Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

Optimal parallel algorithm for shortest-paths problem on interval graphs

MISHRA P.K.

(*Department of Applied Mathematics, Birla Institute of Technology, Mesra-Ranchi, 835215, India*)

E-mail: pkmishra@ieee.org; pkmishra@bitmesra.ac.in

Received Nov. 13, 2003; revision accepted Jan. 8, 2004

Abstract: This paper presents an efficient parallel algorithm for the shortest-path problem in interval graph for computing shortest-paths in a weighted interval graph that runs in $O(n)$ time with n intervals in a graph. A linear processor CRCW algorithm for determining the shortest-paths in an interval graphs is given.

Key words: Parallel algorithms, Shortest-paths problem, Interval graphs
doi:10.1631/jzus.2004.1135 **Document code:** A **CLC number:** O29

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 and J_i and J_{i+1} overlap for every $I \in \{1, 2, \ldots, k-1\}$. The length of σ is the sum of the weights of its interval s 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.

This algorithm solves this shortest-paths problem on interval graphs optimally in $O(n)$ time, when given the model of such a graph, i.e. the actual weighted intervals or circular-arcs and the sorted list of the interval end-points. A node of an interval graph corresponds to an interval and an edge is between two nodes in the graph iff the two intervals corresponding to these nodes intersect each other. Note that an interval or circular-arc graph with n-nodes can have $O(n^2)$ edges. This 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. One of the minimum-weight circle-cover problems, whose definition we briefly review is:

Given a set of weighted circular-arcs on a circle, choose a minimum-weight-subset of the circular-arcs whose union covers the circles.

It is known that the minimum-weight circle-cover problem can be solved by solving q instances of the previously mentioned single source shortest-path problem, where q is the minimum number of arcs crossing any point on the circle.

It is the circle-cover problem that has the main practical applications (Aho *et al*., 1974) and the study of this shortest-paths problem mainly aimed at 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 (Mishra and Sharma, 2002), the shortest-paths result may be useful in other optimization problems. More importantly, this approach holds the promise of shaving a log$n$

factor from the time complexity of other problems on such graphs.

Note that, by using the single-source shortest-paths algorithm, the all pair shortest-paths problem on weighted-interval and circular-arc graphs can be solved in $O(n^2)$ time, which is optimal. The previously best time bound for the all-pair shortest-paths problem (Lee and Lee, 1984) on weighted interval graphs was $O(n^2\log n)$.

An $O(n^2)$ time and space algorithm for the un-weighted case of the all pair shortest-path problem was given, and these bounds were improved by Chen and Lee (1994). We henceforth assume that the intervals are given sorted by their left end-points and also sorted by their right end-points. 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 with the previously best bound for solving that problem which was $O(qn\log n)$ [by using q times the subroutine for solving the $O(n\log n)$ time each]. Use of the shortest-paths algorithm solves the minimum-weight circle-cover problem in $O(qn+n\log n)$ time where the $(n\log n)$ term is from the preprocessing sorting step when the sorted list of end-points 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 is 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 that we are computing the shortest-paths from the source interval to only those intervals whose right end-points are to the right of the right end-points 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 end-points are to the left of the left end-point of the source. Clearly we need not worry about paths to intervals whose right end-points are covered by the source since the problem is trivial as in these intervals–the length of the shortest-path is simply the sum of the weight of the source plus the weight of the destination, provided the weights are all non-negative.

We consider the shortest-paths problem on interval graphs in which the weights of the intervals are non-negative. The minimum-weight circle-cover problem, however, does allow circular-arcs to have negative-weights. Bertossi (1988) reduced any minimum-weight circle-cover problem with both negative- and non-negative-weights to only one non-negative-weights problem (to which the algorithm for computing shortest-paths in interval graphs with non-negative-weights is applicable). Therefore it suffices to solve the shortest-paths problem on interval graphs for the case of non-negative-weights. Bertossi (1988) 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 in $O(n)$ time and $O(n)$ space, the computation for actual shortest-paths and a shortest-path tree, i.e. a tree rooted at the source node such that the path in the tree from the root to each node of the tree is the shortest-path in the graph between them. In the next section we introduce terminology needed in the rest of the 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 sub optimal algorithm which illustrates our main idea and observation, 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 non-negative-weight case to the positive-weight case, and shows how to use the solution of the shortest-paths problem on interval graphs to obtain the solution to that on circular-arc graphs.

TERMINOLOGY

In this section we introduce some additional

terminology.

We say that an interval I contains another interval J, iff $I \cap J = J$. (1)

We say that I overlaps with J iff the intersection is non-empty, and that I properly overlaps with J iff they overlap but neither one contains the other.

An interval I is typically defined (Mishra and Sharma, 1997) by its two end-points, i.e. $I=[a,b]$ where $a \le b$ and a (resp. b) is called the left (resp. right) end-point of I. A point x is to the left (resp. right) of interval $I=[a,b]$ iff $x<a$ (resp. $b<x$). (2)

We assume that the input set S consists of intervals $I_1, \ldots, I_n$ where $I_i=[a_i,b_i]$, $b_1 \le b_2 \le \ldots \le b_n$, (3) and that the weight of each interval I_i is $\omega_i \ge 0$. To avoid unnecessarily cluttering the exposition, we assume that the interval has distinct end-points, that is $i \ne j$ implies $a_i \ne a_j$, $b_i \ne b_j$, $a_i \ne b_j$ and $b_i \ne a_j$ (4) (the algorithm for non-distinct end-points 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^*.

PRELIMINARY ALGORITHM

This section gives a preliminary $O(n\log\log n)$ time (hence sub optimal) algorithm for the special case of the shortest-paths problem on intervals with positive-weights (Booth and Luekher, 1976). This should be viewed as a "warm-up" for the next section, which gives 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 non-negative-weight intervals.

We begin by introducing definitions that lead to the concept of inactive intervals (Gupta *et al*., 1982) 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_i', that consists of S_i, and one are more intervals (not necessarily in S) whose right end-points are larger than b_i (There are of course, infinitely many choices for such an S_i').

Definition 3 An interval I_k in S_i ($k \le i$) is inactive in S_i iff for every extension S_i' of S_i, the following holds:

Every $J \in S_i' - S_i$ for which there is an I_1-to-J path in S_i' has no shortest I_1-to-J path in S_i' 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 the right end-point 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 $J \in \{I_{i+1}, \ldots, I_n\}$).

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 end-point of any active intervals in S_i.

Proof If I_k, $k \le i$, is active in S_i, then by definition 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. It thus follows that every point on the contiguous portion of the line from a_1-to-b_j where b_j is the rightmost end-point of any active interval in S_i is contained in the union of all the active intervals in S_i.

The following corollary follows from Lemma 1.

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

Definition 4 Let $label_j(i)$, $j \le i$, denote the length of a shortest I_1-to-I_i path in S that does not use any I_k for which $k \ge j$. By convention,

$$\text{If } j<i, \text{ then } label_j(i)=+\infty. \quad (5)$$

Observe that, for all i,

$$label_1(i) \geq label_2(i) \geq \ldots \geq label_n(i) \quad (6)$$

For an $I_k \in S_i$ if there is no I_1-to-I_k path in S_i, then obviously

$$label_i(j) = +\infty \quad (7)$$

for every $j=k, k+1, \ldots, i$.

Lemma 2 If $i>k$ and $label_i(i)<label_i(k)$, then I_k is inactive in S_i. (8)

Proof Since $label_i(i)<label_i(k)$, $label_i(i)$ is not $+\infty$. Hence there is a shortest I_1-to-I_i 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 $j \in S_i' - S_i$ where S_i' is an extension of S_i.

The following are immediate consequences of Lemma 2.

Corollary 2 Let $I_{j1}, I_{j2}, \ldots, I_{jk}$ be the active intervals in S_j,

$$j_1 < j_2 < \ldots < j_k \leq i. \quad (9)$$

Then

$$label_i(j_1) \leq label_i(j_2) \leq \ldots \leq label_i(j_k). \quad (10)$$

Note that the right end-points of the active intervals $I_{j1}, I_{j2}, \ldots, I_{jk}$ in S_i are in the same sorted order as that of their labels $label_i(j_1)$, $label_i(j_2)$, …, $label_i(j_k)$. Their left end-points however, are not necessarily in such a sorted order.

Corollary 3 If I_i contains I_k (hence $i>k$) and $label_i(k)<label_i(i)$, then I_k is inactive in S_i. (11)

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.

(1) The shortest I_1-to-I_k path in S_i does not need to use I_i. Then

$$label_{i-1}(k) = label_i(k), \quad (12)$$

and hence

$$label_i(i) < label_i(k). \quad (13)$$

By Lemma 2, I_k is inactive in S_i.

(2) The shortest I_1-to-I_k path in S_i does not use I_i. Then

$$label_i(k) \geq label_i(i) + \omega_k > label_i(i) \text{ (Since } \omega_k > 0) \quad (14)$$

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 end-point of any interval in S_k).

This implies that I_k contains the right end-point of at least one active interval in S_k other than I_k. However, 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 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, …, I_n (i.e. the scan is based on the increasing order of the sorted right end-points 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 binary search 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 post processing step will then compute, in linear time by CRCW parallel computational (Mishra, 2004) model (Concurrent Read Concurrent Write). The correct $label_n(i)$'s of the

inactive I_i's is S_n.

The details of the preliminary algorithm follow next. In this algorithm the right end-points of the active intervals are maintained in the leaves of the tree structure T, one end-point per leaf, in sorted order.

(1) Initialize T to contain I_1.

(2) For i=2, 3, …, 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 iteration i+1. So suppose such a, b_j exists.

Set $label_i(i)=label_{i-1}(j)+\omega_i$, 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 of I_k which is immediately to the left of the leaf for I_i in T, to see whether I_k is inactive S_i (by Lemma 3, i.e. check whether $label_{i-1}(k)<label_i$, 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

$$\text{i.e. } label_i(i)=label_{i-1}(j)+\omega_i \geq label_{i-1}(j). \quad (15)$$

Of course any I_l whose leaf T is not deleted is in fact active in S_i and already has the correct value of $label_i(l)$: it is simply the same as $label_{i-1}(l)$ 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 $label_n(l)$'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 end-points $b_j>a_i$ such that I_j is active in S_n, and set $label_n(i)=label_n(j)+\omega_i$. 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 end-points.

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 $O(n\cdot \log n)$ if we implement T as 2-3 tree (Atallah and Chen, 1989), but $O(n\log\log n)$ if we use the data structure of (Gabow and Tarjan, 1985) (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.

A LINEAR TIME IMPLEMENTATION

As observed earlier, the main bottleneck is Step 2 of the preliminary algorithm given in the previous section. We implement essentially the same algorithm but without using the tree. Instead, we use a UNION-FIND structure (Gabow and Tarjan, 1985) where the elements of the sets are integers in {1, …, n}, with integer i corresponding to interval I_i. 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 elements other than i). During the execution of Step 2 we maintain the following data structures (Mishra and Sharma, 2002) and associated 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_1, I_2, …, I_k, are the active intervals in S_i, $i_1<i_2<\ldots<i_k$, then for every $i_j\in\{i_1, i_2, \ldots, i_{k-1}\}$, the indices of the inactive intervals $\{I_l \mid i_j< i_{j+1}\}$ are all in the set whose name is i_{j+1}. Set i_{j+1} by definition consists of the indices of the above mentioned inactive intervals, and also of the index i_{j+1} of the active interval I_{ij+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 end-points are contained in interval I_1). The next invariant is about intervals that are inactive in

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. Observe that based on Lemma 1, every interval in $Loose(S_i)$ is not empty, then let $CC_1, CC_2, \ldots, CC_i$, be the connected components of $Loose(S_i)$: There is a set named j_1 for every such CC_1, where I_{jl} is the rightmost interval in CC_1 (I_{jl} is interval in CC_1 having the largest right end-point); we say that such an inactive I_{jl} is special inactive. The μ (say) elements in set j_l correspond to the μ intervals in CC_1, 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 $1<l<t$ and that $J_t=i$.

(3) An auxiliary stack contains the active intervals $I_{j1}, I_{j2}, \ldots, I_{jl}$ mentioned in item (1) above, with I_{jk} at the top of the stack. We call it the active stack.

(4) Another auxiliary stack contains the special inactive intervals $I_{j1}, I_{j2}, \ldots, I_{jl}$ at the top of the stack. We call it the special inactive 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) mentioned. 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 $a_i<b_l$, i.e.

$$b_l=\min\{b_r \mid I_r\in S, a_i<b_r\} \tag{16}$$

Note that $l\le i$ and that $l=i$ 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 pre-computed in linear time by scanning right-to-left the sorted list of all the $2n$ interval end-points.

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 three cases.

(1) $j=i$, 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 $i+1$.

(2) If $j<i$ and I_j is active in S_{i-1}, we set

$$label_i(l)=label_{i-1}(j)+\omega_i. \tag{17}$$

Then do the following updates on the two stacks:

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

(b) We repeatedly check whether the top of the active stack, I_{ik}, 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_{ik} becomes inactive, then we do $UNION(i_k, i)$, pop I_{ik} from the active stack, and continue with I_{ik-1}, etc. If the outcome of the test is that I_{ik} 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 $i+1$.

If I_i is a active in S_i, $j=j_1$, and

$$label_i(l)<label_{i-1}(j_2) \tag{18}$$

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). However, I_i does overlap with one or more 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 which I_i overlaps be called, in left to right order $C_1, C_2, \ldots, C_h$. The rightmost intervals of C_1 is I_j. Let $I_{r2}, I_{r3}, \ldots, I_{rk}$ be the rightmost interval of (respectively) $C_2, C_3, \ldots, C_h$ (of course $I_{rh}=I_{i-1}$). Observe that the top h intervals in the special inactive stack are $I_j, I_{r2}, I_{r3}, \ldots, I_{rh}$, with I_{rh} ($=I_{i-1}$) on top. Because of I_i, all of these h intervals will become inactive in S_i (whereas they were special

inactive in I_{i-1}). Their h sets (corresponding to C_1, C_2, …, C_k) 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_{rh}, …, I_{r2}, I_j from the special inactive stack.

(b) Performing *UNION*(r_h, i), *UNION*(r_{h-1}, i), …, *UNION*(r_2, i), *UNION*(j, i).

(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 (Booth and Luekher, 1976) that a sequence of m *UNION* and *FIND* operations on n elements can be performed in $O[m\alpha(m+n, n)+n]$ time (Chen and Lee, 1994), where $O(m+n, n)$ 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 an $O(n)$ 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 (Mishra, 2004). 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 in (Ibarrra *et al*., 1992) in order to result in linear-time performance: It is the linked list LL=(l, 2, …, n) where the element in LL that follows element is $next(l)=l+1$, for every l=1, 2, …, $n-1$ (the requirement in Gupta *et al*. (1982) was 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 *union*(*next*(l), l), as defined (Golumbic, 1980), that is, it concatenates two disjoint but consecutive sub-lists of LL into one contiguous sub-list of LL. On this kind of structure a sequence of m *UNION* and *FIND* operations on n elements can be performed in $O(m+n)$ time (Golumbic, 1980). Therefore, the time complexity of our algorithm is $O(n)$.

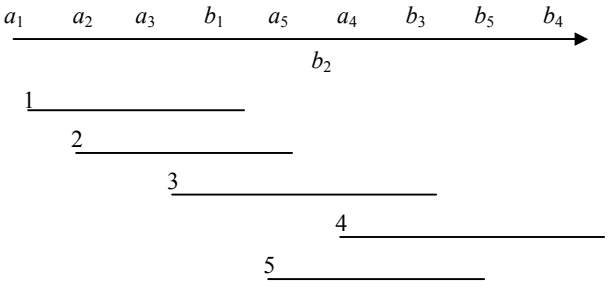
EXAMPLE: INTERVAL OPERATIONS

We consider a set of n intervals

$$I=\{I_1, I_2, \ldots, I_n\} \text{ of a line.} \tag{19}$$

Given an interval I, *maxright*(I) [*minright*(I)] denotes among all intervals that intersect the right end-point of I, the one whose right end-point is the farthest right (left) (Fig.1). The normal definition is as follows:

$$maxright(I_i)=\begin{cases} I_j & \text{if } b_j=\max\{b_k \mid a_k \le b_i \le b_k\} \\ nil & \text{otherwise} \end{cases} \tag{20}$$



maxright(1)=3, *maxright*(2)=5, *maxright*(3)=*maxright*(5)=4, *maxright*(4)=*nil*;
minright(1)=2, *minright*(2)=3, *minright*(3)=5, *minright*(4)=*nil*, *minright*(5)=4;
first({1, 2, 3, 4, 5})=1;
next(1)=5, *next*(2)=4, *next*(3)=*next*(4)=*next*(5)=*nil*

Fig.1 An example of the *maxright*, *minright*, *first* and *next* functions

One way to compute the function *maxright* (and *minright* with the appropriate variations) is given in Algorithm 1.

After Step 1 of Algorithm 1, we know that all the left end-points of the intervals intersecting I_i ($1\le i\le n$) are on the left of its right end-point b_i. Due to the definition of d_i ($1\le i\le n$) and of the prefix maximum on d_i at Step 4, we are sure that for all the right end-points b_i ($1\le i\le n$), e_i gives the right end-point the furthest right of the intervals which intersect I_i and that num_i gives the number of the associated interval, that is to say $maxright(I_i)$. We keep negative values for num_i ($1\le i\le 2n$) for left end-points in order to be able at Step 2 to distin-

guish the left end-points from the right end-points.

Step 1 requires $O[T_s, (n, p)]$ time, whereas all the other steps require $O(n/p)$ local computations.

Step 1 and Step 4 use a constant number of communications rounds. Then, *maxright* (and *minright* with the appropriate modifications) can be computed with time complexity $O[T_s, (n, p)]$ and a constant number of communications rounds.

ALGORITHM 1: *MAXRIGHT*

Input: n intervals I_i $(1\le i\le n)$ (n/p intervals on each processor)

Output: $maxright(I_i)$ $(1\le i\le n)$

Step 1: Global sort of the end-points of the intervals in ascending order.

Step 2: For each $i\in[1, 2n]$ do

assign to end-point c_i the value d_i, defined by

$$d_i=\begin{cases} b_j & \text{if } c_i=a_j, \text{ for some } 1\le j\le n \\ 0 & \text{if } c_j=b_j, \text{ for some } 1\le j\le n \end{cases}$$

Step 3: For each $i\in[1, 2n]$ do

assign to end-point c_i the value num_i, defined by

$$num_i=\begin{cases} -j & \text{if } c_i=a_j, \text{ for some } 1\le j\le n \\ j & \text{if } c_j=b_j, \text{ for some } 1\le j\le n \end{cases}$$

Step 4: Compute prefix maximum on the d_i, and let the result in $e_1, e_2, e_3, \ldots, e_{2n}$ and at the same time update the value num_i according to the following rule:

If $e_i\ne d_i$ and $i>1$ set

$$\begin{cases} -\left|num_{(i-1)}\right|, & \text{if } c_i=a_j, \text{ for some } 1\le j\le n \\ \left|num_{(i-1)}\right|, & \text{if } c_i=b_j, \text{ for some } 1\le j\le n \end{cases}$$

Step 5: For each $i\in[1, 2n]$ do

$$\begin{cases} \text{Set } maxright(I_k)=I_j, \text{ if } c_i=b_k \text{ and } num_i=j \\ \quad \text{and } k\ne j; \\ \text{Set } maxright(I_k)=nil, \text{ if } c_i=b_k \text{ and } num_i=j \\ \quad \text{and } k=j. \end{cases}$$

ALGORITHM 2: *NEXT*

Input: n intervals I_i $(1\le i\le n)$ (n/p intervals on each processor)

Output: $maxright(I_i)$ $(1\le i\le n)$

Step 1: Global sort of the end-points of the intervals in ascending order.

Step 2: For each $i\in[1, 2n]$ do

assign to end-point c_i the value d_i, defined by

$$d_i=\begin{cases} b_j & \text{if } c_i=a_j, \text{ for some } 1\le j\le n \\ 0 & \text{if } c_j=b_j, \text{ for some } 1\le j\le n \end{cases}$$

Step 3: For each $i\in[1, 2n]$ do

assign to endpoint c_i the value num_i, defined by

$$num_i=\begin{cases} -j & \text{if } c_i=a_j, \text{ for some } 1\le j\le n \\ j & \text{if } c_j=b_j, \text{ for some } 1\le j\le n \end{cases}$$

Step 4: Compute suffix maximum on the d_i, and let the result in $e_1, e_2, e_3, \ldots, e_{2n}$ and at the same time update the value num_i according to the following rule:

If $e_i\ne d_i$ and $i>1$ set

$$\begin{cases} -\left|num_{(i-1)}\right|, & \text{if } c_i=a_j, \text{ for some } 1\le j\le n \\ \left|num_{(i-1)}\right|, & \text{if } c_i=b_j, \text{ for some } 1\le j\le n \end{cases}$$

Step 5: For each $i\in[1, 2n]$ do

$$\begin{cases} \text{Set } maxright(I_k)=I_j, \text{ if } c_i=b_k \text{ and } num_i=j \\ \quad \text{and } k\ne j; \\ \text{Set } maxright(I_k)=nil, \text{ if } c_i=b_k \text{ and } num_i=j \\ \quad \text{and } k=j. \end{cases}$$

We define the parameter $first(\mathcal{I})$ as the segment I which "end first", that is, whose right end-point is the furthest left (Fig.1):

$$first(\mathcal{I})=I_j, \text{ with } b_j=\min\{b_i \mid 1\le i\le n\}$$

To compute it, we need only to compute the minimum of the sequence of right end-points of intervals in the family $\mathcal{I}$.

We will also use the function next(I): $\mathcal{I}\to\mathcal{I}$ defined as

$$next(I_i)=\begin{cases} I_j, & \text{if } b_j=\min\{b_k \mid b_i<a_k\} \\ nil, & \text{otherwise.} \end{cases}$$

That is $next(I_i)$ is the interval that ends farthest to

the left among all the intervals beginning after the end of I_i (Fig.1).

To compute $next(I_i)$ ($1 \le i \le n$), we use the same algorithm used for $maxright(I_i)$ (Algorithm 1) with a new Step 4. The algorithm compute the function *next*.

It is easy to see that the given procedure implements the definition of $next(I_i)$ with the same complexity as for computing $maxright(I_i)$ which is $O(n)$.

CONCLUSION

We have given a linear processor CRCW algorithm for determining the shortest-paths in an interval graph which runs in $O(n)$. Our motivation for considering graph was to see if they could be used to solve the shortest-path problem for interval graphs. Our algorithm solves this problem optimally in $O(n)$ time, where n is the number of intervals in a graph. The $n\log n$ term time complexity is obtained from a preprocessing sorting step when the sorted list of end-points is not given as a part of the input.

ACKNOWLEDGEMENT

The author is very grateful to Prof. H.C. Pande, Vice-Chancellor Emeritus, Prof. S.K. Mukherjee, Vice-Chancellor, Prof. N.C. Mahanti, Prof. & Head Dept. of Applied Mathematics, Birla Institute of Technology, Mesra (India) for his support and encouragement.

References

Aho, A.V., Hopcroft, J.E., Ullman, J.D., 1974. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA.

Atallah, M.J., Chen, D.Z., 1989. An optimal parallel algorithm for the minimum circle cover problem. *Information Processing Letters*, **32**:159-165.

Bertossi, A.A., 1988. Parallel circle cover algorithms. *Information Processing Letters*, **27**:133-139.

Booth, K.S., Luekher, G.S., 1976. Testing for the consecutive ones properly, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, **13**:335-379.

Chen, D.Z., Lee, D.T., 1994. Solving the All Pair Shortest-path Problem on Interval and Circular-arc Graphs. Proceedings of the 8th International Parallel Processing Symposium, Cancun, Mexico, p.224-228.

Gabow, H.N., Tarjan, R.E., 1985. A linear time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, **30**:209-221.

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

Gupta, U.I., Lee, D.T., Leung, J.Y.T., 1982. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, **12**:459-467.

Ibarrra, O.H., Wang, H., Zheng, Q., 1992. Minimum Cover and Single Source Shortest-path Problems for Weighted Interval Graphs and Circular-Arc Graphs. Proceeding of the 30th Annual Allerton Conference on Communication, Control and Computing, University of Illinois, Urbana, p.575-584.

Lee, C.C., Lee, D.T., 1984. On a circle cover minimization problem. *Information Processing Letters*, **18**:109-115.

Mishra, P.K., Sharma, C.K., 1997. A Computational Study of the Shortest-path Algorithms in C-Programming Language. Proc. Fifth International Conference on Applications of High Performance Computing in Engineering, Santiago de Compostela, Spain.

Mishra, P.K., Sharma, C.K., 2002. An efficient implementation of scaling algorithms for the shortest-paths problem. *News Bulletin of Calcutta Mathematical Society*, **27**:342-351.

Mishra, P.K., 2004. An efficient parallel algorithm for shortest-paths in planar layered digraphs. *Journal of Zhejiang University SCIENCE*, **5**(5):518-527.