

# Constant-Work-Space Algorithm for a Shortest Path in a Simple Polygon

Tetsuo Asano<sup>1</sup>, Wolfgang Mulzer<sup>2</sup>, Yajun Wang<sup>3</sup>

<sup>1</sup> School of Information Science, JAIST, Japan

<sup>2</sup> Department of Computer Science, Princeton University, USA,

<sup>3</sup> Microsoft Research, Beijing, China.

**Abstract.** We present two space-efficient algorithms. First, we show how to report a simple path between two arbitrary nodes in a given tree. Using a technique called “computing instead of storing”, we can design a naive quadratic-time algorithm for the problem using only constant work space, i.e.,  $O(\log n)$  bits in total for the work space, where  $n$  is the number of nodes in the tree. Then, another technique named “controlled recursion” improves the time bound to  $O(n^{1+\varepsilon})$  for any constant  $\varepsilon > 0$ . Second, we describe how to compute a shortest path between two points in a simple polygon. Although the shortest path problem in general graphs is NL-complete [11], this constrained problem can be solved in quadratic time using only constant work space.

## 1 Introduction

We present two polynomial-time algorithms in a computational model which we call *constant-work-space computation*. In this model, which is also known as “log-space”, the input is given as a read-only array, and the algorithm can access an arbitrary array element in constant time. This differs from the strict data-streaming model where the input can be read only once in a sequential manner. Chan and Chen [7] give algorithms in different computational models varying from a multipass data-streaming model to the random access constant-work-space model in our paper.

One of the most important constant-work-space algorithms is the selection algorithm by Munro and Raman [14] which runs in  $O(n^{1+\varepsilon})$  time using work space  $O(1/\varepsilon)$  for any small constant  $\varepsilon > 0$ . A polynomial-time algorithm for determining connectivity of two arbitrarily specified nodes in a graph by Reingold [15] is another breakthrough in this area. See also [1–4] for applications to image processing. Constant-work-space algorithms for geometric problems are also known: Asano and Rote [5] give efficient algorithms for drawing Delaunay triangulation and Voronoi diagram of a planar point set, and they also show how the Euclidean minimum spanning tree for a planar point set can be constructed quickly in this model.

Here, we focus on the efficiency of algorithms in the constant work space model. Using two geometric problems we showcase some techniques for designing space-efficient algorithms. One technique, named “**computing instead of**

**storing**”, is applied to the problem of finding a simple path between two nodes in a tree. A simple solution in a standard computational model with linear work space goes as follows: compute an Eulerian path between the two nodes and count how often each edge appears on the path. Removing those edges that appear twice gives us the desired simple path. We can implement this idea without using any extra array. Instead of storing a count in each edge, we compute it directly whenever we need to decide whether to include an edge in the path or not. This takes linear time per edge, so in this way we can find a simple path in quadratic time without using any extra array.

Then we describe another technique called “**controlled recursion**”. This technique, which was also used by Munro and Raman [14], limits the recursion depth by a certain value determined by the amount of work space. Using this technique the running time of the algorithm can be improved to  $O(n^{1+\varepsilon})$  for any small positive constant  $\varepsilon$  using work space of size  $O(1/\varepsilon)$ .

The above algorithms can be extended to an algorithm for finding a shortest path between two points in a simple polygon. A naive application leads to a polynomial-time algorithm, but a more careful implementation yields a quadratic-time algorithm.

## 2 Finding a Simple Path on a Tree Using Eulerian Tours

Consider the following question: Let  $T$  be a tree with  $n$  nodes. Given two nodes  $s$  and  $t$ , find a simple path from  $s$  to  $t$  with no node visited more than once. This is our first problem.

Here is a simple naive algorithm. It is well known that any tree has an Eulerian tour visiting every edge exactly twice. Let  $E$  be a subtour that goes from  $s$  to  $t$ . A simple path between  $s$  and  $t$  is obtained by removing duplicate edges out of  $E$ . Thus, if we know how to generate an Eulerian tour, it is easy to reform a part of the tour into a simple path by counting the number of occurrences of each edge. Unfortunately, in our constant work space model no extra array can be used for the counts.

Thus, we apply the technique “**computing instead of storing**” in which whenever we need a value we compute it instead of storing it. Suppose we start from  $s$  with a starting edge  $(s, s')$ . We need to decide whether to include  $(s, s')$  in the path or not. For that, we compute the subtour of the Eulerian tour that starts with  $(s, s')$  and ends at  $t$ , to see whether  $(s, s')$  appears exactly once. If it does, we output the edge and proceed to  $s'$ , otherwise we go to the next edge incident to  $s$ . Note that we do not actually need to find the whole subtour until  $t$ ; rather, we can also stop the search when we hit  $s$  for a second time.

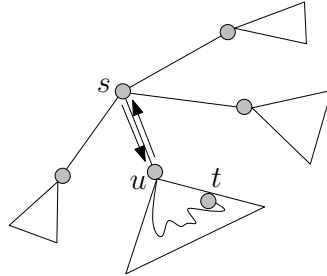
We introduce some terminology for a formal description of the algorithm. We assume that the tree is given by adjacency lists stored in a read-only array. Let  $\text{Adj}(u)$  be the adjacency list of a node  $u \in T$ . The following two functions suffice to generate an Eulerian tour.

**FirstNeighbor**( $u$ ): given a node  $u$ , return the first node in the adjacency list  $\text{Adj}(u)$ .

**NextNeighbor**( $u, v$ ): Given a node  $u$  and an adjacent node  $v$ , return the successor of  $v$  the adjacency list  $\text{Adj}(u)$ . If  $v$  is the last node, return the first node in the list.

The function **FirstNeighbor** can easily be performed in constant time, but the time required for **NextNeighbor** depends on which data structure we assume. If the tree is given by a doubly-connected edge list [6, Chapter 2], then it takes constant time. If a naive data structure is assumed, we may need to search all of  $\text{Adj}(u)$  for the next element, which takes time  $O(\Delta)$ , where  $\Delta$  is the maximum degree of a node in the tree.

Given a tree  $T$ , a starting node  $s$ , a target node  $t$ , and the first edge  $(s, s')$  of an Eulerian tour from  $s$  to  $t$ , **Algorithm 1** finds the shortest path from  $s$  to  $t$ . For this, it repeatedly calls the function **FindFeasibleSubtree** to obtain the next edge on the shortest path by determining the subtree of the current node that contains  $t$ . In **FindFeasibleSubtree**, we use a function **SubtreeSearch** to determine whether a given subtree contains  $t$ : **SubtreeSearch** starts from an edge  $(u, v)$  incident to  $u$  and follows the Eulerian path by applying the function **NextNeighbor**. If we encounter the twin edge  $(v, u)$  before  $t$ , then  $t$  is not contained in the corresponding subtree, i.e., the edge  $(u, v)$  appears twice in the tour (see Figure 1).



**Fig. 1.** Which subtree of  $s$  contains the target node  $t$ ?

**Lemma 1.** For a tree  $T$ , a starting node  $s$ , a starting edge  $(s, s')$  and a target node  $t$ , let  $\ell$  be the length of the shortest path from  $s$  to  $t$  and  $m$  be the length of the Eulerian tour that begins with the edge  $(s, s')$  and ends at  $t$ . Then **Algorithm 1** reports a simple path from  $s$  to  $t$  in  $O(\ell m d)$  time and with  $O(1)$  space. Here,  $d$  depends on which data structure is used: if the tree is given by a doubly-connected edge list then  $d = O(1)$ . If a naive data structure is assumed then  $d = O(\Delta)$ , where  $\Delta$  is the maximum node degree in  $T$ . This running time is always  $O(n^2 d)$ , where  $n$  is the number of nodes in  $T$ .

*Proof.* The algorithm starts from  $s$  and checks for each neighbor of  $s$ , beginning with  $s'$ , whether the corresponding subtree contains  $t$ . Then it proceeds to the

---

**Algorithm 1:** Finding a simple path from  $s$  to  $t$ .

---

**Input:** A tree  $T$ , two nodes  $s$  and  $t$  in  $T$ , and the starting edge  $(s, s')$  for a Eulerian tour from  $s$  to  $t$ .

**Output:** A simple path from  $s$  to  $t$ .

```

begin
  currentNode = s; startNeighbor = s';
  repeat
    report currentNode;
    u = currentNode;
    currentNode = FindFeasibleSubtree(currentNode, startNeighbor, t);
    startNeighbor = NextNeighbor(currentNode, u);
  until currentNode = t
function FindFeasibleSubtree(u, v, t) // returns a child of u whose subtree
contains t
begin
  for each node w in Adj(u), in order starting from v do
    if SubtreeSearch(u, w, t) then return w;
function SubtreeSearch(u, v, t) // checks whether the subtree of u rooted
at v contains t
begin
  currentNode = u; neighbor = v;
  repeat
    nextNode = NextNeighbor(neighbor, currentNode);
    currentNode = neighbor; neighbor = nextNode;
  until (currentNode = t or (currentNode = v and neighbor = u))
  return (currentNode = t);

```

---

appropriate neighbor and continues. Thus, it suffices to show correctness of the function **SubtreeSearch**( $u, v, t$ ) which checks whether the subtree of  $u$  rooted at  $v$  contains  $t$ . This is proved by induction on depth of the subtree.

This algorithm runs in  $O(\ell md)$  time, as **FindFeasibleSubtree** is called at most  $\ell$  times, and each such call takes  $O(md)$  time. Since  $\ell \leq m \leq n$ , it follows that this is always  $O(n^2d)$ .  $\square$

Figure 2 illustrates how the search proceeds.

We have shown that simple paths on a tree can be found in quadratic time if the tree is represented by an appropriate data structure. From now on we assume the doubly-connected edge list representation, so that **NextNeighbor** can be performed in constant time. We can further improve the running time by using “**controlled recursion**,” a technique that also appears in the median-finding algorithm by Munro and Raman [14]. The idea is to use a recursive algorithm such that the depth of the recursion is bounded by a predetermined constant, which reflects the amount of workspace.

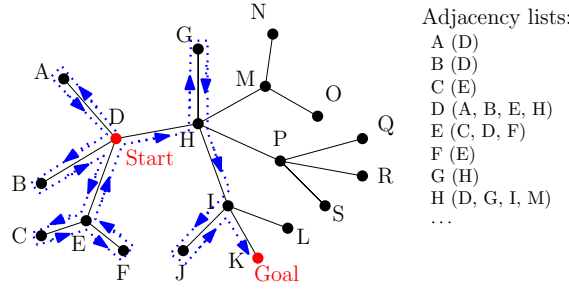


Fig. 2. Canonical traversal of a tree given by adjacency lists.

Suppose  $O(k)$  cells of work space are available. We design algorithms,  $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_k$  such that  $\mathcal{A}_i$  calls  $\mathcal{A}_{i-1}$  for  $i = k, k - 1, \dots, 2$ . For  $\mathcal{A}_1$ , we just use Algorithm 1 from above.

Now we describe the algorithm  $\mathcal{A}_k$  (see Algorithm 2). Let  $s = u_1, u_2, \dots, u_m = t$  be the Eulerian tour from  $s$  to  $t$ , with starting edge  $(s, s')$ . It is a sequence of nodes in which a node may appear repeatedly. Let  $z = m^{(k-1)/k}$ , and let  $B = \langle s = u_1, u_2, \dots, u_z \rangle$  denote the first  $z$  nodes in this sequence. Algorithm 2 finds the lowest common ancestor (lca) of  $u_z$  and  $t$  in  $B$  using binary search. This is done as follows: We maintain a subinterval  $(u_a, u_b)$  of the path  $B$ , and call  $\mathcal{A}_{k-1}$  to determine the shortest path  $P_{ab}$  between  $u_a$  and  $u_b$ . We locate the nodes of  $P_{ab}$  on  $B$ , until we reach the point where a node on  $P_{ab}$  first appears in the second half of the interval  $(u_a, u_b)$ . We then determine whether we have already found the lca, or in which direction we need to shorten the search interval. To perform this test for a node  $v$ , we examine the subtrees rooted at  $v$ : (i) if  $s, u_z$ , and  $t$  are contained in different subtrees of  $v$ , then  $v$  is the lca of  $u_z$  and  $t$ ; (ii) if  $s$  and  $t$  are contained in the same subtree, then the lca of  $u_z$  and  $t$  comes before  $v$  on  $P_{ab}$ ; and (iii) if  $u_z$  and  $t$  are in the same subtree, then the lca of  $u_z$  and  $t$  comes after  $v$  on  $P_{ab}$ . These cases can be distinguished in  $O(m)$  time.

After the lca  $u_a$  has been determined, we call  $\mathcal{A}_{k-1}$  to output the shortest path from  $s$  to  $u_a$ . Then we repeat the same procedure as above, with  $u_a$  instead of  $s$  (and using the Eulerian path from  $u_a$  to  $t$  which starts with the edge that follows in  $u_a$ 's adjacency list after its last adjacent node on  $B$ ). We do this until we reach  $t$ .

**Lemma 2.** *There exists a constant  $\gamma$  such that the following holds: Let  $k \geq 2$  and suppose Algorithm  $\mathcal{A}_{k-1}$  on input  $(s, s')$  and  $t$  finds the shortest path between  $s$  and  $t$  with storage  $O(k - 1)$  and in time at most  $\gamma'(2^{k-1} - 1)m^{1+1/(k-1)} \log m$ , where  $m$  is the length of the Eulerian tour from  $(s, s')$  to  $t$  and  $\gamma' \geq \gamma$ . Then, given  $(s, s')$  and  $t$ , Algorithm  $\mathcal{A}_k$  finds the shortest path between  $s$  and  $t$  with storage  $O(k)$  and in time at most  $\gamma'(2^k - 1)m^{1+1/k} \log m$ , where again  $m$  is the length of the Eulerian tour from  $(s, s')$  to  $t$ .*

*Proof.* Let  $c = b - a$ . Then one iteration of the binary search needs at most  $\gamma'(2^{k-1} - 1)c^{1+1/(k-1)} \log m + O(m)$  steps, where the first summand represents

**Algorithm 2:** The algorithm  $\mathcal{A}_k$ .

---

**Input:** A tree  $T$ , two nodes  $s, t \in T$ , and a starting edge  $(s, s')$ .  
**Output:** A simple path from  $s$  to  $t$ .

**begin**

- $z = m^{(k-1)/k}$ , where  $m$  is the length of the Eulerian tour  $E$  from  $(s, s')$  to  $t$ ;
- while**  $s$  is more than  $z$  steps away from  $t$  on  $E$  **do**
  - $(a, b) = (1, z)$ ; // denote by  $u_1, \dots, u_z$  the first  $z$  nodes on the tour from  $(s, s')$  to  $t$ .
  - repeat**
    - $\alpha = \lfloor (a + b)/2 \rfloor$ ;  $(u_r, u_{r+1}) = (u_a, u_{a+1})$ ;
    - Call  $\mathcal{A}_{k-1}$  with starting edge  $(u_a, u_{a+1})$  and target  $u_b$ ;
    - repeat**
      - Every time  $\mathcal{A}_{k-1}$  outputs a node  $w$ , follow  $E$  from  $(u_r, u_{r+1})$  to the first appearance  $u_o$  of  $w$ ;
      - if**  $o < \alpha$  and  $w \neq u_z$  **then**
        - $(u_r, u_{r+1}) = (u_o, u_{o+1})$ ;
    - until**  $o \geq \alpha$  or  $w = u_z$
    - if**  $u_o$  is the lca of  $t$  and  $u_z$  **then**
      - $a = b = o$ ;
    - else if**  $u_r$  is the lca of  $t$  and  $u_z$  **then**
      - $a = b = r$ ;
    - else if**  $u_o$  is before the lca of  $t$  and  $u_z$  **then**
      - $a = o$ ;
    - else**
      - $b = r$ ; //  $u_r$  is after the lca of  $t$  and  $u_z$
  - until**  $a = b$
  - Call  $\mathcal{A}_{k-1}$  with starting edge  $(s, s')$  and target  $u_a$ , and output the resulting nodes;
  - $(s's) = (u_r, u_{r+1}) = (u_a, w)$ , where  $w$  is the first neighbor of  $u_a$  on the Eulerian tour after  $u_z$ ;
- Call  $\mathcal{A}_{k-1}$  with starting edge  $(s, s')$  and target  $t$ , and output the resulting nodes;

---

the recursive call and the second summand accounts for locating  $u_o$  and the lca-test. Since  $c$  is at least halved in each iteration, the binary search needs total time at most

$$\gamma'(2^{k-1} - 1)z^{1+1/(k-1)} \log m + O(m \log m) = \gamma'(2^{k-1} - 1)m \log m + O(m \log m),$$

by the definition of  $z$ . Choosing  $\gamma$  such that the second summand is at most  $\gamma m \log m$ , and adding the time for calling  $\mathcal{A}_{k-1}$  after the lca has been determined, it follows that one iteration of the outer loop takes time at most  $\gamma'(2^k - 1)m \log m$ . Now, since the Euler tour from  $(s, s')$  to  $t$  is subdivided into at most  $m^{1/k}$  segments by the outer loop, the total running time is  $\gamma'(2^k - 1)m^{1+1/k} \log m$ , as claimed. The bound on the storage space is immediate.  $\square$

We immediately get the following theorem.

**Theorem 1.** *Given two nodes  $s$  and  $t$  in a tree  $T$  with  $n$  nodes in a read-only storage and any positive constant  $\varepsilon$ , there is an algorithm which finds the simple path from  $s$  to  $t$  in  $T$  in  $O(n^{1+\varepsilon})$  time using only  $O(1/\varepsilon)$  cells of work space.*

*Proof.* Set  $k = 2/\varepsilon$ . By Lemma 1, Lemma 2 and induction,  $\mathcal{A}_k$  runs in time  $O((2^{2/\varepsilon} - 1)n^{1+\varepsilon/2} \log n) = O(n^{1+\varepsilon})$ , as desired.  $\square$

In Theorem 1 we assumed a doubly-connected edge list for the input tree. If the tree is given in a simple list, then the basic operation to find the next or previous edge takes time proportional to the length of the adjacency list. Thus, the time complexity becomes  $O(n^{1+\delta} \Delta)$ , where  $\Delta$  is the maximum node degree in  $T$ .

### 3 Shortest Paths in Polygons

Dijkstra’s algorithm for finding a shortest path between two specified vertices in a weighted graph is one of the most popular and important algorithms [10, Chapter 24]. It can find such a path in  $O(n^2)$  time using a simple data structure that maintains the current distance from the source vertex to each other vertex during the progress of the algorithm. Is it still possible to find such a shortest path in the constant-work-space model where the input graph is given by a read-only array and only constantly many storage cells with  $O(\log n)$  bits are available as work space? Unfortunately, no polynomial-time algorithm for the shortest path problem is known in this model. Actually, the problem is NL-complete [11]. In this paper we consider the following restricted version:

**Geodesic Shortest Path within a Simple Polygon:**

Given a simple polygon  $P$  with  $n$  vertices and two points  $s$  and  $t$  in its interior, find the shortest path between  $s$  and  $t$  within the polygon  $P$ .

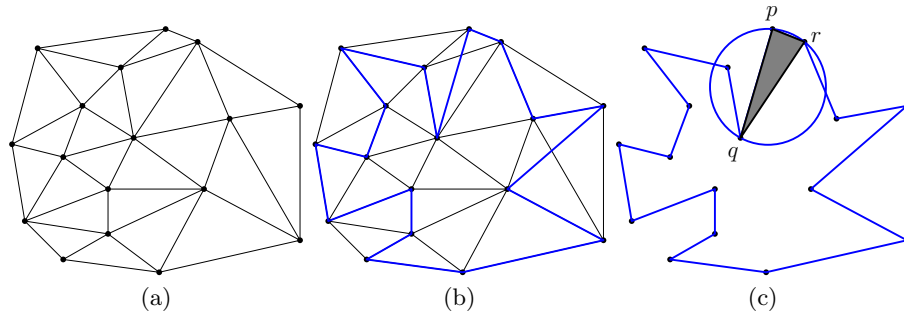
There is a linear-time algorithm if  $O(n)$  work space is allowed [13]. It works as follows: Given a simple polygon  $P$ , we first partition its interior into triangles using Chazelle’s linear-time algorithm [8]. Then, we compute the dual graph  $G^*$  of the triangulation: the vertices of  $G^*$  correspond to the triangles, and two vertices are adjacent if their corresponding triangles share an edge. Since  $G^*$  is a tree, any two vertices in  $G^*$  are connected by a unique simple path. Given two points  $s$  and  $t$  to be interconnected, we locate them in the triangulation and thus in  $G^*$ . Consider the unique path in  $G^*$  from the triangle containing  $s$  to the triangle containing  $t$ . It defines a sequence  $(e_0, e_1, \dots, e_m)$  of triangular edges hit by the path. We walk along this sequence while maintaining a *funnel*. The funnel consists of a *cusp*  $p$ , initially set to  $s$ , and two concave chains from  $p$  to the two end vertices of the current edge  $e_i$ . In each step, there are two cases: (i) if the next edge remains visible from the cusp, we just update the appropriate concave chain, similar to Graham’s scan; (ii) if the next edge is not visible from the cusp, we proceed along the appropriate chain until we find the cusp for the next funnel, and output the vertices encountered along the way as part of the shortest path. Properly implemented, all this takes  $O(n)$  time (see Lee and Preparata [13] for details).

### 3.1 A Shortest-Path Algorithm Using the Dual Graph

We adapt the algorithm by Lee and Preparata [13] to use constant work space. For this, we need to solve two problems: (i) we need to develop an algorithm for triangulating a given simple polygon and then finding a simple path in the dual graph; and (ii) we must maintain the funnel during the traversal. The difficulty here is, of course, that we cannot store any intermediate results.

*Computing the triangulation.* In order to maintain the triangulation of our polygon efficiently, we will use a canonical triangulation of a simple polygon as well as a canonical traversal of the tree. Specifically, we will take the *constrained Delaunay triangulation* [9]. For a point set  $S$ , three points of  $S$  determine a *Delaunay triangle* if and only if the circle defined by the three points contains no point of  $S$  in its proper interior. Such a circle is called an *empty circle*. Delaunay triangles partition the convex hull of the set  $S$ , and the resulting structure is called the *Delaunay triangulation* of  $S$  [6, Chapter 9].

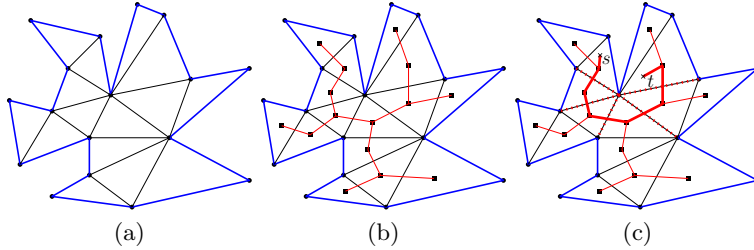
Now we describe how to extend this notion to a simple polygon  $P$  [9]. The vertices of  $P$  define a point set  $V$  and the edges define a set  $E$  of line segments. Constrained Delaunay edges are defined using the notion of a *chord*. A chord is an open line segment between two polygon vertices that does not intersect the boundary of  $P$ . A pair  $(p, q)$  of vertices defines a constrained Delaunay edge if and only if there is a third point  $r$  in  $V$  such that (i)  $(p, q)$  is a chord; (ii)  $(p, r)$  and  $(q, r)$  are chords or polygon edges; and (iii) the circle through  $p, q, r$  does not contain any other point  $s \in V$  that is visible from  $r$ , i.e., for no other point  $s$  in the circle is  $(r, s)$  a chord.



**Fig. 3.** An example of a constrained Delaunay triangle. (a) The Delaunay triangulation of a planar point set; (b) the overlay of a polygon with the Delaunay triangulation of its vertex set; and (c) a Delaunay triangle  $(p, q, r)$  and its associated empty circle. Note that the vertex in the circle is not visible from  $r$ .

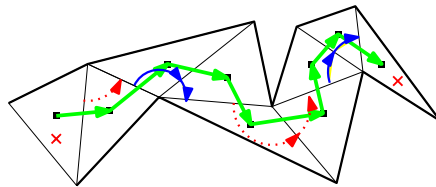
It is known that the constrained Delaunay triangulation  $DT(P)$  is uniquely determined for any simple polygon whose vertices are in general position. Once we have a  $DT(P)$ , we define its dual graph  $DT(P)^*$ : the vertices are triangles,

and two vertices are adjacent if and only if their corresponding triangles share an edge. Since a simple polygon is simply connected,  $DT(P)^*$  is always a tree.



**Fig. 4.** The unique path on the dual graph and its corresponding sequence of Delaunay edges. (a) Constrained Delaunay triangulation of a given simple polygon, (b) the dual graph of the triangulation (a tree), and (c) the unique path between  $s$  and  $t$ .

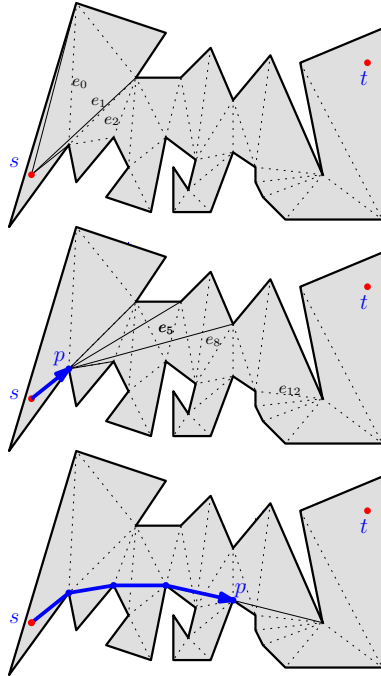
We would like to use Theorem 1 to find the shortest path from  $s$  to  $t$  in  $DT(P)^*$ . For this, we need to implement the function `NextNeighbor()`. By the definition of the dual graph and the fact that each  $DT(P)^*$  has maximum degree at most 3, the next neighbor is determined by the clockwise or counterclockwise next Delaunay edge, as shown in Figure 5. Hence, we need to find the third vertex of a Delaunay triangle for a given edge. More precisely, given a Delaunay edge  $(u, v)$ , we want to find a vertex  $w$  such that (i)  $w$  is visible from the edge  $(u, v)$ ; and (ii) the circle defined by  $u, v$ , and  $w$  is empty, that is, it does not contain any other vertex visible from the edge  $(u, v)$ . Thus, it takes  $O(n^2)$  time to find a vertex with which a Delaunay edge forms a Delaunay triangle, and this is also the time for an invocation of `NextNeighbor()`.



**Fig. 5.** Walking along a path in the dual graph while finding the clockwise next Delaunay edge (solid or blue arrow) or counterclockwise next edge (dotted or red arrow).

*Maintaining the funnel.* While walking along the path from  $s$  to  $t$  in  $DT(P)^*$ , we need to maintain the current funnel. Unfortunately, we do not have space to store the two concave chains. Therefore, we proceed as follows: while walking, we only maintain the cusp  $p$  of the funnel, and the two vertices  $a, b$  that determine

the visibility angle from  $p$  ( $a, b$  are the first vertices on the two concave chains). We initialize  $p$  to the starting point  $s$  and  $a, b$  to the endpoints of the first edge of the path  $e_0$ . In order to process a new edge  $e_i$ , we take the intersection of the current visibility angle with that defined by  $e_i$ . If the new visibility angle is nonempty, we do nothing. Otherwise, we must update the cusp of the current funnel. For this, we start from  $a$  or  $b$ , depending on whether  $e_i$  lies above or below the old visibility angle. Then, we perform a Jarvis march along the boundary of the polygon, outputting the vertices of the concave chain, until we find a vertex from which  $e_i$  is visible again. This vertex becomes our new cusp  $p$ , and  $a, b$  are set to the highest and lowest point on  $e_i$  that is visible from  $p$ ; see Figure 6 for an example. Since every step in Jarvis's march needs  $O(n)$  time, and since



**Fig. 6.** Maintaining the funnel. Initially, the cusp of the funnel is set to  $s$ , and  $a, b$  are the endpoints of the edge  $e_0$ . The visibility angle needs to be updated on encountering  $e_1$  and  $e_2$  (top). At  $e_2$ , the visibility angle vanishes, and we must advance the cusp. Now, the visibility angle is updated at  $e_5$  and  $e_8$ . At  $e_{12}$ , the visibility angle becomes empty again, and we need to perform a Jarvis march on the lower boundary of the polygon until we find the next cusp.

there are  $O(n)$  such steps overall, the total overhead for maintaining the funnel is  $O(n^2)$ . We have thus shown:

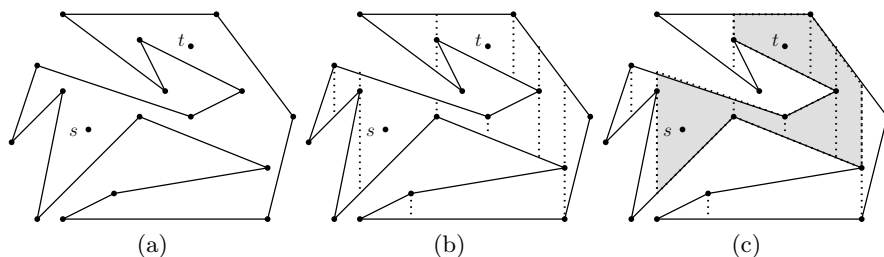
**Theorem 2.** *There is a constant-work space algorithm for finding a shortest path between arbitrary two points in a simple  $n$ -gon  $P$  in time  $O(n^{3+\varepsilon})$  for any small constant  $\varepsilon > 0$ .*

*Proof.* By Theorem 1, the shortest path in  $DT(P)^*$  can be found by  $O(n^{1+\varepsilon})$  applications of the function **NextNeighbor**( $\cdot$ ). Since each such call takes  $O(n^2)$  time, and since the total overhead for the funnel maintenance is  $O(n^2)$ , we obtain the bound in the theorem.  $\square$

### 3.2 A Shortest-Path Algorithm Using Point Location

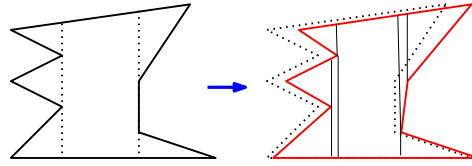
The algorithm from Theorem 2 comes from a direct adaptation of the algorithm by Lee and Preparata [13]. The dual graph, which is a tree, gives us the correct direction toward a given target point. In this section we show that there is a more direct way to find this direction. Suppose we are in some triangle  $A$  in  $DT(P)$ . Removing  $A$  divides  $P$  into at most three parts. We need to find the part that contains  $t$ . For this, we find an edge  $e_t$  just above the target point  $t$ , which is the first polygon edge hit by the vertical ray emanating upward from  $t$ . Then, the part containing  $t$  is the one whose boundary contains the edge  $e_t$ , which is decided using edge indices, assuming that edges are sequentially numbered along the boundary. This kind of operation is called *point location* in computational geometry. It takes just  $O(1)$  time.

Now, we know which way to go from any triangle. Unfortunately, finding adjacent triangles in a canonical triangulation can be slow. The next idea for efficiency is to use the *trapezoidal decomposition* instead of the constrained Delaunay triangulation [6, Chapter 6]. That is, we partition the interior of a given simple polygon by drawing a vertical chord at each vertex toward the interior of the polygon. This decomposition is canonical and easy to compute. Moreover, it inherits the same properties that the triangulation used to have for shortest paths.



**Fig. 7.** Trapezoidal decomposition of a simple polygon for finding a shortest path in a simple polygon. (a) A simple polygon and two internal points  $s$  and  $t$  to be interconnected within the polygon. (b) Trapezoidal decomposition of  $P$ . (c) A sequence of trapezoids between two containing  $s$  and  $t$ .

The trapezoidal decomposition defined above is uniquely determined for any simple polygon. In general, degeneracies can cause one trapezoid to be adjacent to arbitrarily many trapezoids, as shown in Figure 8. Hence, we perform a symbolic perturbation to avoid this issue: each vertex of  $P$  and the two points  $s$  and  $t$  all have integral coordinates with  $O(\log n)$  bits. Then, each integral point  $(x, y)$  is treated as a point  $(x + y\varepsilon, y)$ , ie, it is shifted to the right by  $y\varepsilon$  for a small parameter  $\varepsilon$  such that  $y^*\varepsilon < 1$  for the largest  $y$ -coordinate  $y^*$  of a vertex. After this perturbation, no two vertices share the same  $x$ -coordinate, as shown to the right in Figure 8.



**Fig. 8.** Removing degeneracies by shifting vertices to the right. An original polygon is given to the left. The conversion results in the right polygon in which no two vertices share the same  $x$ -coordinate.

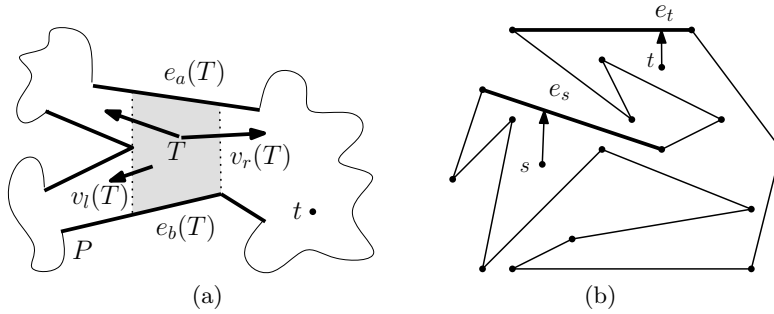
From now on, we assume that no two vertices have the same  $x$ -coordinate. This implies that any trapezoid is adjacent to at most four other trapezoids. Our first goal is to find a sequence of trapezoids between those containing  $s$  and  $t$ . For this, it suffices to find the correct neighbor at each trapezoid. More formally, suppose we are in a trapezoid  $T$  of which we know that it appears in the sequence. Since  $T$  is adjacent to at most four trapezoids, we want to determine which one lies on the correct path.

A characterization of a trapezoid is given in Figure 9. For a trapezoid  $T$ , two polygon edges  $e_a(T)$  and  $e_b(T)$  bound  $T$  from above and below, respectively. The vertical sides of  $T$  are denoted by  $v_l(T)$  and  $v_r(T)$ , the left and right sides, respectively. At a trapezoid  $T$  we have to determine which way we should go toward the target point  $t$ . To that end, we traverse the corresponding boundary to find which part contains the edge  $e_t$  just above  $t$ .

By the observation above we know that we can find the correct next trapezoid toward the target  $t$  in  $O(n)$  time without using any extra array. Since the length of the trapezoid sequence is  $O(n)$ , the total time we need to find the sequence is  $O(n^2)$ .

We still need to describe how to find the shortest path from  $s$  to  $t$ , but this just works as in the previous algorithm: we know how to walk on the sequence using  $O(n)$  time at each step. To find a shortest path we just maintain the funnel from the current starting point.

Given an arbitrary point  $q$  in the interior of  $P$ , we can determine a trapezoid containing  $q$  as follows: first find the polygon edges which are hit by a vertical ray emanating from  $q$  upward. The one closest to  $q$  is the top edge  $e_a(T)$  of the



**Fig. 9.** Characterization of a trapezoid  $T$  by two polygon edges bounded from above and below and two vertical sides. (a) A trapezoid adjacent to three trapezoids. (b) A polygon edge  $e_s$  just above the point  $s$  and a polygon edge  $e_t$  just above  $t$ .

trapezoid  $T$  containing  $q$ . In a similar fashion we can find the polygon edge  $e_b(T)$  just below  $q$ , which is the bottom edge of  $T$ . Then, we compute the left and right vertical sides of the trapezoid  $T$ , denoted by  $v_l(T)$  and  $v_r(T)$ , respectively. We start with four endpoints of  $e_a(T)$  and  $e_b(T)$ .  $v_l(T)$  is initially determined by the rightmost of the two left endpoints of  $e_a(T)$  and  $e_b(T)$ . The initial value of  $v_r(T)$  is similarly determined. Then, we scan each polygon vertex. If it lies inside the current trapezoid and its incident polygon edge enters the trapezoid from its left, then we update the value  $v_l(T)$  to be the  $x$ -coordinate of the vertex. If it lies in  $T$  and its incident edge enters  $T$  from the right, we update  $v_r(T)$ . In this way we can obtain the trapezoid in  $O(n)$  time.

Finally, how can we find trapezoids adjacent to a given trapezoid? Suppose we want to find a trapezoid  $T_r$  which shares a right boundary with  $T$ . To do this, take a point  $q$  which is located to the right of the side at a small enough distance. Using the point  $q$ , the trapezoid  $T_r$  is computed in the same manner as described above. Thus, once we have a trapezoid, we can find its neighbors in  $O(n)$  time.

**Theorem 3.** *Given an  $n$ -gon  $P$  and two arbitrary points in  $P$ , we can find a shortest path between them within  $P$  in  $O(n^2)$  time in the constant work space model.*

We have implemented our algorithm using LEDA [12] to guarantee the detail. An experimental result is shown in Figure 10. An input configuration is shown in (a). After finding the initial segment on the shortest path in (b), we repeatedly generate next trapezoids while the visibility angle vanishes. When we reach the rightmost trapezoid in (c), the visibility angle vanishes and the shortest path is extended as shown in (c). The final result is shown in (d). The shortest path is given by bold lines in the figure.

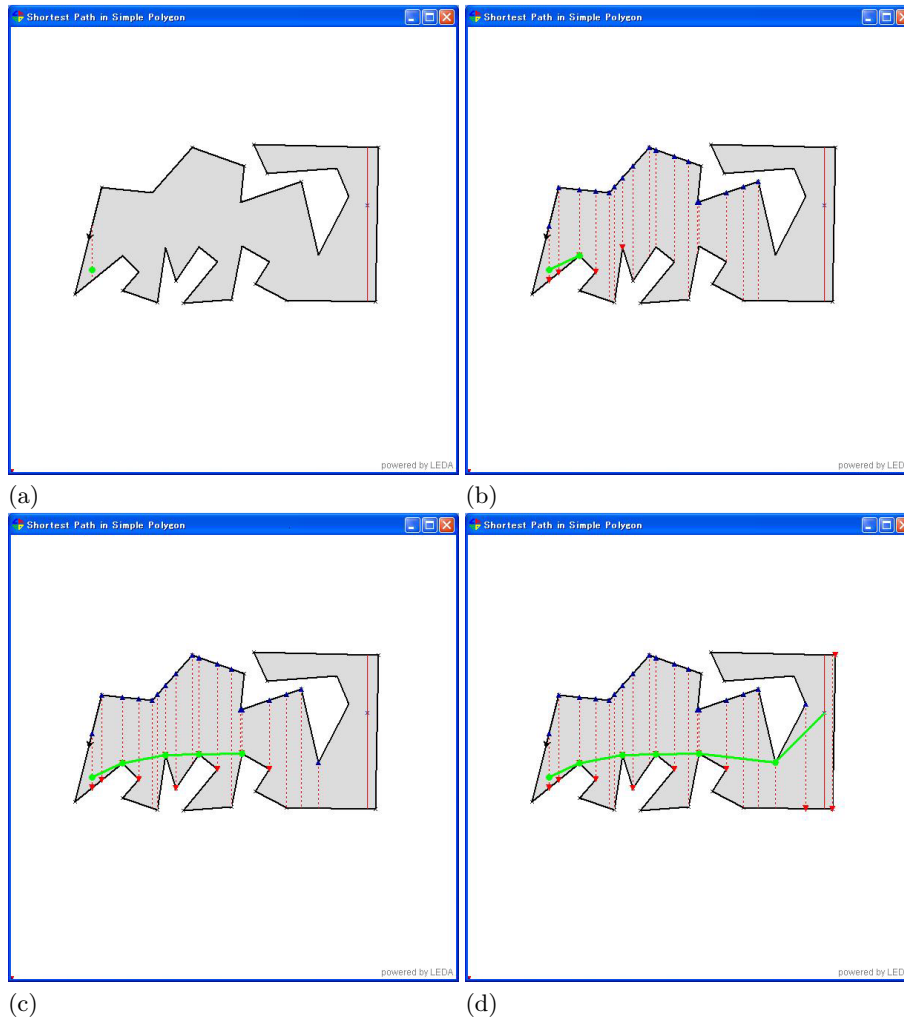


Fig. 10. Experimental results.

## 4 Concluding Remarks

We have presented constant-work-space algorithms for finding a shortest path between two arbitrary points in a tree and in a simple polygon. One obvious future direction is to extend the algorithm to the general Euclidean shortest path problem in the presence of polygonal obstacles. If the number of polygonal obstacles is bounded by some small constant  $k$ , then there are  $O(n^k)$  different ways to convert the problem into ones on a simple polygon by connecting obstacles by chords, so we can obtain a polynomial-time algorithm. However, finding such an algorithm for an unbounded number of obstacles, or proving that no polynomial-time constant work-space algorithm exists, seems more challenging.

Another interesting generalization is to consider the shortest path problem on weighted graphs. No upper bound is known for the problem. A number of geometric problems are open in the constant work space model. For example, does there exist an efficient constant-working-space algorithm for computing the visibility polygon from a point in a simple polygon. Another interesting direction is to investigate time-space trade-offs: how much work space is need to find a shortest path in a simple polygon in linear time?

## Acknowledgments

This work of T.A. was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas and Scientific Research (B).

## References

1. T. Asano. Constant-work-space algorithms: how fast can we solve problems without using any extra array? In *Proc. 19th Annu. Internat. Sympos. Algorithms Comput. (ISAAC), invited talk*, volume 5369 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 2008.
2. T. Asano. Constant-working space algorithm for image processing. In *Proc. of the First AAAC Annual meeting*, page 3, Hong Kong, 2008. April 26–27.
3. T. Asano. Constant-work-space algorithms for image processing. In F. Nielsen, editor, *Emerging Trends in Visual Computing (ETVC 2008)*, volume 5416 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2009.
4. T. Asano. Constant-working-space image scan with a given angle. In *Proc. 24th European Workshop Comput. Geom. (EWCG)*, pages 165–168, Nancy, France, 2009. March 18–20.
5. T. Asano and G. Rote. Constant-working-space algorithms for geometric problems. In *Proc. 21st Canad. Conf. Comput. Geom. (CCCG)*, pages 87–90, Vancouver, 2009.
6. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, third edition, 2008.
7. T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.

8. B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.
9. L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.
10. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.
11. A. Jakoby and T. Tantau. Logspace algorithms for computing shortest and longest paths in series-parallel graphs. Technical Report TR03-077, ECCO Reports, 2003.
12. LEDA. Library for efficient data types and algorithms. *Algorithmic Solutions GmbH*, Germany.
13. D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.
14. J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165(2):311–323, 1996.
15. O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):Art. #17, 24 pp., 2008.