)$ time and $n/\log n$ processors (by using the SUC and SUC^{-1} information). In this way, we create a forest F whose nodes are the arcs in S and the copies $New(I_i)$ for the arcs $I_i \in W$. The roots of the trees in F are these arcs in W. Note that a similar method for obtaining a forest from the SUC function has been used in [2].

For every node I of F, let $Root(I)$ be the element of W that is at the root of the tree in F containing I; also, let $Depth(I)$ be the depth of I in its tree (i.e., the number of nodes on the I-to-$Root(I)$ path in F). Observe that the arrays $Root$ and $Depth$ can both be computed in $O(\log n)$ time and $n/\log n$ processors by using the Euler Tour technique [20] in conjunction with optimal parallel list-ranking [1, 8] (we can use the Euler Tour technique because we have the SUC^{-1} relation available, which provides the list of children for every node in F). This finishes the construction of the shortest path data structure for circular-arc graphs.

5.2 The Shortest Path Queries

Given two arbitrary query points p and q on C, we would like to report the length and the actual path of the p-to-q shortest path $SP(p,q)$ whose arcs are all in S and the union of whose arcs contains both p and q. Observe that $[p,q] \cup [q,p] = C$ and that the union of the arcs on $SP(p,q)$ must contain either arc $[p,q]$ or arc $[q,p]$. Hence $SP(p,q)$ is the shorter of two shortest paths: (i) the shortest path that uses only the arcs in S and goes from p clockwise to q along C, and (ii) the shortest path that uses only the arcs in S and goes from q clockwise to p along C. Note that we don't need to compute the actual paths of both (i) and (ii). As long as we know the lengths of these two shortest paths, we are able to report $SP(p,q)$ correctly. Therefore, we only show how to compute these two shortest paths efficiently.

Observe that a_1 can fall into the interior of either $[p,q]$ or $[p,q]$, but not both. Without loss of generality, assume $a_1 \in [p,q]$. Then the shortest path from q clockwise to p along C can be computed by simply using the algorithm for answering shortest path queries in the interval graph case (with CC playing the role of IC). Therefore, we only need to show how to compute the shortest path from p clockwise to q along C. Such a shortest path crosses a_1 and hence passes the root of a tree in F. We use the following procedure.

(a) Compute $SUC(p)$ and let it be I_i. (For an arbitrary point p, this takes $O(\log n)$ time using one processor.)

(b) If $Root(I_i)$ contains q, then the length of the shortest path we seek is $Depth(I_i)$; otherwise, the length of this shortest path is equal to $Depth(I_i)$ plus the length of the shortest path from $rp(New(Root(I_i)))$ clockwise to q along C. The length of the shortest path from $rp(New(Root(I_i)))$ clockwise to q along C can be computed by using the algorithm for the interval graph case (with CC playing the role of IC). (This takes $O(1)$ time and one processor.)

(c) If the actual path is desired, then use node I_i in F and the value $Depth(I_i)$ to perform level-ancestor queries in parallel in F, in order to report the subpath of the shortest path that consists of the arcs from I_i to $Root(I_i)$. Reporting the arcs on the subpath of the shortest path from $rp(New(Root(I_i)))$ clockwise to q along C is done in the same way as in the interval graph case.

Given two arcs I_i and I_j in S, if $I_i \cap I_j \neq \emptyset$, then $SP(I_i, I_j)$ is trivially available.

Otherwise, $SP(I_i, I_j)$ is obtained from one of the following two shortest paths: (i) the shortest path from $rp(I_i)$ clockwise to $lp(I_j)$ along C, and (ii) the shortest path from $rp(I_j)$ clockwise to $lp(I_i)$ along C. At this point, it should be easy to see that $|SP(I_i, I_j)|$ can be found in $O(1)$ time using one processor, and $SP(I_i, I_j)$ can be reported in $O(1)$ time using k processors, where $k = |SP(I_i, I_j)|$.

References

[1] R.J. Anderson and G.L. Miller. "Deterministic parallel list ranking," *Algorithmica*, 6 (1991), pp. 859–868.

[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] M.J. Atallah, D.Z. Chen, and D.T. Lee. "An optimal algorithm for shortest paths on weighted interval and circular-arc graphs, with applications," *Algorithmica*, 14,5, Nov. 1995, 429-441.

[4] O. Berkman and U. Vishkin. "Finding level-ancestors in trees," *Journal of Computer and System Sciences*, Vol. 48 (1994), pp. 214–230.

[5] O. Berkman and U. Vishkin. Personal communication.

[6] K.S. Booth and G.S. Lueker. "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.

[7] R. Cole. "Parallel merge sort," *SIAM J. Comput.*, 17 (4) (1988), pp. 770–785.

[8] R. Cole and U. Vishkin. "Approximate parallel scheduling, part I: The basic technique with applications to optimal parallel list ranking in logarithmic time," *SIAM J. Comput.*, 17 (1) (1988), pp. 128–142.

[9] D. Coppersmith and S. Winograd. "Matrix multiplication via arithmetic progression," *Proc. 19th ACM Symposium on Theory of Computing*, 1987, pp. 1–6.

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

[11] 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.

[12] 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.

[13] N. Korte and R.H. Möhring. "A simple linear-time algorithm to recognize interval graphs," *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science*, Ed. by G. Tinhofer and G. Schmidt, 1986, Germany, pp. 1–16.

[14] C. P. Kruskal, L. Rudolph, and M. Snir. "The power of parallel prefix," *IEEE Trans. on Computers*, C-34 (10) (1985), pp. 965-968.

[15] R. E. Ladner and M. J. Fischer. "Parallel prefix computation," *J. of the ACM*, 27 (4) (1980), pp. 831–838.

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

[17] S. Olariu, J.L. Schwing, and J. Zhang. "Optimal parallel algorithms for problems modeled by a family of intervals," *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, No. 3 (1992), pp. 364–374.

[18] R. Ravi, M.V. Marathe, and C.P. Rangan. "An optimal algorithm to solve the all-pair shortest path problem on interval graphs," *Networks*, Vol. 22 (1992), pp. 21–35.

[19] R. Seidel. "On the all-pair-shortest-path problem," *Proc. 24th ACM Symposium on Theory of Computing*, 1992, pp. 745-749.

[20] R. E. Tarjan and U. Vishkin. "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, 14 (4) (1985), pp. 862–874.