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

Tetsuo Asano[1], Wolfgang Mulzer[2], Yajun Wang[3]

[1] School of Information Science, JAIST, Japan
[2] Department of Computer Science, Princeton University, USA,
[3] Microsft 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 "controlled recursion" improves the time bound to $O(n^{1+\varepsilon})$ for any positive constant ε. Second, we describe how to compute a shortest path between two points in a simple n-gon. Although the shortest path problem in general graphs is NL-complete, 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*, which is also known as "log-space" algorithms. In this model, the input is given as a read-only array, and the algorithm can access an arbitrary array element in constant time, This is a difference 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 multi-pass data-streaming model to the random access constant-work-space model in our paper.

One of the most important constant-work-space algorithms is a selection algorithm by Munro and Raman [9] 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 [10] is also another breakthrough in this area. See also [2–5] for applications to image processing. Constant-work-space algorithms for geometric problems are also known. Asano and Rote [1] 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 more than once gives us a desired simple path. We can implement this idea without using any extra array. Instead of storing a count in each edge, we compute it whenever it is needed, which takes linear time. In this way we can compute a simple path in quadratic time without using any extra array.

Then we describe another technique named "**controlled recursion**" which limits the depth of recursion by a predetermined value determined by the amount of work space. Using this technique the running time of the algorithm is improved into $O(n^{1+\varepsilon})$ for any small positive constant ε 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

As a warm up, consider a simple problem: Let T be a tree with n nodes. Given two nodes s and t, find a simple path with no node visited more than once from s to t. 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 A be such a tour. A simple path between s and t is obtained by considering the portion of A between s and t and removing redundant edges where an (undirected) edge is redundant if it appears twice on A. 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. Whenever we extend a path by an edge e to generate an Eulerian path between two nodes, we generate the Eulerian path to count how often the edge e appears. If it appears exactly once, we report the edge.

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

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

NextNeighbor(u, v)**:** Given a node u and one of its adjacent nodes, say v, return the next node in its adjacency list $\mathrm{Adj}(u)$. 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], then it takes constant time. If a naive data structure is assumed, we may need to search all of Adj(u) for the next element, which takes time $O(\Delta)$ where Δ is the maximum degree of a node in the tree.

Given a tree T, a starting node s, and a target node t, the following function **FindFeasibleSubtree** finds the subtree of s which contains t, in other words, it tells us which edge to follow toward the target t. In the algorithm we successively find the next edge following an Eulerian path starting from s. We start from an edge (s, u) incident to s and follow an Eulerian path by applying the function **NextNeighbor**. If we come back to its twin edge (u, s) before finding the target node t, it means the subtree of s rooted at u does not contain the target node t.

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

```
Input: A tree T and two nodes s and t in T.
Output: A simple path from s to t.
begin
    currentNode = s;
    repeat
        report currentNode;
        currentNode = FindFeasibleSubtree(currentNode);
    until currentNode = t
end
function FindFeasibleSubtree(u, t)   // returns a child of u whose subtree contains t
begin
    for each node v in Adj(u) do
        if SubtreeSearch(u, v, t) then return v;
end
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);
end
```

Lemma 1. *Given a tree T, a starting node s, and a target node t, **Algorithm 1** reports a simple path from s to t in $O(n^2 d)$ time using only constant work space, where n is the number of nodes of T and 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 Δ is the maximum node degree in T.*

Figure 1 illustrates how the search proceeds.

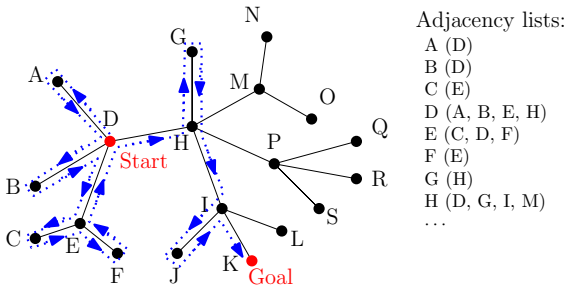

Fig. 1. Canonical traversal of a tree given by adjacency lists.

We have shown that a simple path on a tree can be found in quadratic time if a tree is represented by an appropriate data structure. Further improvement on its running time is possible. A key idea is a "**controlled recursion**," which was also used by Munro and Raman [9] for finding a median . The controlled recursion is an algorithmic technique which controls the recursion depth so that the depth never exceeds a predetermined constant, which reflects the amount of work space.

Suppose $O(k)$ work space is available. Then, we design algorithms, $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_k$ such that $\mathcal{A}_i$ calls $\mathcal{A}_{i-1}$ for $i = k, k-1, \ldots, 2$. As describe above, the algorithm $\mathcal{A}_1$, finds a simple path between two nodes in a tree by removing redundant edges in quadratic time.

The algorithm $\mathcal{A}_2$ uses a decomposition of the Eulerian path from s to t. Let $s = u_0, u_1, \ldots, u_m = t$ be the Eulerian path from s to t, which is a sequence of nodes in which a node may appear more than once. We decompose the path into $O(\sqrt{m})$ blocks, $B_1, B_2, \ldots, B_{\sqrt{m}}$. For the first block $B_1 = \{s = u_0, u_1, \ldots, u_{\sqrt{m}} = v\}$ we find the lowest common ancestor of v and t in B_1 using a binary search. The binary search starts at a node x which is the $\sqrt{m}/2$-th step from u_0 toward v. Now we have three nodes x, v, and t. For each of them we compute its first occurrence and the last occurrence, denoted by $F(x), L(x), F(v), L(v), F(t)$ and $L(t)$, respectively. As is easily seen, the node x is a common ancestor of v and t if and only if $F(x) < F(v) \leq L(v) \leq F(t) \leq L(t) < L(x)$ holds. In the binary search, if $F(v) < F(x)$, that is, if we visit the node v before x when we walk along the Eulerian path from s to t, then we have to walk back to satisfy $F(x) < F(v)$. If $F(x) < F(v)$ but $L(x) < L(t)$, then we have to walk toward t to satisfy $L(t) < L(x)$. In each test we halve the number of steps starting from $\sqrt{m}/2$. Whenever we find a common ancestor of v and t, we compare it with the current lowest common ancestor of v and t. It is easy since it suffices to compare the number of steps from s. The node of the longer steps is lower than the other. Then, we walk toward t to find a possible lower common ancestor.

In this way we can find the lowest common ancestor u of v and t. Each step of the binary search is done in $O(n)$ time for the Eulerian tour from s to s. Thus, it is done in $O(n \log n)$ time in total.

Finally, we compute a simple path from s to u, which is the initial part of the simple path from s to t. The simple path is computed by removing redundant edges from the Eulerian path from s to u. Here note that its length is at most $\sqrt{m}$. Thus, if we use the function $\mathcal{A}_1$, it returns the simple path in $O(\sqrt{m}^2) = O(m) = O(n)$ time.

In the next block B_2 we can start from the lowest common ancestor we just computed. In the same way we can extend the simple path toward t. In this way we extend the simple path $O(\sqrt{m})$ times. Since each iteration is done in $O(n \log n)$ time, the total time we need is $O(n^{3/2} \log n)$.

The algorithm $\mathcal{A}_3$ partitions the Eulerian path from s to t into $O(n^{1/3})$ blocks and finds the lowest common ancestor by the binary search, and finally returns the Eulerian path from the starting node to the lowest common ancestor by using the algorithm $\mathcal{A}_2$. The first part is done in $O(n^{4/3} \log n)$ time and the second part is done in $O((n^{2/3})^{3/2} \log n^{2/3} n^{1/3}) = O(n^{4/3} \log n)$ time in total. Thus, it runs in $O(n^{4/3} \log n)$ time.

The algorithm $\mathcal{A}_k$ partitions the Eulerian path from s to t into $n^{1/(k+1)}$ parts of length $O(n^{k/(k+1)})$. It runs in $O(n^{1+1/(k+1)} \log^{k-1} n)$.

Lemma 2. *Algorithm $\mathcal{A}_k$ finds the simple path between any two nodes in a tree of size n stored in a read-only storage by doubly-connected edge lists in $O(n^{1+1/(k+1)} \log^k n)$ time using $O(k)$ work storage.*

Proof. The lemma is easily proved using induction on k. Note that the amount of work space is now $O(k)$ instead of constant. It is because Algorithm $\mathcal{A}_k$ successively calls algorithms $\mathcal{A}_{k-1}$, $\mathcal{A}_{k-2}$, $\mathcal{A}_1$ in order. □

Now, if we set

$$\varepsilon = \frac{1}{k+1}, \tag{1}$$

then the time complexity of Algorithm $\mathcal{A}_k$ is $O(n^{1+\varepsilon} \log^{1/\varepsilon} n / \log n)$. Suppose we have

$$n^{\varepsilon} = \log^{1/\varepsilon} n. \tag{2}$$

Then, we have

$$\varepsilon = \sqrt{\frac{\log \log n}{\log n}}.$$

Now, the time complexity of Algorithm $\mathcal{A}_k$ is

$$O(n^{1+2\varepsilon}). \tag{3}$$

This means 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 δ, there is an algorithm which finds the simple path from s to t in T in $O(n^{1+\delta})$ time using only $O(1/\delta)$ amount of work space.*

In the theorem we assumed a doubly-connected edge list for a given 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 $O(n^{1+\delta}\triangle)$, where $\triangle$ 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. It can find such an path in $O(n^2)$ time using a simple data structure that maintains the current distance from a source vertex to each 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 a constant number of storage cells of length $O(\log n)$ is available as work space? Unfortunately, no polynomial-time algorithm for the shortest path problem is known in the model. In this paper we consider a restricted version of the problem:

Geometric Shortest Path within a Simple Polygon:
Given a simple polygon P with n vertices and two points s and t in the interior of P, find the shortest path between s and t within the polygon P.

A linear-time algorithm is known for the problem if $O(n)$ work space is allowed. 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 a triangular 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^* between the triangle containing s to the triangle containing t. It defines a sequence of triangular edges hit by the path. Let $(e_0, e_1, \ldots, e_m)$ be this edge sequence. We walk along the sequence while keeping the visibility angle from the starting point s and two vertices v_{low} and v_{high} that determine the visibility.

Whenever the visibility angle vanishes at a triangular edge, we choose a vertex v, either v_{low} or v_{high}, depending on which direction the path bends, and output the edge (s, v) as a part of the shortest path and repeat the same operation after replacing s with v. Obviously, every step is done in constant time. Thus, the algorithm runs in $O(n)$ time.

3.1 A Shortest-Path Algorithm Using a Dual Graph

We adapt the algorithm to use constant work space. That is, we develop an algorithm for triangulating a given simple polygon and then finding a unique path in

the dual graph. The difficulty here is, of course, that we cannot store any intermediate result. To overcome the difficulty, we will use a canonical triangulation of a simple polygon and also a canonical traversal of the tree.

Our canonical triangulation is the *constrained Delaunay triangulation*. 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 passing through three points of S 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.

Now we describe how to extend this notion to a simple polygon P. The vertices of P define a set V of points 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 any edge in E. 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.

It is known that a constrained Delaunay triangulation $\mathrm{DT}(P)$ is uniquely defined for any simple polygon whose vertices are in general position. Once we have a $\mathrm{DT}(P)$, we define its dual graph $\mathrm{DT}(P)^*$: 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, $\mathrm{DT}(P)^*$ is always a tree.

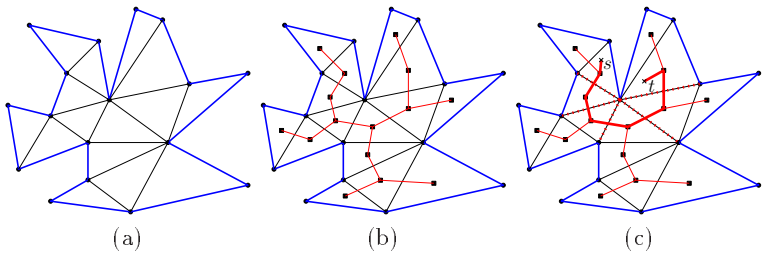

Fig. 2. The unique path on the dual graph and its corresponding sequence of Delaunay edges. (a) Constrained Delaunay triangulation of a given simple polygon, (b) the dual graph of the triangulation (a tree), and (c) the unique path between s and t.

Once we have a path in $\mathrm{DT}(P)^*$, we walk along the path while extending the visibility angle. Let $(e_0, e_1, \ldots, e_m)$ be an edge sequence corresponding to the path. We first compute the visibility angle defined by the starting point s and the first edge e_0. Then, we take the intersection between the current visibility angle with that defined by the next edge e_1. We keep the intersection as the current visibility angle. If the intersection becomes empty, we know that the line segment between the current vertex and previous starting point must be a part of the shortest path. So, we output that line segment and then we start the new

propagation of the visibility angle from the current vertex. Figure 3 illustrates how this algorithm proceeds.

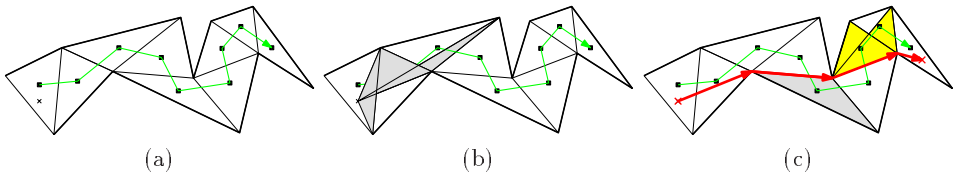

Fig. 3. Finding a shortest path along a sequence of Delaunay edges. (a) The unique path in the dual graph, (b) extension of visibility region from the starting point until it vanishes, and (c) shortest path within the simple polygon.

Finally, we need to describe how to implement the function **NextNeighbor**() for DT$(P)^*$. By the definition of the dual graph and the fact that each DT$(P)^*$ has maximum degree at most 3, the next neighbor is found by finding the clockwisely or or counter-clockwisely next Delaunay edge as shown in Figure 4. Hence, we need to find a third vertex of a Delaunay triangle for a given edge. Given a Delaunay edge (u, v), we want to find a vertex w such that
(1) w is visible from the edge (u, v), and
(2) the circle defined by three points u, v, and w is empty, that is, it does not contain any other vertex visible from the edge (u, v).

A vertex w is visible from the edge (u, v) when there is no edge intersecting a line segment $\overline{uw}$ or $\overline{vw}$. Thus, we can find in $O(n^2)$ time a vertex with which a Delaunay edge forms a Delaunay triangle.

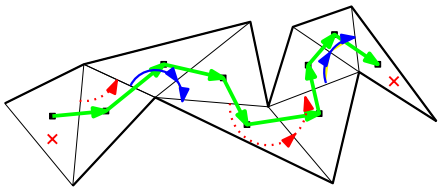

Fig. 4. Walking along a path in the dual graph while finding the clockwisely next Delaunay edge (solid or blue arrow) or counterclockwisely next edge (dotted or red arrow).

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. Such a shortest path can be found by applying the function **NextNeighbor**() $O(n^{1+\varepsilon})$ times. Since it takes $O(n^2)$ time for each call of the function, we obtain the bound in the theorem. □

3.2 A Shortest-Path Algorithm Using Point Location

The algorithm given above is obtained by a direct adaptation of the algorithm for reporting a simple path between two nodes in a tree. The dual graph, which is a tree, gives us a correct direction toward a given target point. In this section we show that there is a more direct way to find a correct direction. Suppose we are in some triangle A in $\mathrm{DT}(P)$. Removing A divides P into at most 3 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 found by walking along the boundary. This kind of operation is called *point location* in computational geometry. It takes $O(n)$ time since the total length of the boundary is $O(n)$.

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. 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 property as the triangulation used to have for shortest paths.

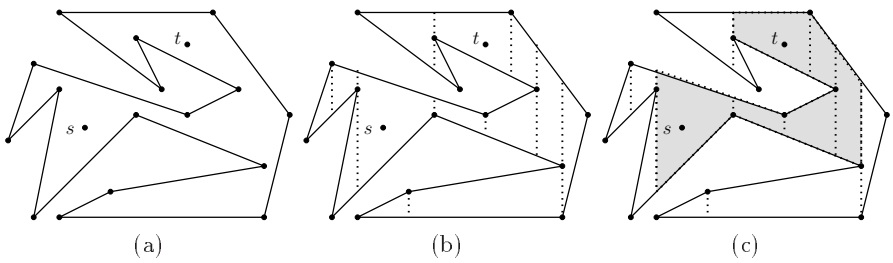

Fig. 5. Trapezoidal decomposition of a simple polygon for finding a shortest path in a simple polygon. (a) A simple polygon and two internal points s and t to be interconnected within the polygon. (b) Trapezoidal decomposition of P. (c) A sequence of trapezoids between two containing s and t.

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 6. 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 ε 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 6.

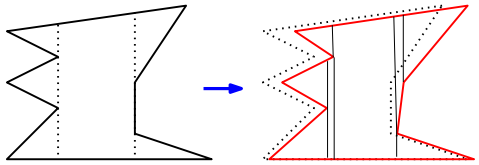

Fig. 6. Removing degeneracies by shifting vertices to the right. An original polygon is given to the left. The conversion results in the right polygon in which no two vertices share the same x-coordinate.

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 the two containing s and t. For the purpose, it suffices to find a 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 7. 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.

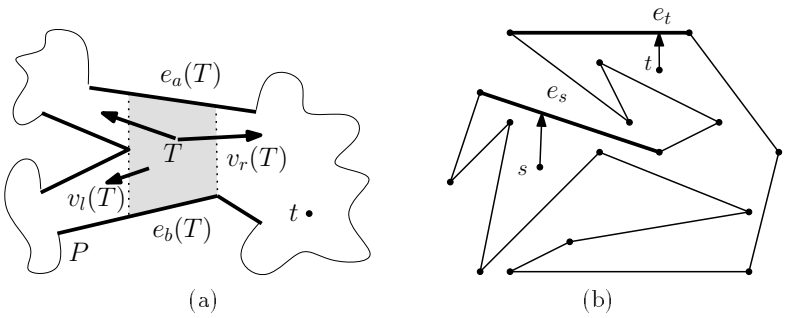

Fig. 7. Characterization of a trapezoid T by two polygon edges bounded from above and below and two vertical sides. (a) A trapezoid adjacent to three trapezoids. (b) A polygon edge e_s just above the point s and a polygon edge e_t just above t.

By the observation above we now 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 visible part of vertical sides of those trapezoids in the sequence. Whenever the entire side becomes invisible, we create a new bending point and recompute the visible part.

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. If we find the edge among them that is closest to q, it is the top edge $e_a(T)$ of the trapezoid T containing q. In a similar fashion we can find a polygon edge $e_b(T)$ just below q which is the bottom edge of the trapezoid that contains q. 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 in 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.

A trapezoid is specified in this way. Then, 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 is described above. Thus, once we have a trapezoid, we can find trapezoids adjacent to it 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.*

4 Concluding Remarks

We have presented a constant-work-space algorithm for finding a shortest path between two arbitrary points in a simple polygon in polynomial-time. 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 and G. Rote, "Constant-Working-Space Algorithms for Geometric Problems," Proc. CCCG, pp.87-90, Vancouver, 2009.
2. T. Asano, "Constant-Working-Space Algorithms: How Fast Can We Solve Problems without Using Any Extra Array?," Invited talk at ISAAC 2008, p.1, Dec. 2008.
3. T. Asano, "Constant-Working-Space Algorithms for Image Processing," Monograph: "ETVC08: Emerging Trends and Challenges in Visual Computing," ETVC 2008: pp.268-283, edited by Frank Nielsen, 2009.
4. T. Asano, "Constant-Working-Space Image Scan with a Given Angle," Proc. 24th European Workshop on Computational Geometry, March 18-20, pp. 165–168, 2008, Nancy, France.
5. T.Asano, Constant-Working Space Algorithm for Image Processing, Proc. of the First AAAC Annual meeting, Hong Kong, April 26-27, p. 3, 2008
6. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, "Computational Geometry: Algorithms and Applications, Third Edition" Springer-Verlag, 2008.
7. T. M. Chan, E. Y. Chen, "Multi-Pass Geometric Algorithms. Discrete & Computational Geometry 37(1), pp.79-102, 2007.
8. B. Chazelle, "Triangulating a simple polygon in linear time," Discrete and Computational Geometry, 6(1), pp.485–524, 1991.
9. J. I. Munro and V. Raman, "Selection from read-only memory and sorting with minimum data movement," Theoretical Computer Science **165**, pp.311–323, 1996.
10. O. Reingold, Undirected connectivity in log-space, J. ACM **55**, (2008), Article #17, 24 pp.