On the Shooter Location Problem: Maintaining Dynamic Circular-Arc Graphs

Stefan Langerman
Computer Science
Rutgers University

Abstract

We present a data structure for maintaining the Minimum Clique cover and Maximum Independent Set of a circular-arc graph. These functions can be maintained during the interchange of adjacent arc endpoints in $O(\log n)$ amortized time per interchange. The new data structure extends the dynamic tree structure of Sleator and Tarjan [5] so that cutting and linking up to $O(n)$ subtrees can be done in one $O(\log n)$ time operation. We also give as an application and motivation for our data structure, an $O(n^4 \log n)$ algorithm for a generalized version of the shooter location problem [7]. The best previous algorithm has complexity $O(n^5)$ for a restricted version of this problem [2].

1 Introduction

A shooter at position $x \in R^2$ aims along a ray $r_x(\delta) = \{x+t\delta, t > 0\}$ in direction δ and hits all objects incident to the ray with a single shot. Given a set of n segments to be shot, the shooter-location-problem is to find a position x from which the fewest shots are needed.

An efficient solution to this problem required the development of a new data structure - the main contribution of this paper - which should be of independent interest. The endpoints of the segments define $N = \binom{2n}{2}$ lines which partition the plane into a complex of $O(n^4)$ convex cells, the arrangement of the N lines. In each cell, all points require the same minimum number of shots, and that value is obtained by finding the minimum clique cover of the circular arc graph determined by the segments and that cell. The new data structure allows one to "walk" from cell to cell in the arrangement, adjust the circular arc graph, and update the known solution from the neighboring cell, all in $O(\log n)$ amortized time per cell. This gives an $O(n^4 \log n)$ algorithm, improving a previous $O(n^5)$ one.

A *circular-arc family* is a set of arcs on a circle. A graph $G = (V, E)$ is a *circular-arc graph* if there exists a circular-arc family $F = \{a_v | v \in V\}$ for which $(i, j) \in E$ if and only if a_i and a_j overlap. Observe that a circular-arc graph is fully described by the clockwise ordering of the endpoints of the arcs of its corresponding circular-arc family. Circular-arc graphs are of independent interest. A simple bibliography search on the web returned more than a hundred hits on the subject. Circular-arc graphs are used to represent problems in scheduling, geometry, bioinformatics, and chemistry. They are also a generalization of interval graphs which have a wide range of applications.

A set $S \subseteq F$ of arcs is said to be *independent* if no two arcs in S intersect. The Maximum Independent Set (MIS) problem is to find such a set whose size is maximum. The size of the maximum independent set for a circular-arc family will be written $\alpha(F)$. A set P of points on the circle is a *clique cover* for F if $a \cap P \neq \emptyset, \forall a \in F$. The Minimum Clique Cover problem (MQC) is to find such a set whose size is minimum. $\theta(F)$ will be used to denote the size of that set.

The new data structure is designed to maintain efficiently MIS and MQC during the interchange of endpoints of adjacent arcs. For this we had to extend the dynamic tree structure of Sleator and Tarjan [5] so that cutting and linking $O(n)$ subtrees could be accomplished within the same $O(\log n)$ time bounds. We believe this data structure to be of independent interest. In particular, it could be used to dynamically maintain the graph of any function of a set to itself.

In the next section, we describe a greedy algorithm due to Hsu and Tsai [3] for MIS and MQC, and explain why we want to use dynamic trees for fast updates. In section 3, we present the details of the new dynamic tree data structure, and in section 4, we show how to use it to maintain the MIS and the MQC. In section 5, we solve a generalized version of the shooter location problem: Instead of segments, we are given a set of n convex polygons $S_1, ..., S_n$ in the plane, and look for a point $x \in R^2$ from which all polygons can be hit with the fewest number of shots. As long as $|S_i| < n^{(n^2)}$, we can find an optimal location in $O(n^4 \log n)$. For the case where $|S_i| = 2$ (segments), Wang and Zhu [7] gave an $O(n^5 \log n)$ algorithm which was improved to $O(n^5)$ by Chaudhuri and Nandy [2]. We believe our method could be successfully applied to other geometric problems.

2 Circular-Arc Graphs

In [3], Hsu and Tsai describe a very nice greedy algorithm for finding the MIS and the MQC of a circular-arc family in $O(n)$ time. In this section, we will outline the main ideas of their algorithm, and outline a strategy for allowing fast updates following an interchange of arc endpoints.

First, we need to introduce some notation. An arc i will be represented by its two extreme points (p, q) in clockwise order. p will be called the *head* of i, or $h(i)$, and q will be called the *tail* of i, or $t(i)$. We will also use the notation (a, b) to denote the set of points on the circle lying in the clockwise trajectory from a to b.

Definition 1 (Greedy Independent Set) *Given an arc i, define $NEXT(i)$ to be an arc j such that $h(j) \in (t(i), t(j))$, and whose tail is the first encountered in a clockwise transversal from $t(i)$. Define $GD(i)$ to be the maximal independent set of the form $\{i = i_1, i_2, ..., i_k\}$, where $i_t = NEXT(i_{t-1}), t = 2, ..., k$. Define $LAST(i)$ as $NEXT(i_k)$.*

We then define the directed graph $D = (V(D), E(D))$, whose vertex set is $V(D) = F$, and $E(D) = \{(i, j) | j = NEXT(i)\}$. Note that the out-degree of every vertex in D is exactly 1, and thus, D is a set of connected components, each of which contains exactly one cycle. We call a node of D (or the corresponding arc in F) *good* if $|GD(i)| = \alpha(F)$.

Theorem 1 (Hsu and Tsai [3]) *Let C be a directed cycle in D. Each node in C is a good node.*

This theorem already gives an algorithm for MIS: find a node i in any cycle of D, and output $GD(i)$. The following definition and theorem give a similar result for the MQC problem.

Definition 2 (Greedy Clique Cover) *Given an arc i in a cycle of D, let $\{i = i_1, i_2, ..., i_{\alpha(F)}\} = GD(i)$. Define $GD_q(i)$ to be $\{t(i_1), t(i_2), ..., t(i_{\alpha(F)})\} \cup \{t(LAST(i))\}$. So, $|GD_q(i)|$ is $\alpha(F)$ if $LAST(i) = i$, and $\alpha(F) + 1$ otherwise.*

Theorem 2 (Hsu and Tsai [3]) *Let i be a node in a cycle C of D, then $GD_q(i)$ is an MQC for F*

Since we want to output $\alpha(F)$ without having to list a corresponding independent set, we will also need the following. Define the *density* of a cycle C at $t(i)$ to be the number of arcs of C that contain $t(i)$. The density s of the cycle C is the maximum density over all tails of arcs in C.

Lemma 1 (Hsu and Tsai [3]) *If $|C| > \alpha(F) > 1$, then $\alpha(F) = \left\lfloor \frac{|C|}{s} \right\rfloor$.*

In order to compute the density of a cycle, we prove the following:

Lemma 2 *For any point p on the circle, the density s of a cycle C is $\hat{s} = |\{(i, j) \in C | p \in (t(i), t(j))\}|$*

Proof: First observe that $\hat{s}$ is the same for any p. To see this, walk on the circle from p in clockwise direction until you meet the next $t(i)$ for some i in C. Let $j \in C$ be such that $i = NEXT(j)$. After crossing $t(i)$, our moving point, which used to be in $(t(j), t(i))$ is not in that interval anymore, but is in $(t(i), t(NEXT(i)))$, so the count of $\hat{s}$ hasn't changed.

Next, we can see that $\hat{s} \geq s$ by choosing p to be the tail of highest density in C. We complete the proof by showing that $\hat{s} \leq s$. Let $t(i)$ be the tail of highest density in C, and suppose $\hat{s} > s$. Let S be the set of all arcs in C that contain $t(i)$.

For all $j \in S$, there is a $l_j \in C - S$ for which $j = NEXT(l_j)$ (since they all intersect, the predecessor of j cannot be in S), and $t(i) \in (t(l_j), t(j))$. Since $\hat{s} > s$, there is one more edge (u, v) in C such that u and v are not in S, and $t(i) \in (t(u), t(v))$. But we know that $t(u) \in (h(j), t(i))$ for all j in S, otherwise, either $NEXT(u) = j$ or $NEXT(l_j) = v$ (note that $t(i) \in (h(j), h(v))$). So, the density of $t(u)$ is $s + 1$, which contradicts the fact that t(i) was of highest density. □

In our data structure, we will keep a representation of D that permits finding a cycle and its length quickly. Every edge (i,j) of D such that $0 \in (t(i), t(j))$ for some arbitrary point 0 on the circle, will be marked with a cost of 1. Our data structure will be able to rapidly compute the sum of the edge costs on a cycle. This will allow us to find $\alpha(F)$ and $\theta(F)$ within the claimed time bound. We now investigate the different changes that can occur as two adjacent arc endpoints swap positions. The possible cases are described by the change of the clockwise order of the endpoints for two arcs a and b:

1. $..., h(a), h(b), ... \Rightarrow ..., h(b), h(a), ...$ This doesn't change anything to D, since the $NEXT$ function depends mainly on tails.
2. $..., t(a), h(b), ... \Rightarrow ..., h(b), t(a), ...$ If $b = NEXT(a)$, then recompute $NEXT(a)$, and modify D accordingly.
3. $..., h(b), t(a), ... \Rightarrow ..., t(a), h(b), ...$ Inverse of the previous case: if $t(b)$ should become the new $NEXT(a)$, modify D accordingly.
4. $..., t(a), t(b), ... \Rightarrow ..., t(b), t(a), ...$ All j such that $NEXT(j) = a$ change to b if $t(j) \in (t(a), h(a))$. So this might require to move $\Theta(n)$ edges of D.
5. A single segment swaps its two endpoints (doesn't produce any change).
6. $..., t(a), 0, ... \Rightarrow ..., 0, t(a), ...$ Unmark the outgoing edge of a, mark all its incoming edges.

7. $..., 0, t(a), ... \Rightarrow ..., t(a), 0, ...$ Unmark the incoming edges of a, mark its outgoing edge.
8. swap of 0 and $h(a)$. Do nothing.

D is a set of connected components, each being a tree with one extra edge (forming the cycle). We will thus represent D using Sleator and Tarjan's dynamic tree structure, which we will need to modify to allow moving a large number of edges at once, as required by operation 4. We will also include in that data structure a way of keeping track of the extra edge, and a way to move the root of the tree to any node in the cycle.

3 Extended Dynamic Tree Structure

3.1 Definition

Sleator and Tarjan's dynamic tree structure [5] maintains a forest of vertex-disjoint trees with a real valued *cost* associated with each edge, and supports the following operations in $O(\log n)$ time:

parent(**vertex** v)**:** returns parent of v.

root(**vertex** v)**:** returns root of the tree containing v.

cost(**vertex** v)**:** returns cost of the outgoing edge of v.

mincost(**vertex** v)**:** returns a minimum cost edge on the path from v to $root(v)$.

update(**vertex** v, **real** x)**:** add x to the cost of every edge on the path from v to $root(v)$.

link(**vertex** v, w, **real** x)**:** link the tree rooted at v to the node w of a different tree. The cost of the added edge is x.

cut(**vertex** v)**:** Divide the tree containing v into two parts, by deleting the edge linking v to its parent. Return the cost of the outgoing edge of v.

Since we want to be able to move $O(n)$ edges in one operation, we need a way to designate the set of edges in $O(1)$ space, and retrieve the corresponding set in $O(\log n)$ time. We also need to be able to merge those sets in $O(\log n)$ time. For this, we will label each tree nodes v with a positive real number $l_v \in [0, r)$ (r can be chosen arbitrarily). The *between* expression $v \in [a, b]$ for $a, b \in [0, r)$ is true if $a \leq b$ and $l_v \in [a, b]$, or if $b < a$ and $l_v \notin (a, b)$. We will impose the following restriction on our tree: if the nodes u and v have the the same parent p, then p is the parent of all $w \in [l_u, l_v]$.

We will keep all children of a node p in a structure called *List Of Trees* (LOT), implemented as *splay trees*(splay trees are self adjusting binary search trees, see [6]). We will enforce the above restriction for any LOT the algorithm will manipulate: if u and v are in a LOT and $w \in [l_u, l_v]$, then w is in that same LOT. Furthermore, w will be placed between u and v in the LOT, if possible (the only case where this will not be possible is when the LOT contains all nodes in the forest). Tree roots can also be grouped in LOTs as long as these LOTs satisfy the same betweenness conditions. Tree roots not in any LOT will be wrapped in a one-node LOT. Given a LOT L, $lparent(L)$ will contain a pointer to the parent of its elements, or *null* if it is a LOT of roots. $kids(v)$ will contain the LOT of the incoming edges to vertex v. The LOT structure will also contain the cost of the outgoing edges of its elements. If those elements are root nodes, the cost will still be kept for an eventual future *link* operation.

Since we also want to be able to keep track of an extra edge, going from the root to an internal node of its tree, we will allow a root node to be an element of the LOT of one of its descendants, but the edge will be marked *invisible*, and no operation will be done on it. A bit $isroot(v)$ will be stored in every node v to identify the root of the tree (and the fact that its outgoing edge is invisible). The *link* and *cut* operations will be changed to the following:

link(**LOT** L, **vertex** w)**:** make w the new parent of all the elements of L. If w already had children, this assumes that $parent(L) = null$, that the set of children after the link will satisfy our tree restrictions, and that w is not in any of the trees in L.

cut(**vertex** v, **real** a, b)**:** find all children u of v for which $u \in [a, b]$. Delete the edge linking them to v, and return the LOT containing those cut nodes. This assumes that the set of children after the cut will satisfy our tree restrictions, and that $root(v)$ is not in any of the cut LOTs, or their descendants.

We will also define the following operations:

addcost(**LOT** L, **real** x)**:** add x to the cost the outgoing edges of all nodes in LOT L.

totcost(**vertex** v)**:** return the sum of the cost of all edges on the path from v to $root(v)$.

depth(**vertex** v)**:** return the number of nodes on the path from v to $root(v)$.

isincycle(**vertex** v)**:** returns true if v is in the cycle formed by the extra (invisible) edge.

setcycle(**vertex** v)**:** sets the invisible edge from $root(v)$ to v. This assumes that the LOT of $root(v)$ only contains the root.

moveroot(**vertex** r)**:** Moves the root of the tree to r. assumes $isincycle(r)$ returns true.

The restrictions described above are not very limiting: it would be quite easy to relabel nodes, or even entire LOTs in order to link them anywhere we want. The only purpose of the labeling system is to be able to quickly describe and retrieve large sets of nodes.

3.2 Implementation Sketch

As in [5], our trees will be decomposed into a set of vertex disjoint paths. Paths support the following operations:

path(**vertex** v)**:** return the path containing v.

head(**path** p)**:** return the head (bottom node) of p.

tail(**path** p)**:** return the tail (top node) of p.

$before$(**vertex** v)**:** return the predecessor of v, or *null* if v is the head.

$after$(**vertex** v)**:** return the successor of p, or *null* if v is the tail.

pcost(**vertex** v)**:** return the cost of the outgoing edge of v.

pmincost(**path** p)**:** return the vertex v of minimum out degree edge cost, that is closest to *tail*(p) in p.

pupdate(**path** p, **real** x)**:** add x to the cost of every edge in p.

concatenate(**path** p, q, **real** x)**:** join paths p and q by adding the edge (*tail*(p), *head*(q)), with cost x.

split(**vertex** v)**:** delete the edges incident to v, and return the tuple (p,q,x,y), where p is the path from *head*(*path*(v)) to $before(v)$, q is the path from $after(v)$ to *tail*(*path*(v)), and x and y are the cost of the deleted edges linking v to p and q respectively.

The data structure for paths will be that used by Sleator and Tarjan [5]. We will just have to add the two functions *ptotcost*(path p), computing the sum of the cost of all edges in p, and *plength*(path p), returning the number of nodes in p. Both can be trivially implemented as a variation of *pmincost*(.).

As mentioned before, LOTs will be implemented as splay trees. A splay tree [6] is a self adjusting binary search structure based on the *splay*(tree T, node v) operation, which brings the node v to the root of the tree T, while maintaining the inorder of the nodes. We will store in the root of the splay tree, the name of the LOT it corresponds to. The operation $lotof$(vertex v) will return the LOT of the splay tree containing v by performing a splay operation at v. We will use the operation *split*(LOT T, real a, b) to detach the nodes $u \in [a, b]$ from the tree T, and return the detached tree. This can be done with at most two splay operations. *join*(LOT L_1, L_2) is a standard operation for splay trees. We will assume it joins the two LOTs in the correct order. *last*(LOT L) and $first$(LOT L) will return the rightmost and leftmost nodes in L respectively.

Each node v of the splay tree will contain a real number $c(v)$, and the cost of the outgoing edge of a node v will be the sum of the c's on the path from the root to v. The splay tree algorithm can be modified to maintain the correct c values after splay operations. This makes the implementation of *addcost*(L, x) a simple matter of adding x to the c of the root of L. Retrieving the cost of the outgoing edge of v (operation *getcost*(v) comes down to splaying at that node and returning its c. We will also use the operation *setcost*(vertex v, real x), that will set the cost of the outgoing edge of v to x in the splay tree, without changing the value of the other costs in the tree (this is done by removing from the c values of the children of v the same value added to the c value of v).

The tree edges (excluding the invisible edge) will be partitioned into two kinds: solids and dashed, and no more than one solid edge enters a given vertex. The dashed edges will be represented using the LOT data structure as explained earlier. The dark edges will be managed using path operations. Since the cost of a node may change using *pupdate*(.) or *addcost*(.), we make the convention that the cost of the outgoing edge of v will not be valid in the $lotof(v)$ as long as that edge is solid. As soon as this edge becomes dashed, its cost in its LOT will be updated.

Two operations organize the path structure:

splice(**path** p)**:** extend the solid path by converting to solid the dashed edge leaving *tail*(p), and converting to dashed the solid incoming edge of $parent$(*tail*(p)) (if any). Return the extended path. This assumes $isroot$(*tail*(p)) is false.

$expose$(**vertex** v)**:** construct a solid path from v to $root(v)$ by converting the solid incoming edge of v (if any) to dashed, and applying *splice*(.) repeatedly on *path*(v).

The reader can check the details of the operations given in Appendix A.

3.3 Running Time

In order to avoid having to reproduce the amortized cost analysis from [5], we made sure each configuration produced by the above operations could be produced by a sequence of operations in the original data structure. One may then verify that the main proofs go through, namely that:

1. The total number of splices over a sequence of m operations is $O(m \log n)$.

2. A sequence of m tree operations takes $O(m \log n)$ time, plus the time required by all splay operations.

So we just need to bound the cost of the splay operations. Assign to every node v of a splay tree a weight $w(v)$ whose value is arbitrary, but fixed. The size of v, $s(v)$, is the sum of the weights of all the nodes in the subtree rooted at v. Let the rank be $r(v) = \log(s(v))$.

Lemma 3 (Access lemma [6]) *The amortized time to splay a tree with root t at node v is $O(r(t) - r(v))$.*

For every LOT node v, v is also a node in our dynamic forest. Let $w(v)$ be equal to the number of nodes of the forest contained in the subtree (in the dynamic tree structure, not the splay tree) rooted at v. Since the nodes in a LOT correspond to disjoint subtrees, we know that the size of a node cannot exceed n. Therefore a single splay operation on a LOT will run in $O(\log n)$ amortized time. Notice that all tree operations execute a constant number of *splay* and *expose* operations. Thus, all we need to do is to bound the cost of the splays in an *expose*(.) operation.

Lemma 4 *The amortized cost of all splay operations during a single expose is $O(\log n)$.*

Proof: The splays done during *expose*(v) are all done along the path from v to *root*(v). Let $t_1, v_1, t_2, v_2, ..., t_k, v_k$ be the nodes involved in those splays, where v_i is the i^{th} node being splayed, and t_i is the root of that splay tree. Then, the amortized cost of all the splays is $r(t_1) - r(v_1) + r(t_2) - r(v_2) + ... + r(t_k) - r(v_k)$. But since t_i is below v_{i+1} in the tree, the subtree rooted at v_{i+1} contains the subtree rooted at t_i, and so, $s(t_i) \leq s(v_{i+1})$, and $r(t_i) \leq r(v_{i+1})$. Therefore, the sum telescopes, and the amortized cost is less than $r(t_k) - r(v_1)$ which is $O(\log n)$. □

4 Maintaining the Minimum Clique Cover

First, we will determine fixed integer labels for the circular-arc endpoints. Put down a point named 0 on the circle, and list the endpoints (tails and heads) in clockwise order starting at 0. Assign to those endpoints the labels $1, 2, ..., 2n$ in the order in which they were listed. $h(v)$ and $t(v)$ will be the labels assigned to the head and the tail of v respectively. Swapping two adjacent endpoints will be done by swapping the labels of those endpoints.

We need a data structure to compute the $NEXT$ function efficiently, and that can be updated in $O(\log n)$ time per swap. Create an array M of size $2n$, and store in the first n places the labels of the n tails in sorted order. Store with each label $t(i)$ in the array, the name i of the corresponding circular-arc, and the label $h(i)$ if $h(i) < t(i)$ or $h(i) - (2n + 1)$ otherwise. The three components of the array at position j will be written $M[j].t$, $M[j].v$, and $M[j].h$. For $j = n + 1, ..., 2n$, we set $M[j].t = M[j-n].t+(2n+1)$, $M[j].v = M[j-n].v$, and $M[j].h = M[j - n].h + (2n + 1)$.

We next build a static binary tree on top of this array, with a value stored in each node. The leaves of the tree are the $2n$ elements of the array, and the value of the leaf for $M[j]$ is $M[j].h$. The value for an internal node is the maximum of the values of its children. To compute $NEXT(i)$, walk down the tree, always to the leftmost child with a value $> t(i)$.

Whenever swapping any two tails whose tail labels are at $M[a].t$ and $M[b].t$, exchange $M[a].v$ with $M[b].v$, and $M[a].h$ with $M[b].h$. Likewise, when swapping two heads, just swap $M[a].h$ with $M[b].h$. Swapping a head and a tail will never change the ordering of the tails, so just change the corresponding labels.

Whenever modifying head labels in the array, update the values on the path from those leaves to the root of the tree. This can all be done in $O(\log n)$ per operation. The zero (0) position will start with the value 0.5, and will be moved by increments of $+1$ or -1. Add or subtract $2n$ to its value to keep it in the $[0, 2n]$ range. It might seem strange that we give a nonzero value to the "zero" point, but this is consistent with the rest of our scheme, as we are mapping all endpoints to different values. The values only exist to preserve the order of the points on the circle.

In the preprocessing phase of the algorithm, the graph D will be constructed explicitly in our dynamic tree structure. This can be done in $O(n \log n)$ time. The updates to be performed in the tree structure after the $NEXT$ data structure is updated are described in details in Appendix B.

The size of the MIS and MQC can be found using the following for any root r:

$$\alpha(F) = \left\lfloor \frac{totcost(parent(r)) + cost(r)}{depth(parent(r))} \right\rfloor$$

$$\theta(F) = \left\lceil \frac{totcost(parent(r)) + cost(r)}{depth(parent(r))} \right\rceil$$

5 The Shooter Location Problem

Given a point p on the plane and n convex polygons, the projection of the convex polygons on a unit circle centered at p forms a circular-arc family $F(p)$. The number of rays necessary to hit every polygon at least once from p is exactly $\theta(F(p))$.

For any pair of convex polygons of size s_1 and s_2, one can compute the four common tangents to those polygons in $O(\log s_1 + \log s_2)$ time, using a binary search algorithm [4]. Notice that as a point p crosses one of those lines, the endpoints of two circular-arcs in $F(p)$ are swapped.

Let the sizes of our n polygons be $s_1, ..., s_n$. We can compute the common tangents of every pair of polygons in $O(n.\sum_{i=1}^{n}\log s_i)$. These $O(n^2)$ lines form an arrangement dividing the plane into $O(n^4)$ cells. Using the topological walk algorithm [1], one can walk from cell to cell, crossing one line at each step, and cover the entire arrangement in $O(n^4)$ steps, using $O(n^2)$ space. Using our data structure, we can produce $\theta(F(p))$ as p moves from cell to cell in $O(\log n)$ amortized cost per step, and so,

Theorem 3 *The cell with maximum $\theta(F(p))$ (optimal shooter location) can be found in $O(n^4 \log n + n\sum_{i=1}^{n}\log s_i)$ time and $O(n^2)$ space.*

In particular, if $s_i < n^{(n^2)}$, the running time is $O(n^4 \log n)$.

Acknowledgments

I would like to thank Bill Steiger for his many useful comments.

References

[1] Tetsuo Asano, Leonidas J. Guibas, and Takeshi Tokuyama. Walking in an arrangement topologically. *Internat. J. Comput. Geom. Appl.*, 4(2):123–151, 1994.

[2] J. Chaudhuri and S. C. Nandy. Generalized shooter location problem. *Lecture Notes in Computer Science*, 1627:389–399, 1999.

[3] Wen-Lian Hsu and Kuo-Hui Tsai. Linear time algorithms on circular-arc graphs. *Inform. Process. Lett.*, 40(3):123–129, 1991.

[4] Franco P. Preparata and Michael Ian Shamos. *Computational geometry.* Springer-Verlag, New York, 1985. An introduction.

[5] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[6] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[7] Cao An Wang and Binhai Zhu. Shooter location problems revisited. In *Proc. 9th Canad. Conf. Comput. Geom.*, pages 223–228, 1997.

A Implementation of the Tree Structure

```
function splice(path p)
    vertex v; path q, r; real x, y;
    v ← lparent(lotof(tail(p)));
    (q, r, x, y) ← split(v);
    setcost(v, y);
    if q ≠ null then setcost(tail(q), x) endif;
    p ← concatenate(p, path(v), getcost(tail(p)));
    if r ≠ null then p ← concatenate(p, r, y) endif;
    return p;
end splice

function expose(vertex v)
    path p, q, r; real x, y;
    (q, r, x, y) ← split(v);
    if q ≠ null then setcost(tail(q), x) endif;
    if r ≠ null then setcost(v, y) endif;
    p ← path(v);
    while not(isroot(tail(p)))
            and lparent(lotof(tail(p))) ≠ null do
        p ← splice(p) end;
    return p;
end expose
```

The implementation of *root*(.), *mincost*(.) and *update*(.) is identical to the one in [5]. The implementation of the other operations is as follows:

```
function parent(vertex v)
    return lparent(lotof(v));
end parent

function cost(vertex v)
    if v = tail(path(v)) then return getcost(v);
    else return pcost(v) endif;
end cost

procedure link(LOT L, vertex v)
    path p; vertex u;
    p ← expose(v); u ← last(L);
    if isroot(u) then u ← first(L) endif;
    L ← join(L, kids(v));
    lparent(L) ← v;
    concatenate(path(u), p, getcost(u));
end link
```

Note that the solid/dashed state of the tree after this *link* operation will be exactly the same as if we had performed an old *link* on all elements of L separately.

```
function cut(vertex v, real a, b)
    LOT L;
    expose(v);
    L ← split(kids(v), a, b);
    lparent(L) ← null;
    return L;
end cut

function totcost(vertex v)
    return ptotcost(expose(v))
end totcost
```

```
function depth(vertex v)
    return plength(expose(v))
end depth

function isincycle(vertex v)
    vertex r, u;
    r ← root(v); /*note that this exposes v*/
    u ← lparent(lotof(r));
    if v = u return true;
    else
        expose(u);
        if head(path(v)) = v then return false;
        else return true endif;
    endif;
end isincycle

procedure setcycle(vertex v)
    vertex r;
    r ← root(v);
    L ← join(kids(v), lotof(r))
    lparent(L) ← v;
end setcycle

procedure moveroot(vertex v)
    vertex r;
    r ← root(parent(v));
    isroot(v) ← true;
    isroot(r) ← false;
    expose(parent(v));
end moveroot
```

Note that a *moveroot* operation produces the same solid/dashed configuration as a *cut* at v, followed by a *link* of r. This is important for the running time analysis.

B Circular-Arc Graph Updates

$..., t(a), h(b), ... \Rightarrow ..., h(b), t(a), ...$

```
    LOT L; vertex c;
    if parent(a) = b then
        c ← NEXT(a);
        if isincycle(a) then moveroot(a) endif;
(*)     L ← cut(kids(b), t(a), t(a));
        isroot(a) ← true;
        if 0 ∈ [h(b), t(b)] then addcost(L, −1) endif;
        if 0 ∈ [t(a), t(c)] then addcost(L, 1) endif;
        if root(c) = a then
            setcycle(c);
        else
            isroot(a) ← false;
            link(L, c);
        endif;
    endif
```

In line (*), a is the rightmost child of b.

$..., h(b), t(a), ... \Rightarrow ..., t(a), h(b), ...$

```
    LOT L; vertex c;
    if NEXT(a) = b then
        c ← parent(a);
        if isincycle(a) then moveroot(a) endif;
        L ← cut(kids(c), t(a), t(a));
        isroot(a) ← true;
        if 0 ∈ [h(b), t(c)] then addcost(L, −1) endif;
        if 0 ∈ [t(a), t(b)] then addcost(L, 1) endif;
        if root(b) = a then
            setcycle(b);
        else
            isroot(a) ← false;
            link(L, b);
        endif;
    endif
```

$..., t(a), t(b), ... \Rightarrow ..., t(b), t(a), ...$

```
    LOT L;
    if isincycle(a) then moveroot(a) endif;
    L ← cut(kids(a), t(b), h(b));
    if root(parent(a)) ≠ a then
        isroot(a) ← false;
    endif;
    if lotof(root(b)) = L then
        isroot(root(b)) = true;
        if left(L) = right(L) then setcycle(b);
        else link(L, b) endif;
    else
        link(L, b);
    endif;
```

$..., t(a), 0, ... \Rightarrow ..., 0, t(a), ...$

```
    expose(parent(a));
    setcost(a, 0);
    addcost(kids(a), 1);
```

$..., 0, t(a), ... \Rightarrow ..., t(a), 0, ...$

```
    addcost(kids(a), −1);
    expose(parent(a));
    setcost(a, 1);
```