Memory-Constrained Algorithms for Simple Polygons[*]

Tetsuo Asano[†] Kevin Buchin[‡] Maike Buchin[‡] Matias Korman[§] Wolfgang Mulzer[¶]
Günter Rote[¶] André Schulz[‖]

Abstract

A constant-work-space algorithm has read-only access to an input array and may use only $O(1)$ additional words of $O(\log n)$ bits, where n is the input size. We show how to triangulate a plane straight-line graph with n vertices in $O(n^2)$ time and constant work-space. We also consider the problem of preprocessing a simple n-gon P for shortest path queries, where P given by the ordered sequence of its vertices. For this, we relax the space constraint to allow s words of work-space. After quadratic preprocessing, the shortest path between any two points inside P can be found in $O(n^2/s)$ time.

1 Introduction

In algorithm development and computer technology, we observe two opposing trends: On the one hand, there are vast amounts of computational resources at our fingertips. On the other hand, more and more specialized tiny devices with limited memory or power supply become available. The first trend leads to software that is written without regard to resources; with this approach, today's latest equipment, be it workstations or hand-held devices, will soon appear unacceptably slow. It is the second trend on which we will focus here: we want to process data with a limited amount of memory. In particular, we consider the setting where the input data is given in a read-only data structure and we have only s words of memory at our disposal, for a parameter $s = o(n)$.

The input of our problem is a polygon P of n vertices in a read-only data structure. We assume that we can access the x- and y-coordinates of any polygon vertex in constant time. We also assume that we can obtain the (clockwise and counterclockwise) neighbor vertex in constant time. We count the storage in terms of the additional number s of *cells* or *words* that an algorithm uses. As usual, a word is large enough to contain either an input item (such as a point coordinate) or an index into the data (of $\log n$ bits). We also assume that basic operations on the input (such as determining if a point is above a given line) take constant time.

Our Results. First, we show how to triangulate a plane straight-line graph (and hence a simple polygon) with constant work-space in $O(n^2)$ time (Section 3). Then, in Section 4, we apply this result to the construction of memory-adjustable data structures for shortest path queries. That is, given a polygon P and a parameter s, we construct a data structure of size $O(s)$ for P. We then use this structure to compute the shortest path between any two points inside P in $O(n^2/s)$ time using $O(s)$ work-space.

Related Work. Given their many applications, a significant amount of research has been dedicated to memory-constrained algorithms, even as early as in the 1980s [6]. A classic example with a geometric flavor is the well-known gift-wrapping algorithm (also known as Jarvis march [7]). It computes the convex hull of a planar n-point set in $O(nh)$ time and $O(1)$ work-space, where h is the number of convex hull vertices. The systematic study of constrained memory in a geometric context was initiated by Asano et al. [2]. Among other results, they describe how to construct well-known geometric structures (such as Delaunay triangulations, the Voronoi diagrams and minimum spanning trees) with a constant number of variables. Recently, Barba et al. [3] gave a constant-work-space algorithm for computing the visibility region of a point inside a simple polygon.

Although we know of no algorithm that triangulates a simple polygon with $o(n)$ work-space, it is known how to find an ear of a given simple polygon P in linear time and with constantly many variables [5]. However, since the input cannot be modified, there seems to be no easy way to extended this method in order to obtain a complete triangulation of P. We also note that there exists a method for triangulating planar point sets [2]. The same paper also contains an

[*]This work was initiated at the Dagstuhl Workshop on Memory-Constrained Algorithms and Applications, November 21–23, 2011. We are deeply grateful to the organizers as well as the participants of the workshop for helpful discussions during the meeting.

[†]JAIST, Japan, t-asano@jaist.ac.jp

[‡]TU Eindhoven, Netherlands, {k.a.buchin,m.e.buchin}@tue.nl MB is supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 612.001.106.

[§]ULB Brussels, Belgium, mkormanc@ulb.ac.be

[¶]Freie Universität Berlin, Germany, {mulzer,rote}@inf.fu-berlin.de

[‖]WWU Münster, Germany, andre.schulz@uni-muenster.de

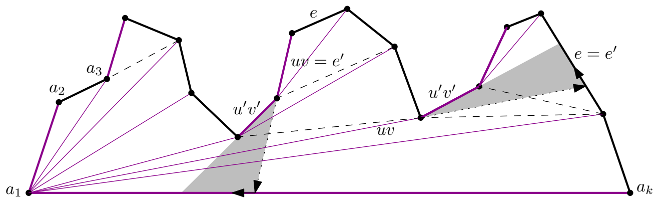

Figure 1: The shortest path tree SPT from a_1 to all other vertices (in purple). Additional triangulation edges generated in our algorithm are dotted. The figure illustrates a forward step (right shaded area) and a backward step (middle area).

algorithm for finding the shortest path between any two points inside a simple polygon. This algorithm uses constant work-space and runs in quadratic time.

For space reasons, we can only give a sketchy idea of the algorithms here. A more complete description is available in the full version [1].

2 Triangulating a Mountain

A *monotone mountain* (or *mountain* for short) is a polygon H whose vertices $a_1, a_2, \ldots, a_k$ have increasing x-coordinates. The edge $a_1 a_k$ is called the *base* of H. In this section, we describe how to triangulate an explicitly given mountain, while using only one scan over the boundary of H. This will allow us to extend the algorithm in the next section in order to triangulate a mountain that is provided implicitly as part of a decomposition of a plane straight-line graph.

For the algorithm, we need the *shortest-path tree* SPT from a_1 to all other vertices of H. We make the following observations, see Figure 1.

Proposition 1 *SPT has the following properties:*
(i) SPT makes only upward bends.
(ii) Each face f of SPT is bounded by the shortest paths from a_1 to two consecutive vertices a_i and a_{i+1}. (This holds in any simple polygon.)
(iii) Each face f of SPT is a pseudotriangle, bounded from below by an SPT edge ua_{i+1}, from the right by an edge $a_i a_{i+1}$ of H, and from above by a concave chain of SPT edges (as seen from inside).
(iv) The face f can be triangulated uniquely by connecting the rightmost vertex a_{i+1} with the reflex vertices on the upper boundary.

The algorithm traverses SPT in depth-first order, visiting the children of each vertex in *counterclockwise* order. Since there is no space for a stack, this must be done in a "stateless" manner. As in an Euler tour, we visit each non-leaf edge twice, once in forward and once in backward direction. However, a backward step immediately followed by a forward step is considered as a single *sideways* step (this corresponds to the case where the tour moves from a node to its sibling in SPT). We call a vertex *finished* if it has been visited by the traversal and will not be visited again. Otherwise, the vertex is *unfinished.* In an Eulerian traversal of SPT, the vertices of H become finished in order from right to left.

Our algorithm maintains two edges: (i) the current edge of the tour uv, with v lying to the right of u; and (ii) the edge $e = a_i a_{i+1}$ of H such that $\{a_{i+1}, a_{i+2} \ldots, a_k\}$ are the finished vertices of the tour. In each step we distinguish three different cases, and we accordingly perform a step as follows.

Case 1: *v is not incident to e.* We perform a *forward* step into the subtree rooted at v.

Case 2: *$v = a_i$, but uv is a chord of H.* We perform a *sideways* step to the next child of u that follows v in counterclockwise order.

Case 3: *$v = a_i$, and uv is an edge $a_{i-1}a_i$ of H.* We do a *backward* step and return to the parent of u.

We start the algorithm with a sideways step from $uv := e := a_1 a_k$ (as an exception to the above rules). The algorithm continues until all vertices are finished and it tries to make a backward step from $e = a_1 a_2$. The implementation of the three steps is straightforward. The only information about H that is required by the algorithm is one sequential scan of the sequence of vertices $a_1, a_k, a_{k-1}, \ldots, a_2, a_1$. Thus, the algorithm can also be applied if H is given implicitly and if it takes $O(n)$ time to advance from one edge to the next, as in the next section.

Theorem 2 *Let H be a mountain with k vertices given as part of a larger plane straight-line graph with n vertices. Then we can output the triangles of a triangulation of H in time $O(nk)$ and with constant work-space.*

For an explicitly given mountain of k vertices, we get a running time of $O(k^2)$ as a special case.

3 Triangulating a Plane Straight-Line Graph

Let G be a plane straight-line graph. In order to apply Theorem 2 to the problem at hand, we decompose the convex hull of G into mountains by computing the *vertical decomposition* (or *trapezoidation*) of G and by inserting a chord between any two nonadjacent vertices of G contained in the same trapezoid (cf. Chazelle and Incerpi [4]). These edges partition the convex hull into mountains: Since every vertex (except for the left- and the rightmost ones) has at least one incident edge to either side, the faces must be monotone polygons f. By definition of the decomposition, if there were a face f with vertices on both the upper and the lower boundary, there would be a trapezoid with a vertex at the upper and at the lower boundary. In this case, however, we would have inserted a diagonal.

An edge e of G or of the convex hull is a lower base of a mountain if and only if it is incident to more than one trapezoid above it. This can be checked in linear time. (An inserted chord is never the base of a mountain.) Once the base of a mountain H is at hand, we can visit the edges of H in counterclockwise order by enumerating the trapezoids incident to each vertex of H. This takes linear time per trapezoid. Now, in order to triangulate G, we consider each edge of G and each convex hull edge and determine whether it is a base of a mountain H. If so, we triangulate H using Theorem 2. The convex hull edges can be found in linear time per edge through Jarvis march [7]. Thus, our algorithm needs $O(n^2)$ time plus the time for triangulating the mountains. Since the total size of all mountains is $O(n)$, we get the following result.

Theorem 3 *Given a plane straight-line graph G with n vertices, we can output all triangles in a triangulation of G in $O(n^2)$ time with constant work-space.*

4 Memory-Adjustable Data Structures

Generally, the purpose of a *data structure* D for some set S of n objects is to answer certain types of *queries* on the set S efficiently. For this, D stores some useful information about S. In the best case, D has linear size, and the query algorithm searches within D with only $O(1)$ work-space (more precisely, a work-space of $O(\log n)$ bits). In the classical setting, the whole input data is contained in the data structure, so the storage must be at least as large as the input.

Here, we take a different approach: recall that our input is read-only and cannot be modified. Thus, our approach is to preprocess the data and to store some additional information in a data structure of size s (for some parameter $s = o(n)$). The objective is to design an algorithm that uses this additional information in a way that allows for efficient query processing.

In the following, we use this model for computing the shortest path between two points inside a simple polygon P. In our allotted space, we store a partition of P with $O(s)$ chords into $O(s)$ subpolygons with $\Theta(n/s)$ vertices each. The boundary of each subpolygon is given by subsequences of the original polygon boundary and some chords (hereafter called *cut edges*) associated with it. Additionally, we store the adjacencies between the subpolygons across the cut edges. The total space to represent these subpolygons is $O(s)$.

Theorem 4 *Let P be an n-gon, and let $s \in \{1, \ldots, n\}$. There are $O(s)$ pairwise non-intersecting chords that partition P into $O(s)$ subpolygons, each having $\Theta(n/s)$ vertices. The chords can be found in $O(n^2)$ time using constant work-space.*

We now describe the query algorithm. Given two points $p, q \in P$, we would like to compute the geodesic π_{pq} between them—putting to use the precomputed polygon decomposition. The general idea is to apply the constant work-space method of Asano et al. [2] and to concatenate the resulting paths. Let us thus first give a quick overview of this method.

The algorithm of [2] uses the following invariant: we have partially reported the shortest path from p up to an intermediate point v (where either $v = p$ or v is a reflex vertex of P). Furthermore, we store a *visibility cone* (i.e., a search direction) in which we know that π_{pq} must go. The visibility cone is represented by a wedge emanating from v. The algorithm then shoots a ray inside the wedge that partitions the polygon into two parts. Depending on which part point q is in, we might obtain a new intermediate point, or a portion of the polygon will be discarded.

Our algorithm uses a similar strategy, but it restricts itself to a subpolygon whenever possible. We start by locating the subpolygons P_p, P_q containing p and q. If $P_p = P_q$, we apply the constant work-space method within that subpolygon. Otherwise, consider the tree associated with the polygon partition and find the path between P_p and P_q. Every edge on that path corresponds to a polygon chord that π_{pq} must cross, in the same order. On the other hand, edges of the tree that were not on the path will not be crossed by π_{pq}, so the corresponding cut edges are treated as obstacles.

We first introduce some notation: let P_{curr} be the polygon containing the current vertex v. For any subpolygon P_i that is traversed by π_{pq}, we define the *entrance* as the cut edge that π_{pq} must cross to enter P_i. Analogously, we define the *exit* of P_i.

In the *standard situation*, everything is confined within one subpolygon P_{curr}. In this case, we can apply the search strategy of the constant-work-space algorithm almost without change: pick a search direction, according to the same rules as in [2], and shoot a ray R' partitioning the subpolygon into two.[1] We use the exit chord of P_{curr} to determine which half contains the target. The only problem arises when R' hits the boundary in the exit chord.

In this case, we switch to the *long-jump situation*. We must first complete the ray shooting operation: we continue the ray R' into the adjacent subpolygon. If R' again hits the exit chord of this subpolygon, we continue into the third subpolygon, and so on, until R' hits a point p' on the boundary of P. The ray R' splits the wedge into two parts, and we update the wedge to the part that contains the target. The running time is $O(n/s)$ times the number of subpolygons visited.

[1]We treat the case that v is outside P_{curr} and the two rays bounding the visibility cone hit the boundary of P in P_{curr} as in the standard situation. This extension does not affect the algorithm.

In the general long-jump situation we have a current start vertex v and two shortest paths SP^+ from v to a point p^+ and SP^- from v to a point p^-, forming a *funnel* with apex v. As an invariant, we know that the target lies between p^+ and p^-, and so the shortest path must go into the funnel. Note that p^+ and p^- may lie in different subpolygons P^+ and P^-. In this case, w.l.o.g., we assume that P^+ is more advanced than P^-. Then, our first goal is to incrementally extend the shortest path SP^- to the lower endpoint of entrance gate of P^+ (Procedure *Catch-up*). Whenever $P^- = P^+$, we shoot a ray R' and extend one of the SP edges (Procedure *Extend*), and proceed depending on which side of the ray R' the exit of P^+ lies. From the funnel boundaries SP^+ and SP^-, we only store the $O(s)$ SP edges that cross some cut edge. We now give the details of the two procedures.

Procedure Catch-up. Since our goal is to proceed towards the exit of P^-, we make an angular clockwise sweep inside P^- from the endpoint p^- of the current path SP^-, starting from the direction opposite to the last edge of SP^- and only considering the $O(n/s)$ points of P^- on the lower boundary between the entrance and the exit cut. After determining the first point that is hit, we may remove some of the last vertices from SP^- (possibly also from the beginning of SP^+). Finally, we add a new edge to SP^-.

Since we do not explicitly store SP^- and SP^+, we may have to look for the predecessor or successor edge by a counter-clockwise angular sweep, as in the backward step of Section 2. These operations are only needed if the point that we look for lies in the same subpolygon, hence we will need at most $O(n/s)$ time (which we charge to the removed vertex). The shortest path SP^- will eventually reach the lower endpoint of the exit gate of P^-. Then we advance to the next subpolygon, and iterate until we reach P^+.

Procedure Extend. Assume that one of the two funnel boundaries, say SP^-, has more than one edge. We take the last-but-one edge of SP^- and extend it into P^-. If this ray R' does not hit the exit cut of P^+, we determine, in $O(n/s)$ time, on which side of R' the target lies. If it lies above R' we pop the last edge of the funnel, as in Procedure *Catch-Up*, and proceed. If it lies below R' we can throw away everything above R'. The ray R' and the last SP edge start at the same vertex and end in P^+, forming a new funnel. We can thus consider P^+ with these two extra edges as our current polygon P_{curr}, and proceed.

There is also the situation that R' exits through the exit gate. In this case, we extend R' until it hits the boundary of P, in some subpolygon P'. Again, we determine on which side the target lies. If the target lies above R', R' forms the new last edge of SP^-, P^- is advanced to P'. If the target lies below R', we have a simple funnel consisting only of R' and the last edge of SP^-. We advance P^+ to P'. In either case we continue with procedure *catch-up*.

Theorem 5 *Let P be an n-gon, and $1 \leq s \leq n$. We can build a data structure of size $O(s)$ in by $O(n^2)$ time and $O(1)$ variables such that shortest path queries in P can be computed in $O(n^2/s)$ time using $O(s)$ variables.*

5 Open Problems

Obvious topics for future research are improvements of the results. For example, it would be interesting to construct a memory-adjustable data structure for triangulating a plane straight-line graph. Also, Theorem 4 describes how to find a good cut edge for a simple polygon by essentially triangulating the whole polygon and giving the most balanced cut. A natural question is if we can obtain a balanced cut in subquadratic time. Moreover, the cut would not necessarily have to be a diagonal connecting two vertices.

References

[1] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *CoRR*, abs/1112.5904, December 2011.

[2] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *J. Computat. Geometry*, 2(1):46–68, 2011.

[3] L. Barba, M. Korman, S. Langerman, and R. Silveira. Computing the visibility polygon using few variables. In *Proc. 22nd International Symposium on Algorithms and Computation (ISAAC'11)*, volume 7074 of *Lecture Notes in Computer Science*, pages 80–89. Springer-Verlag, 2011.

[4] B. Chazelle and J. Incerpi. Triangulation and shape-complexity. *ACM Trans. Graph.*, 3:135–152, 1984.

[5] H. ElGindy, H. Everett, and G. Toussaint. Slicing an ear using prune-and-search. *Pattern Recogn. Lett.*, 14:719–722, September 1993.

[6] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[7] R. Seidel. Convex hull computations. In *Handbook of Discrete and Computational Geometry*, chapter 22, pages 495–512. CRC Press, Inc., 2nd edition, 2004.