Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 0, no. 0, pp. 0–0 (0)

Constant-Work-Space Algorithms for Shortest Paths in Trees and Simple Polygons

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

[1]School of Information Science,
JAIST, Japan
[2]Institut für Informatik,
Freie Universität Berlin, Germany
[3]Microsoft Research Asia,
Beijing, China

Abstract

Constant-work-space algorithms model computation when space is at a premium: the input is given as a read-only array that allows random access to its entries, and the algorithm may use constantly many additional registers of $O(\log n)$ bits each, where n denotes the input size.

We present such algorithms for two restricted variants of the shortest path problem. First, we describe how to report a simple path between two arbitrary nodes in a given tree. Using a technique called "computing instead of storing", we obtain a naive algorithm that needs quadratic time and a constant number of additional registers. We then show how to improve the running time to linear by applying a technique named "simulated parallelization". Second, we show how to compute a shortest geodesic path between two points in a simple polygon in quadratic time and with constant work-space.

Submitted:	Reviewed:	Revised:	Accepted:
April 2010	November 2010	December 2010	May 2010
	Final:	Published:	
	Article type:		Communicated by:

A preliminary version of this paper appeared as T. Asano, W. Mulzer, and Y. Wang, *A Constant-Work-Space Algorithm for Shortest Paths in Simple Polygons.* in Proc. 4th WALCOM, pp. 9–20, 2010. T. Asano 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). W. Mulzer was supported by a Wallace Memorial Fellowship in Engineering, and in part by NSF grant CCF-0634958 and NSF CCF 08327.

E-mail addresses: t-asano@jaist.ac.jp (Tetsuo Asano) mulzer@inf.fu-berlin.de (Wolfgang Mulzer) yajunw@microsoft.com (Yajun Wang)

1 Introduction

We consider two restricted variants of the shortest path problem in a computational model that we call *constant-work-space computation*. In this model, which is also known as "log-space" [1], the input is given as a read-only array. Each cell of the array stores $O(\log n)$ bits and can be accessed in constant time. Additionally, the algorithm may use constantly many *registers*, each of which stores $O(\log n)$ bits and can be read and written in constant time. These registers are referred to as the *work-space* of the algorithm, and the number of registers is the size of the work-space.

This model has been investigated before. One of the most important results in this area is the selection algorithm by Munro and Raman [22] which runs in $O(n^{1+\varepsilon})$ time using work-space $O(1/\varepsilon)$, for any small constant $\varepsilon > 0$. More recently, a constant-work-space algorithm by Reingold [23] for determining whether there exists a path between two arbitrary vertices in an undirected graph solved a long-standing open problem in complexity theory. Asano [2–5] describes applications to image processing. Constant-work-space algorithms for some geometric problems are also known: Several authors describe algorithms for enumerating the vertices and facets of geometric arrangements without additional space [7,9,10,16,17,24]. Bose and Morin [11] describe how to find a path between two given vertices in a planar triangulation in an online setting that allows only constant space. Asano et al [6] give efficient constant-work-space algorithms for drawing the Delaunay triangulation and the 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. They also show a different algorithm for geodesic shortest paths in polygons.

Our setting is similar to the strict data-streaming model, where the algorithm also has only a restricted amount of work-space at its disposal. However, in the streaming model the input can be read only once in a sequential manner. Chan and Chen [12] give algorithms in different computational models varying from a multipass data-streaming model to the random access constant-work-space model considered here.

In this paper, we focus on the design of fast algorithms in the constant-work-space model. Using two restricted versions of the shortest path problem, we showcase some techniques for designing such algorithms. The first technique, named "**computing instead of storing**", is applied to the problem of finding a simple path between two nodes in a given tree. A simple solution using linear work-space goes as follows: compute an Eulerian path between the two nodes and count how often each edge appears on the path. Then remove those edges that appear twice. This gives the desired simple path. We can implement this idea using only constantly many registers: 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 we obtain a quadratic-time constant-work-space algorithm for the problem.

Another important technique is called "**simulated parallelization**". It may be considered as a generalization of the "baby-step, giant-step" method [19, 25], which uses two pointers with different speeds to detect a loop in a given linked list. In simulated parallelization, we proceed as follows: given a vertex u on the simple path, we want to determine which edge to follow toward the target vertex t. For this, we use two pointers for scanning the subtrees of u for t. We alternately advance these pointers until one of them exhausts its subtree or encounters t. If a pointer exhausts its subtree, we move it to the next remaining subtree of u. If no subtree is left, we can conclude that the subtree searched by the other pointer must contain t. Simulated parallelization is quite powerful. In fact, just by incorporating this technique into the naive quadratic-time algorithm we can improve its running time to linear.

The above algorithms can be extended to an algorithm for finding shortest geodesic paths in a simple polygon. Given a simple polygon P with n vertices and two points s and t in its interior, we would like to find the shortest path between s and t that stays within P. A naive approach leads to a cubic-time algorithm, but a more careful implementation yields a quadratic-time algorithm. See also Asano et al [6] for a different approach that also yields a quadratic time algorithm.

What about general weighted graphs? Of course, in the linear-work-space model there is the classic and popular shortest path algorithm by Dijkstra [15, Chapter 24], which can be implemented in $O(n^2)$ time using very simple data structures. Is it still possible to find an analogous algorithm in the constant-work-space model? Unfortunately, no such algorithm is known, and it is unlikely to exist, since the shortest path problem for general weighted graphs is NL-complete [18].

2 Simple Paths in Trees Using Eulerian Tours

As our first problem, we consider the following question: let T be a tree with n nodes. Given two distinct nodes s and t in T, find a simple path from s to t with no node visited more than once.

Here is a simple algorithm using $O(n)$ time and space: it is well known that every tree has an Eulerian tour, i.e., a closed walk that visits every edge exactly twice. Compute such a tour and take a subtour E that goes from s to t. Remove from E all edges that occur twice. This yields a simple path between s and t. Unfortunately, in our constant-work-space model no extra array can be used for storing the tour and finding the duplicate edges.

To obtain a constant-space algorithm, we apply the technique "**computing instead of storing**", where we recompute a subtour of the Eulerian tour whenever we need it. We assume that the tree is given by adjacency lists stored in a read-only array. Let 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 Adj(u).

NextNeighbor(u, v)**:** Given a node u and an adjacent node v, return the successor of v in 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. More precisely, when generating the Eulerian tour, after following an edge from a vertex v to a vertex u, we need to find the entry for v in $\text{Adj}(u)$ so that we can execute **NextNeighbor**(u, v). If the tree is given by a doubly-connected edge list (DCEL) [8, Chapter 2], this can be done in constant time. On the other hand, if we represent $\text{Adj}(u)$ as a simple list, we may need to search the whole list for the successor of v, 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, an algorithm that finds the shortest path from s to t is shown in Algorithm 1. The algorithm 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., (u, v) and all edges in the subtree appear twice in the tour and can be omitted (see Figure 1).

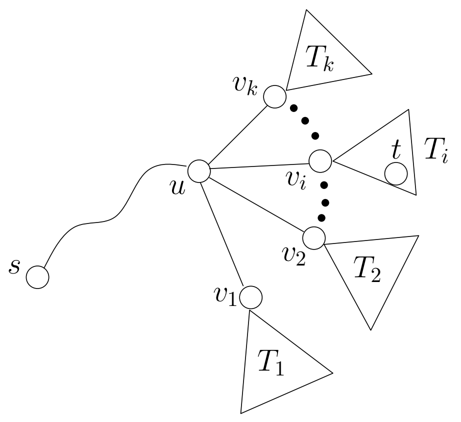


Figure 1: Which subtree of u contains the target node t?

Theorem 1 *Given a tree T with n nodes, a starting vertex s and a target vertex t, let ℓ be the length of the shortest path from s to t. There is an algorithm that reports the path from s to t in $O(\ell n d)$ time and with constant work-space. Here, d depends on which data structure is used: if T is given by a DCEL, then $d = O(1)$. If T is given by mere adjacency lists, then $d = O(\Delta)$, where Δ is the maximum node degree in T. The running time is always $O(n^2 d)$.*

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

```
Input: A tree $T$, two nodes $s$ and $t$ in $T$.
Output: A simple path from $s$ to $t$.
begin
    currentNode = $s$; startNeighbor = FirstNeighbor($s$);
    repeat
        report currentNode;
        $u$ = currentNode;
        currentNode = FindFeasibleSubtree($u$, startNeighbor, $t$);
        startNeighbor = NextNeighbor(currentNode, u) :
    until currentNode == $t$
// returns a child of $u$ whose subtree contains $t$
function FindFeasibleSubtree($u, v, t$)
begin
    for each node $w$ in Adj($u$), in order starting from $v$ do
        if SubtreeSearch($u, w, t$) then return $w$;
// checks whether the subtree of $u$ rooted at $v$ contains $t$
function SubtreeSearch($u, v, t$)
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 == $t$);
```

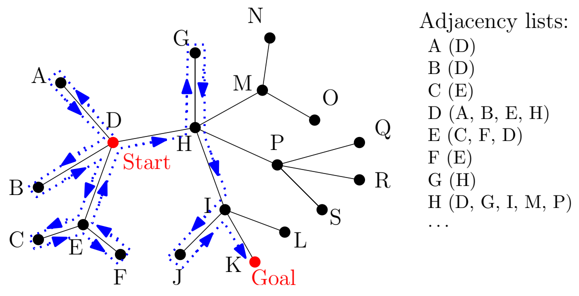
Proof: Refer to Algorithm 1. The algorithm starts from s and checks for each neighbor of s whether the corresponding subtree contains t. Then it proceeds to the appropriate neighbor and continues. Thus, it suffices to show correctness of the function **SubtreeSearch**(u, v, t) which checks whether the subtree of u rooted at v contains t. This is proved by induction on the height of the subtree.

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

Figure 2 illustrates how the search proceeds.

A Linear Time Algorithm. We now improve the running time of Algorithm 1 to $O(n)$, using "simulated parallelization".

Consider Algorithm 1. The worst case happens when each call to **FindFeasibleSubtree**(u, v, t) takes almost linear time. In other words, the subtrees containing t are large. Consider the edge (u, v). If t is in the subtree rooted at



Adjacency lists:
A (D)
B (D)
C (E)
D (A, B, E, H)
E (C, F, D)
F (E)
G (H)
H (D, G, I, M, P)
...

Figure 2: Canonical traversal of a tree given by adjacency lists.

v and if this subtree is large, **SubtreeSearch**(u, v, t) could take a long time. However, if instead we call **SubtreeSearch**(u, w, t), for all other children of u, we can quickly conclude that t is in the subtree for child v, with running time proportional to the size of those other subtrees. If we could guarantee that all subtrees not containing t are cut off from the search space at most once, we could safely charge the running time to the nodes in those subtrees, resulting in a linear-time algorithm.

The only apparent difficulty is that we do not know which subtree contains t. Let us start with a trivial solution that runs **SearchSubtree**(u, v, t) in parallel, one process for each subtree of u except the one containing s, such that the steps of the parallel processes are synchronized. Let $N(u, t)$ be the total number of nodes in the subtrees of u which do not contain s or t. We claim that we can find the subtree with t in $O(N(u, t))$ steps, where we sum the number of steps over all parallel processes. For this, we stop all parallel processes as soon as one of the following conditions is satisfied: (a) one process returns true; (b) all processes but one return false. Clearly, the total number of steps in all processes is at most $2N(u, t)$, as claimed.

Since we are working with a sequential machine, we must simulate the parallel execution by iteratively advancing each process in turn. However, if the maximum degree is linear, we cannot afford to simulate $O(n)$ processes since we have to store the internal states for all of them. Instead, we only maintain two copies of **SubtreeSearch**(u, v, t) at a time. We start another copy if one process finishes and if there are more subtrees to explore. Remember that we can stop if there is only one process left. In this way, we can find the simple path from s to t in linear time. For completeness, we present pseudocode for **FindFeasibleSubtree**(u, v, t) in Algorithm 2.

Theorem 2 *Given two nodes s and t in a tree T represented by a DCEL, there is an algorithm that finds the simple path from s to t in T in $O(n)$ time using $O(1)$ additional space.*

Proof: In Algorithm 2, **FindFeasibleSubtree**(u, v, t) takes time linear in the number of nodes in the subtrees of v which do not contain t. Notice that every subtree that does not contain t is explored at most once, due to our setting of `numOfNeighbors` in the algorithm. Therefore, Algorithm 1 takes linear time with the new implementation of **FindFeasibleSubtree**(u, v, t) in Algorithm 2, as claimed. □

3 Geodesic Shortest Paths in Polygons

We now turn to the problem of finding a shortest path between two arbitrary points s and t within a given polygon P. If linear work-space is allowed, there is a classic linear-time algorithm due to Lee and Preparata [20].

Their algorithm works as follows: We first partition the interior of P into triangles using Chazelle's method [13]. Then, we compute the dual graph G^* of the triangulation: the vertices of G^* correspond to the triangles, and two vertices are adjacent precisely if their corresponding triangles share an edge. We locate the points s and t in the triangulation. Since G^* is a tree, any two vertices in G^* are connected by a unique simple path. Consider the path between the triangle containing s and the triangle containing t. It defines a sequence $(e_0, e_1, \ldots, e_m)$ of diagonals hit by the path. The algorithm walks along this sequence while maintaining a *funnel*. The funnel consists of a *cusp* p, initially set to s, and of two concave chains from p to the two endpoints of the current diagonal e_i. In each step, there are two cases: (i) if the next diagonal remains visible from the cusp, we just update the appropriate concave chain, similar to Graham's scan; (ii) if the next diagonal is not visible from the cusp, we proceed along the appropriate chain until we find the cusp for the next funnel, and we output the vertices encountered along the way as part of the shortest path. Implemented properly, all this can be done in linear time (see Lee and Preparata [20] for details).

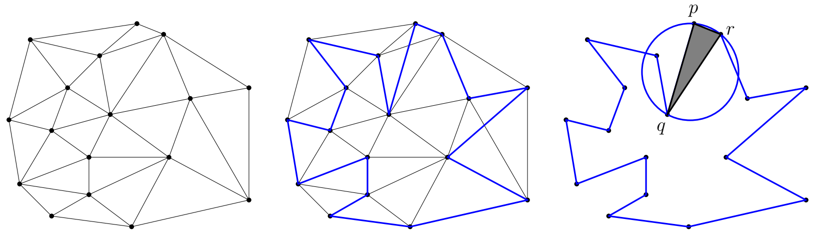
3.1 A Shortest-Path Algorithm Using the Dual Graph

We adapt the algorithm by Lee and Preparata [20] 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 P efficiently, we need a *canonical* triangulation of P. This means that for every triple of vertices in P, there should be an easily checkable condition to determine whether the triple defines a triangle or not. This allows us to recompute the individual triangles encountered on the path as needed.

Specifically, our canonical triangulation will be the *constrained Delaunay triangulation* [14]. 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*. The Delaunay triangles partition the convex hull of the set S, and the resulting structure is called the *Delaunay triangulation* of S [8, Chapter 9].

Constrained Delaunay triangulations offer a way to extend this notion to a simple polygon P [14].[1] 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 *diagonal*. A diagonal 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 diagonal; (ii) (p, r) and (q, r) are diagonals 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 does (r, s) define a diagonal.



(a) (b) (c)

Figure 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 there is a unique constrained Delaunay triangulation $\mathrm{DT}(P)$ for any simple polygon whose vertices are in general position. We denote the dual graph by $\mathrm{DT}(P)^*$. Recall that since a simple polygon is simply connected, $\mathrm{DT}(P)^*$ is a tree.

We would like to use Theorem 2 to find the shortest path from s to t in $\mathrm{DT}(P)^*$. For this, we need to implement the function **NextNeighbor**(). By the definition of the dual graph and the fact that each $\mathrm{DT}(P)^*$ has maximum degree at most three, 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 which completes a Delaunay edge to a Delaunay triangle, and this is also the time for an invocation of **NextNeighbor**(). We emphasize

[1]More generally, constrained Delaunay triangulations allow us to define a triangulation that contains a prespecified set of edges and is as close to the usual Delaunay triangulation as possible.

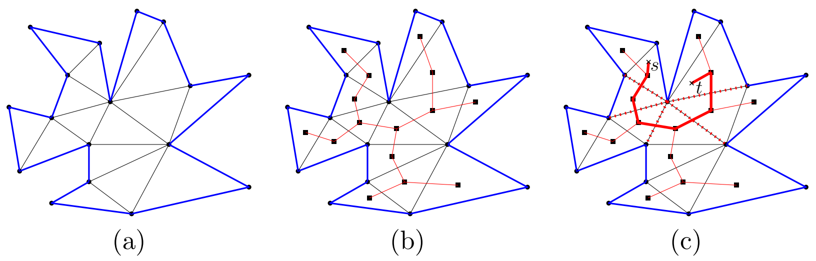


Figure 4: The unique path on the dual graph and its corresponding sequence of Delaunay edges. (a) The constrained Delaunay triangulation of a given simple polygon; (b) the dual graph of the triangulation (a tree); and (c) the unique path between s and t.

that we never store $\mathrm{DT}(P)^*$ explicitly, but we only access it on the fly by direct computation. This is another application of the "computing instead of storing" principle.

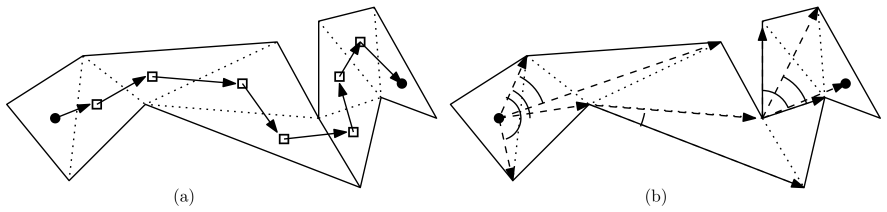


Figure 5: Finding a shortest path between two points within a simple polygon. (a) Walking along a path in the dual graph while finding the clockwise or counterclockwise next triangular edge; (b) the evolution of the visibility angles.

Maintaining the funnel. During the walk in $\mathrm{DT}(P)^*$ from s to t, 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 and 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 diagonal intersected by the shortest path, e_0. In order to process a new diagonal 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. See Figure 6. 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 (i.e., a sequence of gift-wrapping steps [15, Chapter 33.3]) 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 can be checked in $O(n)$ time per vertex). 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 7 for an example.

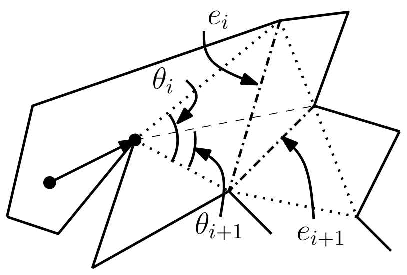


Figure 6: Angle changes while walking along the edge sequence.

Since every step in Jarvis's march needs $O(n)$ time, and since there are $O(n)$ such steps overall, the total running time for maintaining the funnel is $O(n^2)$. We have thus shown:

Theorem 3 *There is a constant-work-space algorithm for finding a shortest path between any two points inside a simple n-gon P in time $O(n^3)$.*

Proof: By Theorem 2, the shortest path in $\mathrm{DT}(P)^*$ can be found by $O(n)$ applications of the function **NextNeighbor**(). Since each such call takes $O(n^2)$ time, and since the total running time for maintaining the funnel as described above 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 3 comes from a direct adaptation of the algorithm by Lee and Preparata [20]. The dual graph 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, see also Asano et al [6] for another approach.

We assume without loss of generality that polygon vertices are numbered sequentially 0 through $n-1$ around the boundary (since the vertices of the polygon are given sequentially in our input array, these numbers could correspond to the addresses of the cells storing the vertices). Suppose we are in some triangle A in $\mathrm{DT}(P)$. Removing A divides P into at most three parts, and we need to find the part that contains t. To do this, we use the indices associated with the polygon edge e_t just above t. The part of the polygon that contains t is the part whose boundary contains the edge e_t, unless the upward vertical line segment from t to e_t crosses A before reaching e_t. The process of finding the part of the polygon that contains t is called *point location.* Since we just need to compare indices and check for intersection with A, point location takes just $O(1)$ time once we have precomputed e_t.

Now we know which way to go from any given triangle. Unfortunately, finding adjacent triangles in a canonical triangulation can be slow, and it turns

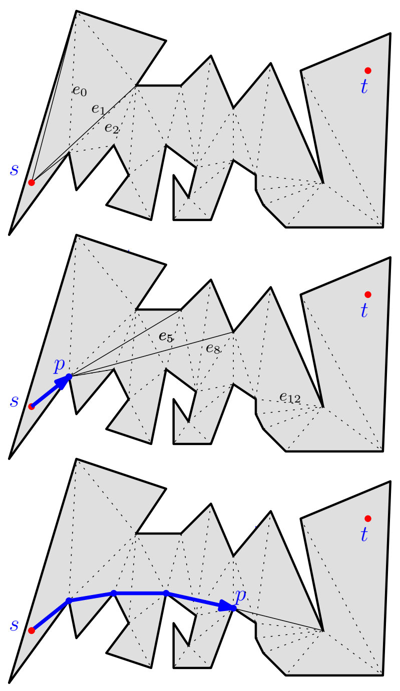


Figure 7: Maintaining the funnel. Initially, the cusp of the funnel is set to s, and a, b are the endpoints of the diagonal e_0. The visibility angle needs to be updated on encountering e_1 (top). At e_5, the visibility angle vanishes, and we must advance the cusp. Now, the visibility angle is updated at 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.

out to be more efficient to replace the constrained Delaunay triangulation by the *trapezoidal decomposition* [8, Chapter 6]. That is, we partition the interior of P by drawing a vertical chord at each vertex toward the interior of the polygon. This decomposition is canonical and easy to compute. Moreover, it offers the same properties that the triangulation used to have for shortest paths.

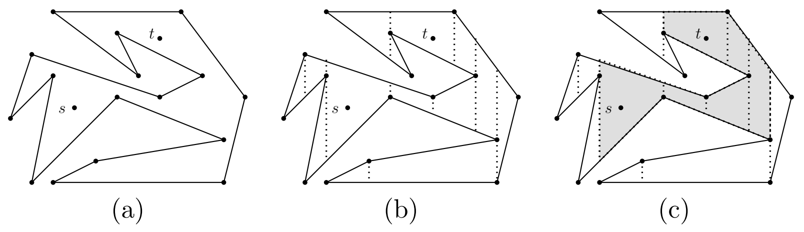


Figure 8: Trapezoidal decomposition of a simple polygon for finding a shortest path in a simple polygon. (a) A simple polygon and two internal points s and t to be interconnected within the polygon. (b) Trapezoidal decomposition of P. (c) A sequence of trapezoids between two containing s and t.

The trapezoidal decomposition defined above is uniquely determined for any simple polygon. However, degeneracies can cause one trapezoid to be adjacent to arbitrarily many other trapezoids, as shown in Figure 9. We perform a symbolic perturbation to avoid this issue: the vertices of P and the two points s and t all have integral coordinates with $O(\log n)$ bits. Each integral point (x, y) is treated as a point $(x + y\varepsilon, y)$, i.e., 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^* appearing in the input. After this perturbation, no two vertices share the same x-coordinate, as shown to the right in Figure 9.

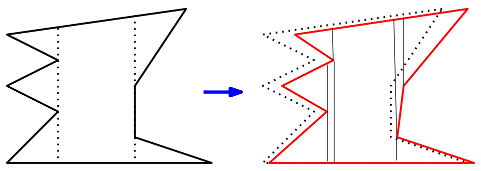


Figure 9: Removing degeneracies by shifting vertices to the right. An original polygon is given to the left. The conversion results in the right polygon in which no two vertices share the same x-coordinate.

From now on, we assume that every trapezoid is adjacent to at most four other trapezoids. Our first goal is to find the sequence of trapezoids between those containing s and t. For this, it suffices to find the correct neighbor at each trapezoid along the path.

A characterization of a trapezoid is given in Figure 10. Given an arbitrary point q in the interior of P, we can determine the 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 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.

At a trapezoid T we have to find its neighbors and determine how to proceed toward the target point t. The first question can be solved as follows: 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, we can find all adjacent trapezoids in $O(n)$ time. To determine which trapezoid to follow, simply traverse the boundaries of the regions divided by the adjacent trapezoids to find which region contains the edge e_t just above t. This is done using indices of endpoints of those subpolygons associated with the current trapezoid.

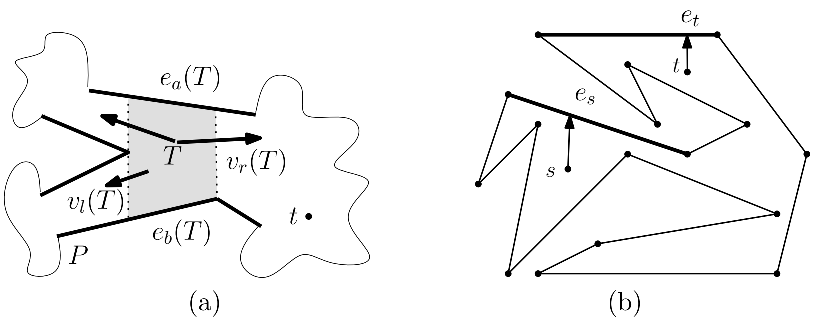


Figure 10: Characterization of a trapezoid T by two polygon edges bounded from above and below and two vertical sides. (a) A trapezoid adjacent to three trapezoids. (b) A polygon edge e_s just above the point s and a polygon edge e_t just above t.

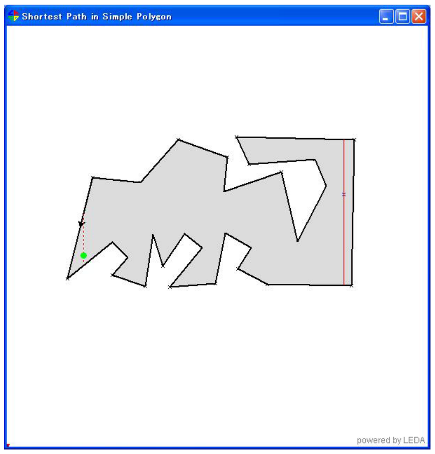
Hence, we can find the correct neighbor in $O(n)$ time with constant work-space. Since we need to traverse $O(n)$ trapezoids, the total time is $O(n^2)$.

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

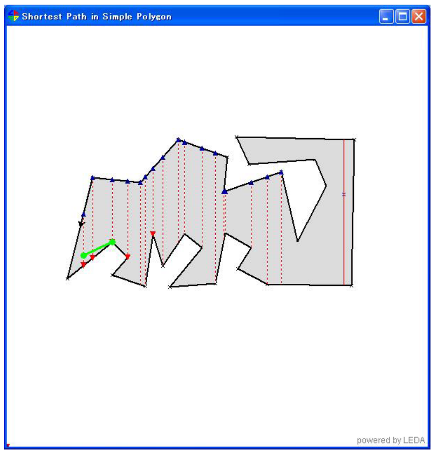
Theorem 4 *Given a simple polygon P with n vertices and two arbitrary points*

s and t in P, we can find a geodesic shortest path between s and t in $O(n^2)$ time with constant work-space.

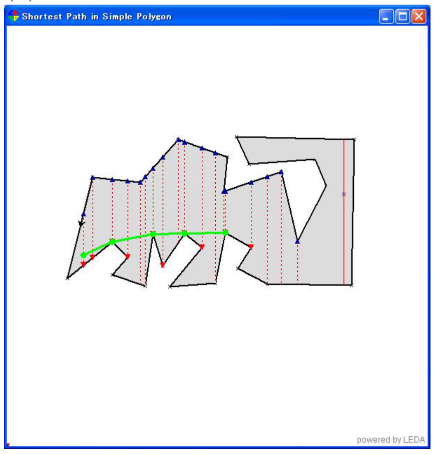
We have implemented our algorithm using LEDA [21] to check the details. An experimental result is shown in Figure 11. The input configuration is shown in (a). After finding the initial segment on the shortest path in (b), we repeatedly generate next trapezoids until the visibility angle vanishes, i.e., the last vertical edge is not visible from the current cusp of the funnel. This happens in (c), so the shortest path is extended and the cusp is moved as shown in (c). The final result is appears in (d). The shortest path is given by bold lines in the figure.



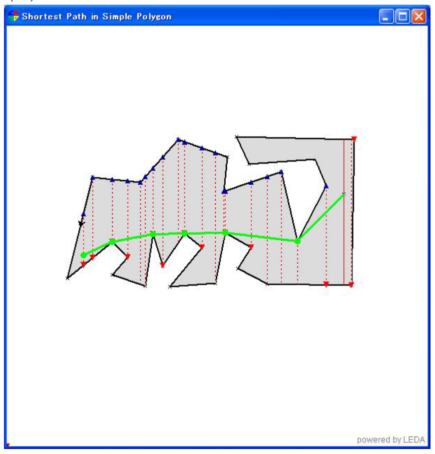
(a)



(b)



(c)



(d)

Figure 11: Experimental results. The cusp of the current funnel is shown as the last vertex of the green path.

4 Concluding Remarks

We have presented constant-work-space algorithms for finding shortest paths in trees and polygons. One obvious future direction is to extend the latter algorithm to the general Euclidean shortest path problem in the presence of polygonal obstacles. If the number of obstacles is bounded by some small constant k, then there are $O(n^k)$ different ways to convert the problem into one on a simple polygon by connecting the 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.

A number of geometric problems are open in the constant work-space model. For example, does there exist an efficient constant-work-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 necessary to find a shortest path in a simple polygon in linear time?

References

[1] S. Arora and B. Barak. *Computational complexity. A modern approach.* Cambridge University Press, Cambridge, UK, 2009.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-working-space algorithms for geometric problems. *Journal on Computational Geometry*, page to appear, 2011. See also CCCG 2009.

[7] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8(3):295–313, 1992.

[8] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, third edition, 2008.

[9] M. de Berg, M. J. van Kreveld, R. van Oostrum, and M. H. Overmars. Simple traversal of a subdivision without extra storage. *International Journal of Geographical Information Science*, 11(4):359–373, 1997.

[10] P. Bose and P. Morin. An improved algorithm for subdivision traversal without extra storage. *Internat. J. Comput. Geom. Appl.*, 12(4):297–308, 2002.

[11] P. Bose and P. Morin. Online routing in triangulations. *SIAM J. Comput.*, 33(4):937–951, 2004.

[12] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.

[13] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.

[14] L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, Cambridge, MA, third edition, 2009.

[16] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[17] C. M. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. *Int. J. Geographical Information Systems*, 1(2):137–148, 1987.

[18] A. Jakoby and T. Tantau. Logspace algorithms for computing shortest and longest paths in series-parallel graphs. Technical Report TR03-077, ECCC Reports, 2003.

[19] D. Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, 1973.

[20] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[21] K. Mehlhorn and S. Näher. *LEDA. A platform for combinatorial and geometric computing.* Cambridge University Press, Cambridge, 1999.

[22] 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.

[23] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):Art. #17, 24 pp., 2008.

[24] G. Rote. Degenerate convex hulls in high dimensions without extra storage. In *Proc. 8th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 26–32, 1992.

[25] D. Shanks. The infrastructure of a real quadratic field and its applications. In *Proc. Number Theory Conference*, pages 217–224, 1972.

Algorithm 2: New implementation of **FindFeasibleSubtree**(u, v, t).

```
// return a child of $u$ whose subtree contains $t$.
function FindFeasibleSubtree$(u, v, t)$
begin
    fNeighbor = $v$
    lastNeighbor = sNeighbor = NextNeighbor$(u, v)$
    numOfNeighbors = $\deg(u) - 1$
    // in case of $s$, we must explore all subtrees.
    if $u == s$ then numOfNeighbors++
    if numOfNeighbors == 1 then return fNeighbor
    one = $u$; oneNext = fNeighbor; two = $u$; twoNext = sNeighbor;
    while true do
        // advance two copies.
        (sigf1, sigc1, one, oneNext) = AdvSearch($u$, fNeighbor, one,
        oneNext)
        (sigf2, sigc2, two, twoNext) = AdvSearch($u$, sNeighbor, two,
        twoNext)
        if sigf1 then return fNeighbor
        if sigf2 then return sNeighbor
        if not sigc1 then
            // copy 1 finishes without finding $t$.
            oneNext = fNeighbor = NextNeighbor($u$, lastNeighbor)
            lastNeighbor = fNeighbor; one = $u$
            numOfNeighbors--
            if numOfNeighbors == 1 then return sNeighbor
        if not sigc2 then
            // copy 2 finishes without finding $t$.
            twoNext = sNeighbor = NextNeighbor($u$, lastNeighbor)
            lastNeighbor = sNeighbor; two = $u$
            numOfNeighbors--
            if numOfNeighbors == 1 then return fNeighbor
// advance the Eulerian tour in the subtree rooted at $v$.
function AdvSearch$(u, v, u', v')$
begin
    $v''$ = NextNeighbor$(v', u')$; $u'' = v'$;
    if $u'' == v$ and $v'' == u$ then
        return (found = false, continue = false, $u''$,$v''$)
    if $v'' == t$ then
        return (found = true, continue = false, $u''$,$v''$)
    return (found = false, continue = true, $u''$,$v''$)
```