Original Lecture #8: 4 February 1992
Topics: Shortest Path Problems
Scribe: Jim Stewart

1 Shortest Path Problems

The topic of this lecture is the application of triangulations to shortest path problems. Only simple polygons will be considered. Given two points inside a polygon, the goal is to find the shortest path lying *inside* the polygon between the two points; see Fig. 1. One can think of the shortest path as a taut string connecting the two points.

We begin with some definitions. Let P be a simple polygon with n vertices. Then $\pi(p, q)$ is the Euclidean shortest path inside P connecting two points p and q inside P. Also, $d(p, q) = |\pi(p, q)|$, which is the length of $\pi(p, q)$. $d(p, q)$ is a metric and is more commonly referred to as the *geodesic distance*.

Lemma 1. *$\pi(p, q)$ is unique.*

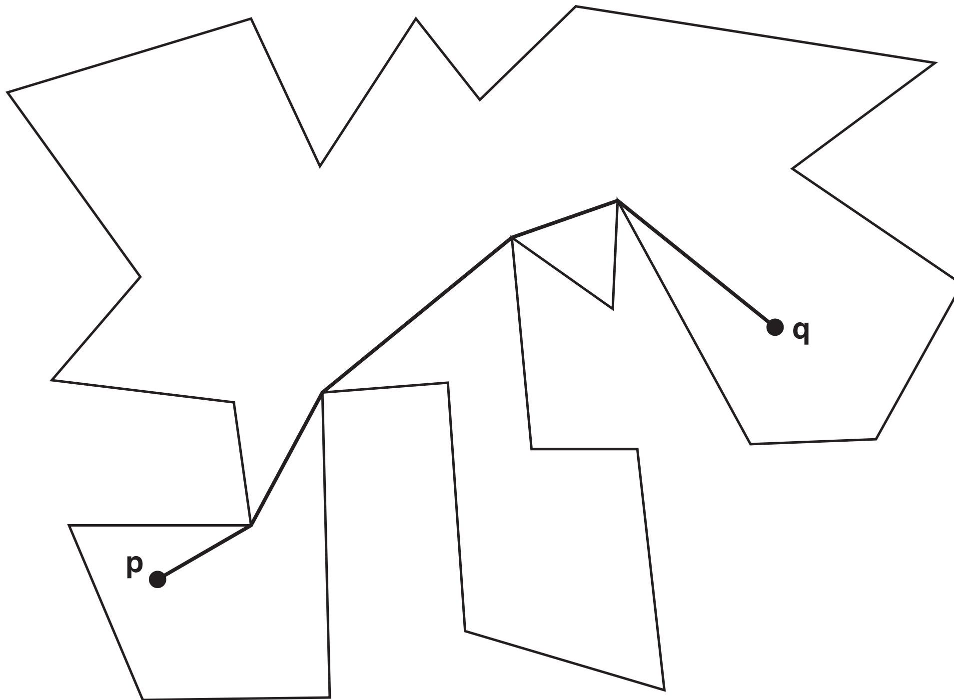

Figure 1: The shortest path between points p and q.

Proof: If $\pi(p,q)$ is not unique, then let $\pi_1(p,q)$ and $\pi_2(p,q)$ be two distinct shortest paths from p to q. Let α and β be two points of $\pi_1(p,q) \cap \pi_2(p,q)$ such that the two paths are disjoint between α and β; that is, $\pi_1(\alpha,\beta) \cap \pi_2(\alpha,\beta) = \{\alpha,\beta\}$. We have $|\pi_1(\alpha,\beta)| = |\pi_2(\alpha,\beta)| = d(\alpha,\beta)$. The two paths $\pi_1(\alpha,\beta)$ and $\pi_2(\alpha,\beta)$ enclose some region of P's interior, free of obstacles, since P is simple. At least one of the two paths has a convex corner; cutting off the corner shortens the path, which is a contradiction, since the path was chosen to have minimum length. □

Lemma 2. *$\pi(p,q)$ is a polygonal chain whose internal vertices are vertices of P, and whose edges are visibility edges or polygon edges (i.e., each edge of the path connects two vertices of P and does not intersect the exterior of P).*

2 Funnels

The concept of a *funnel* is used to compute shortest paths. A funnel is defined as follows. Let α and β be the endpoints of an edge or diagonal of P, and let s be a point inside P. The shortest paths $\pi(s,\alpha)$ and $\pi(s,\beta)$, together with $\overline{\alpha\beta}$, define a region called a *funnel*. Funnels possess the following properties.

1. $\pi(s,\alpha)$ and $\pi(s,\beta)$ share a common initial chain, then separate at a vertex a, called the *apex* of the funnel (by Lemma 1), as shown in Fig. 2.
2. The region bounded by $\pi(a,\alpha)$, $\pi(a,\beta)$ and $\overline{\alpha\beta}$ is contained in P. (For simplicity, assume s, α and β are vertices of P. $\pi(s,\alpha)$ excludes one chain of the boundary of P and $\pi(s,\beta)$ excludes the other; therefore the funnel interior does not contain any edges of P.)
3. $\pi(a,\alpha)$ and $\pi(a,\beta)$ are inward convex (any outward convex portions could be removed to form a shorter path).

Triangulation

Our shortest path algorithms are based on triangulation. Therefore, we assume from now on that P has been triangulated. Let $\tau(p)$ be the triangle containing p, and $\tau(q)$ be the triangle containing q. We define $\tau(\pi(p,q))$ to be the minimal set of triangles containing $\pi(p,q)$.

Lemma 3. *$\tau(\pi(p,q))$ forms a path in the dual tree of the triangulation. This sequence of triangles is called a* sleeve.

Proof: $\pi(p,q)$ crosses each diagonal of the triangulation at most once (else it could be shortened). Therefore, it crosses only the diagonals that separate p and q, each exactly once. □

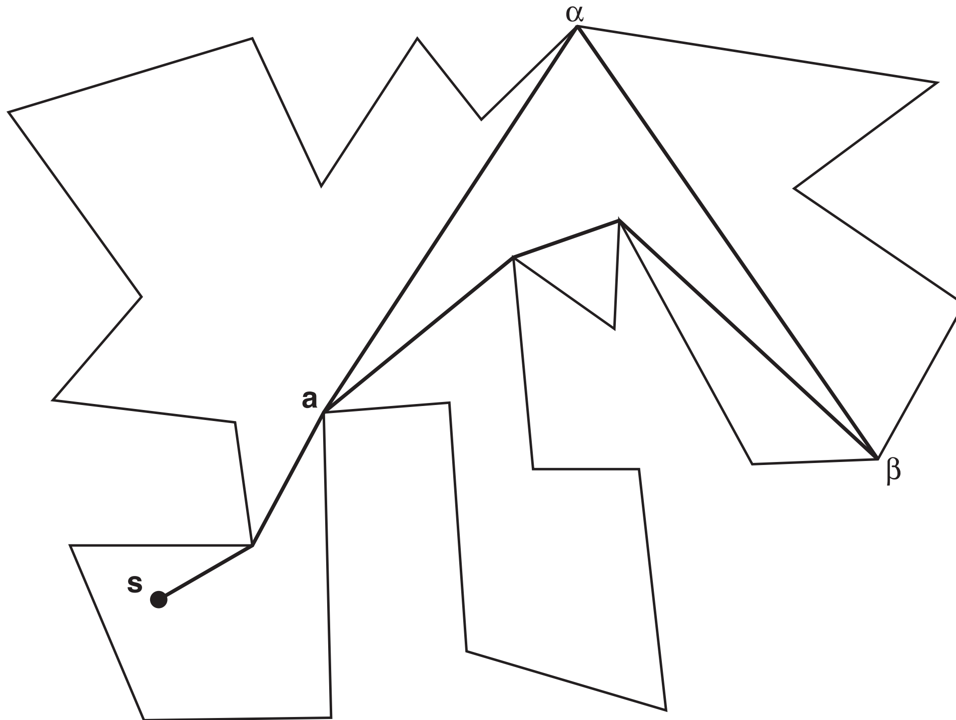

Figure 2: Illustration of a funnel; point a is the apex

3 The Funnel Algorithm for Computing a Shortest Path

The funnel algorithm computes the funnel from p to every diagonal separating p and q in $O(n)$ total time. It marches from p to q, maintaining a current funnel and apex as it goes. In the process, it computes, for every sleeve vertex x, the vertex $\texttt{pred}(x)$ adjacent to x along $\pi(p, x)$. This vertex is the *predecessor* of x; see Fig. 4. The algorithm consists of the following steps.

1. Locate $\tau(p)$ and $\tau(q)$.

2. Identify $\tau(\pi(p, q)) \Longleftrightarrow$ an ordered sequence $d_1, d_2, \ldots, d_m$ of the diagonals separating p and q, in the order that the shortest path crosses them. If $d_m = \overline{\alpha_m \beta_m}$ then set $d_{m+1} = \overline{\alpha_m q}$ (see Fig. 3).

3. The funnel F_i from p to d_i can be represented as a double-ended queue that stores the vertices in order, with the endpoints of d_i at the ends of the queue. The initial funnel is $F_1 = (\alpha_1, p, \beta_1)$, the funnel defined by $\pi(p, \alpha_1)$, $\pi(p, \beta_1)$, and $\overline{\alpha_1 \beta_1}$. The quantity in parentheses is the double-ended queue. Set $\texttt{pred}(\alpha_1) \leftarrow \texttt{pred}(\beta_1) \leftarrow p$, and set the funnel apex, $a \leftarrow p$.

4. For $i \leftarrow 2$ to $m + 1$ do (modify F_{i-1} to get F_i):

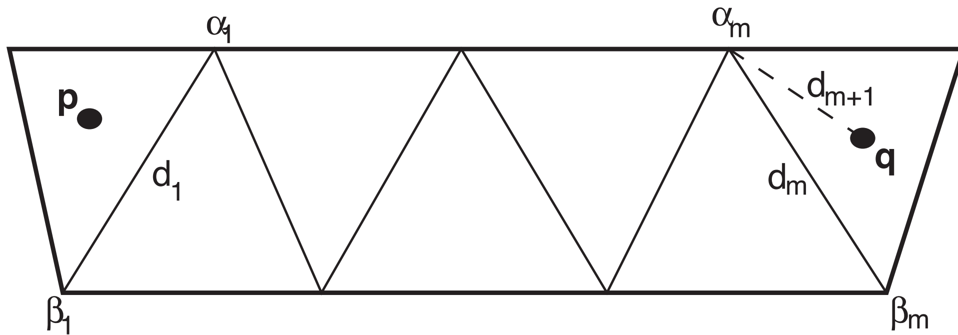

Figure 3: The diagonals $d_1, d_2, \ldots, d_m$ crossed by the shortest path from p to q. Segment d_{m+1} is added so that the algorithm terminates at point q.

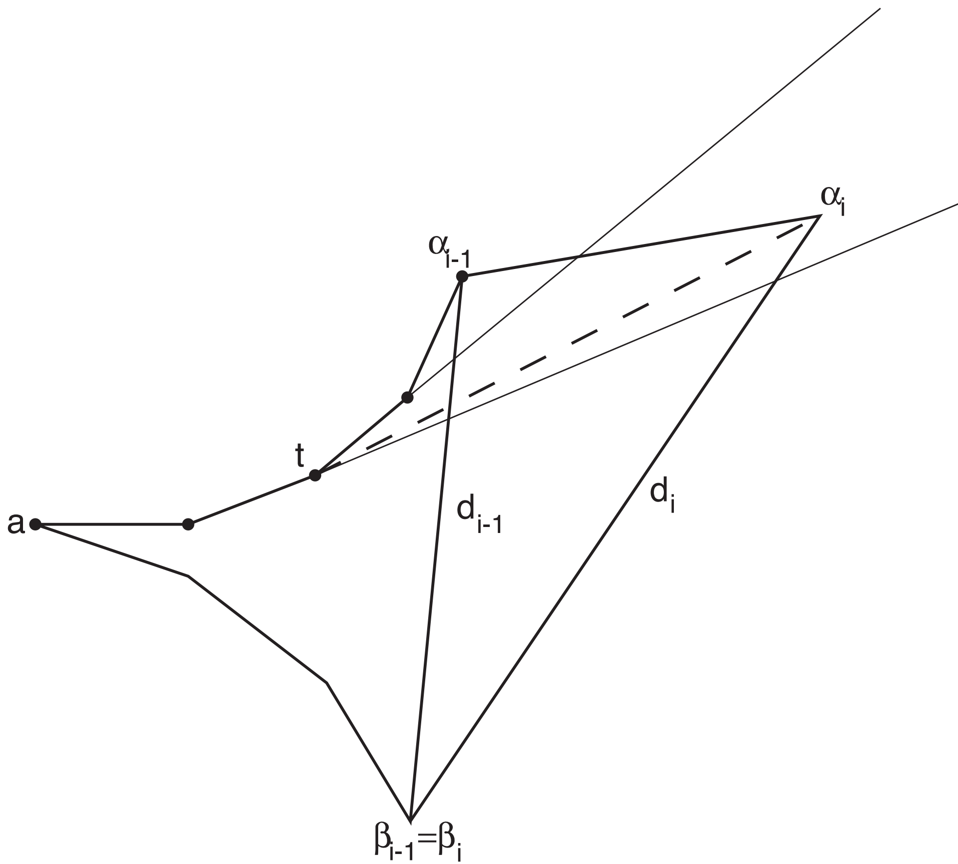

Figure 4: t is the *predecessor* of α_i, i.e., t is the point preceding α_i in the shortest path from p to α_i.

- Without loss of generality assume α_{i-1} is not incident to d_i (see Fig. 4).
- Pop off vertices from the left (L) end of the double-ended queue until the head of this end is the predecessor of α_i on $\pi(\alpha_i, p)$, (To see more concretely which vertex is the predecessor of α_i, consider extending the edges of F_i into rays directed across the diagonal d_{i-1}. These extensions partition the plane into triangular wedges, exactly one of which contains α_i. The vertex at the head of this wedge is the predecessor of α_i. See Fig. 4).
- Set $\texttt{pred}(\alpha_i) \leftarrow \text{Head}(\text{L})$
- If the apex was popped, set $a \leftarrow \text{Head}(\text{L})$
- Push (α_i, L), i.e., add α_i to the funnel by pushing it onto the left end of the double-ended queue.

The $\texttt{pred}(\cdot)$ pointers encode a linked list of the vertices on the shortest path from q to p. Each vertex of the sleeve is pushed exactly once and popped at most once. Pushes and pops dominate the cost. The overall time complexity is linear in m, and $O(m) = O(n)$.

4 The Shortest Path Tree

Let s be a point in P, and let p_i be the vertices of P. Then the *Shortest Path Tree* (SPT) is defined as $SPT = \cup_i \pi(s, p_i)$. It is a tree by Lemma 1. The tree thus contains the shortest path from s to each vertex of P. The following algorithm computes the SPT by recursively splitting funnels. The algorithm is essentially a depth first exploration of the dual tree, starting from $\tau(s)$. Without loss of generality assume s is a vertex of P. Here is the top level of the algorithm (initialization and recursive invocation):

For each triangle incident to s do

- Let d be the edge opposite s, and let α and β be the left and right endpoints of d, respectively, as seen from s.
- Construct the funnel $F = (\alpha, s, \beta)$ with apex $a = s$.
- Set $\texttt{pred}(\alpha) \leftarrow \texttt{pred}(\beta) \leftarrow s$.
- Call $\texttt{Split}(d, F, a)$.

$\texttt{Split}(d, F, a)$ is a recursive procedure that computes a single predecessor vertex, uses that vertex to split the current funnel in two, and calls itself recursively on each sub-funnel. Here is pseudo-code for `Split`:

$\texttt{Split}(d, F, a)$ (d is the current edge or diagonal, F is the funnel to it, and a is the apex of F)

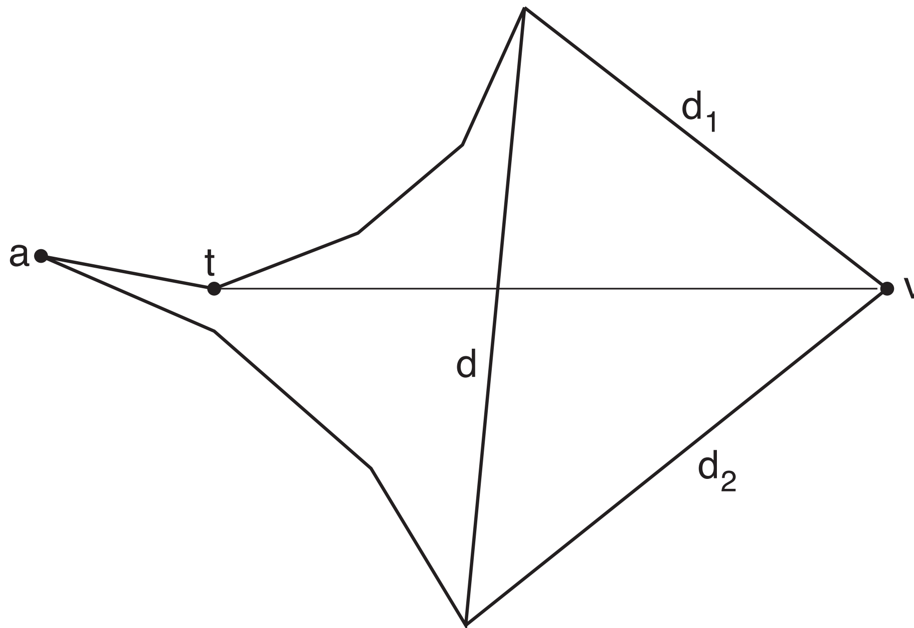

Figure 5: Notation for the definition of the `Split` operator

1. If d is a polygon edge, then return.
2. Let v be the opposite vertex of the triangle beyond d (we know d is a diagonal), and define d_1 and d_2 as shown in Fig. 5.
3. Compute $t = \texttt{pred}(v)$. That is, locate v in one of the triangular sectors defined by the extensions of the funnel edges, as described on page 5.
4. Without loss of generality assume the order of F is $\ldots, t, \ldots, a, \ldots$, and define $\texttt{prefix}(F)$ and $\texttt{suffix}(F)$ as follows:

$$\overbrace{\ldots,t}^{\texttt{prefix}(F)}\underbrace{,\ldots,a,\ldots}_{\texttt{suffix}(F)}$$

5. Adding the edge $\overline{vt}$ splits the funnel F into two smaller funnels; the `Split` operation is repeated on each new funnel:

   Call $\texttt{Split}(d_1, \texttt{prefix}(F) \| v, t)$
   Call $\texttt{Split}(d_2, v \| \texttt{suffix}(F), a)$

   Here the $\|$ operator concatenates v onto the appropriate end of the prefix or suffix of F.

5 Analysis of the SPT Algorithm

If $|F| = m$ and t is in the k'th position of F, the cost of computing $\texttt{pred}(v) = t$ is some function $g(k, m-k)$, which is monotone non-decreasing in each argument. Assume for now that the cost of constructing the arguments for the recursive calls to `Split` is also $O(g(k, m-k))$. We'll justify this assumption below.

How can we analyze the cost of the shortest path tree construction? As a first attempt, we might define $f(m)$ be the worst-case cost of a recursive call on a funnel of size m. Then

$$f(m) \le f(k+1) + f(m-k+2) + g(k, m-k), \qquad 1 \le k \le m$$

But this is no good, since nothing enforces termination. We need to include information from the triangulation structure in the recurrence relation.

To define a better recurrence relation, let m be the current funnel size, and introduce a new parameter r that counts the number of triangles beyond the current diagonal d (that is, on the side of d away from s). The cost of a recursive call to `Split` is then

$$f(r, m) \le f(h, k+1) + f(r-h-1, m-k+2) + g(k, m-k)$$

$$1 \le k \le m, \qquad 0 \le h \le r-1$$

where $f(0, m) = 0$. The sum of the first arguments in the $f(\cdot, \cdot)$ terms on the right hand side shows a decrease of one (because the triangle incident to d is not processed by either recursive call), while the sum of the second arguments shows an increase of three. Using this fact we combine the arguments to get a relation with just a single argument. Let $f(r, m) \equiv C(3r + m)$. Now,

$$\begin{aligned} C(3r+m) &\le C(3h+k+1) + C(3r-3h+m-k-1) + g(k, m-k) \\ &\le C(3h+k+1) + C(3r-3h+m-k-1) \\ &\qquad + g(3h+k+1,\ 3r-3h+m-k-1), \end{aligned}$$

since $k \le 3h+k+1$ and $m-k \le 3r-3h+m-k-1$. Letting $N = 3r+m$ and $K = 3h+k+1$ we get

$$C(N) \le C(K) + C(N-K) + g(K, N-K), \qquad 2 \le K \le N-2,$$

where $C(2) = 0$. Initially $N \le 3n$. The running time of the algorithm is therefore $O(C(3n))$.

6 Implementation of the SPT Algorithm

There are several ways to implement the double-ended queue that represents F:

6.1 Binary search trees

Represent F by a binary search tree. In this case $g(i,j) = O(\log(i+j))$; splitting F at the prefix and suffix costs the same, as does appending v to one end of each subfunnel. This implies $C(N) = O(N \log N)$.

6.2 Linked lists

Use linked lists and a dovetailed linear search from both ends of F. (That is, search from both ends of F simultaneously, alternating steps inward. Stop both searches when either succeeds.) In this case $g(i,j) = O(\min(i,j))$; splitting F is the same cost. Thus $C(N) \leq C(K) + C(N-K) + \min(K, N-K)$.

> **Digression:** How to analyze recurrence relations of the form
>
> $$C(N) \leqslant C(K) + C(N-K) + G(\min(K, N-K)), \qquad 0 < K < N$$
> $$C(1) = 0.$$
>
> This models the cost of recursively partitioning a set when the cost of partitioning is a function of the size of the smaller subset. A convenient way to analyze the cost is to charge each element of the smaller subset for an equal share of the splitting cost. That is, if $K \leq N/2$, charge each element of the K-element set an amount equal to $G(K)/K$. Each element gets charged at most once when it belongs to a set of size $2^i \leq K < 2^{i+1}$, for each i. (The next time it is charged, its set size will be at most $K/2$.) Therefore, the total charge applied to each element is at most
>
> $$\sum_{i=1}^{\log n} G(2^i)/2^{i-1}.$$
>
> If $G(K) = K$, as in the present case, each element is charged at most $\sum_{i=1}^{\log n} 2 = O(\log n)$, for a total cost of $O(N \log N)$.

Thus the total cost of this implementation is proportional to $C(N) = O(N \log N)$.

6.3 Finger search trees or equivalents

Represent funnels by finger search trees or by an array-based scheme with equivalent performance. Finger search trees support searching and splitting on an ordered list in amortized time $O(\log \min(k, m-k))$, where m is the size of the list and k is the rank of the searched-for or splitting element in the list. It follows that $g(i,j) = O(\log \min(i,j))$, and $C(N) = O(N)$ by the charge-based analysis above. (For more information on finger search trees refer to Huddleston and Mehlhorn, *Acta Informatica* **17** (1982), 157–184, or to Mehlhorn, *Data Structures and Efficient Algorithms, Vol. I: Sorting and Searching*, Springer-Verlag (1984).)

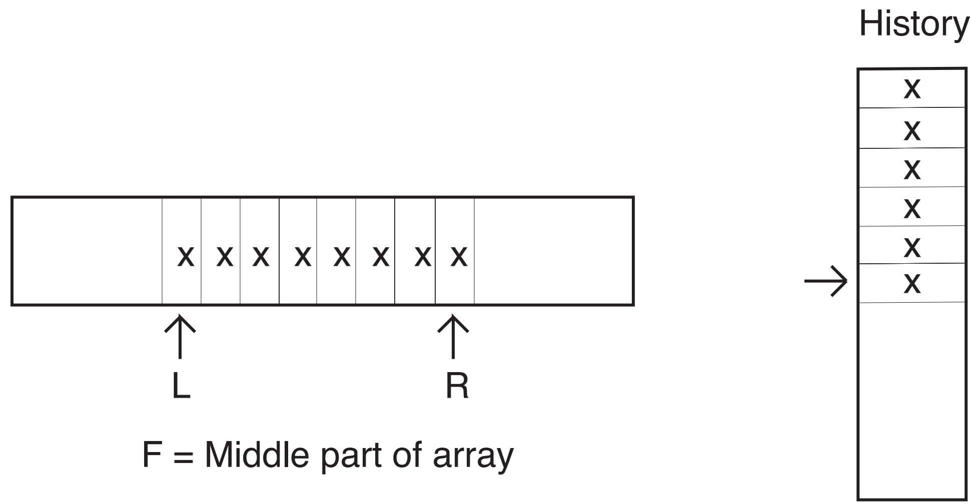

Figure 6: Data structures for array-based searching and splitting.

Finger search trees are complicated and hard to program, so it is worth looking for simpler structures with the same performance. An alternative to finger search trees for the *SPT* algorithm was recently discovered by Hershberger and Snoeyink (*WADS '91*, 331–342).

We can achieve the desired searching performance with a dovetailed doubling search on an array. Suppose F is stored in the middle part of an array, between array indices L and R, as shown in Fig. 6. Then we can search for t from both ends of the queue at the same time, using a search that takes $O(\log d)$ time, where d is the distance of t from the end of the queue where the search started. Both searches stop when either one succeeds, so the total search time is $O(\log \min(k, m-k))$, as desired. Pseudo-code for searching from the left end of the queue is shown below; this search would be dovetailed with a similar search from the right end.

- incr $\leftarrow$ 1
- while goal is outside the subarray $F[L \ldots L + \text{incr}]$ do:
  incr $\leftarrow 2 *$ incr
- Find goal by binary search in $F[L \ldots L + \text{incr}]$

The funnel-splitting algorithm does not need the full power provided by finger search trees. Because it explores the triangulation with a depth first search, it processes one of the two sub-funnels completely before starting to process the other. We exploit this fact by storing the two sub-funnels in the same array, one after the other.

Our algorithm uses three operations for funnel manipulation:

`Add`(v, LR): Push v onto the left or right end of F, as indicated by LR.

`Remove`(k, LR): Remove k elements from the specified end of F.

`Undo`$()$: Reverse the effect of the most recent `Add` or `Remove` operation that has not yet been undone.

These operations are implemented using a *history stack* that records enough information about each `Add` or `Remove` operation to undo it later. Consider the operation `Add`(v, LR). Let $F[k]$ be the array element that will be overwritten (either $k = L - 1$ or $k = R + 1$). We push onto the history stack a record $(\texttt{Add}, LR, F[k])$, then set $F[k] \leftarrow v$ and set the specified one of L and R to be k. To implement `Remove`(k, LR), we push onto the history stack a record $(\texttt{Remove}, LR, k)$, then set $L \leftarrow L + k$ or $R \leftarrow R - k$, as appropriate. To implement `Undo`$()$, we pop the top record off the history stack and reverse the effect of the operation it records.

Here is a version of `Split` that uses these operations. The array representing F has space for $2n$ elements, and we initialize it by putting the three vertices of the first funnel in the middle of the array (so $L = n - 1$ and $R = n + 1$). No more than $n - 2$ vertices are pushed on either end of the array during the algorithm, so the array cannot overflow.

`Split`(d, F, a) (d is the current edge or diagonal, F is the funnel to it, and a is the apex of F)

1. If d is a polygon edge, then return.

2. Define v, d_1, and d_2 as in Fig. 5.

3. Compute $t = \texttt{pred}(v)$. Let t be the k'th element of F. Without loss of generality assume that t is left of a in F.

4. `Remove`$(m - k, \langle\text{right}\rangle)$; `Add`$(v, \langle\text{right}\rangle)$;
   `Split`(d_1, F, t);
   `Undo`$()$; `Undo`$()$;

   `Remove`$(k - 1, \langle\text{left}\rangle)$; `Add`$(v, \langle\text{left}\rangle)$;
   `Split`(d_2, F, a);
   `Undo`$()$; `Undo`$()$;

Observe that each time `Split` is called recursively, the array containing F has been modified to represent the appropriate sub-funnel, as specified in the first version of `Split` on page 5. The three operations `Add`, `Remove`, and `Undo` take $O(1)$ time apiece, so the running time of this algorithm is dominated by the search in step 3, which takes $O(\log \min(k, m - k))$ time. The total running time of the algorithm is therefore $O(n)$, as for finger search trees.

7 The Shortest Path Map

The shortest path map is an enhancement of the shortest path tree. Whereas the shortest path tree encodes the shortest path from every polygon vertex to the source s, the shortest path map encodes the shortest path from every point inside P to s. To produce a shortest path map, given a shortest path tree, we subdivide each funnel of the shortest path tree. For each polygon edge e, we partition the funnel associated with it by extending the funnel edges (as in the predecessor computation on page 5). This partitions the funnel into triangular sectors, each with a distinguished *apex*, namely the vertex opposite e. For any point q inside a particular sector, $\texttt{pred}(q)$ is the apex of the sector.

Using the shortest path map, we can answer queries about the length of the shortest path from s to a query point in $O(\log n)$ time. We construct the shortest path map for s, then preprocess it for point location. At the same time we compute $d(s, p_i)$ for each polygon vertex p_i. Given a query point q, we locate its containing sector in $O(\log n)$ time. Let a be the apex of the sector. Then $d(q,s) = d(q,a) + d(a,s)$. The second term has already been precomputed, and the first term is just the length of the segment $\overline{qa}$. If we need the full shortest path $\pi(q,s)$, we can extract it from the `pred` pointers in time proportional to the number of vertices along it.