Time-Space Trade-offs for Triangulating a Simple Polygon*

Boris Aronov† Matias Korman‡ Simon Pratt§ André van Renssen¶‖ Marcel Roeloffzen¶‖

October 11, 2015

1 Introduction

An s-workspace algorithm is an algorithm that has read-only access to the input data and uses only $O(s)$ additional words of space. We give a randomized s-workspace algorithm for triangulating a simple n-gon P, for any $s \in \Omega(\log n) \cap O(n)$, in $O(n^2/s)$ expected time. Minor modifications of our approach can be used to compute other similar structures such as the shortest-path map (or tree) of any point $p \in P$, or to partition P by diagonals into sub-polygons of size $\Theta(s)$. See arXiv preprint for details [1].

The first constrained-workspace algorithm for triangulating simple polygons, due to Asano *et al.* [2], runs in $O(n^2)$ time using $O(1)$ variables. An algorithm for triangulating monotone polygons, by Asano and Kirkpatrick [3], requires $O(n \log_s n)$ time using $O(s)$ variables. Despite extensive research on the problem, neither time-space trade-off algorithm for general simple polygons nor trade-off lower bounds are known.

2 Preliminaries

We use the *s-workspace* model of computation in which the input data is given in a read-only array or a similar structure. In our case, the input is a simple polygion P, given by the list $v_1, v_2, \ldots, v_n$ of its vertices in clockwise order around its boundary. We assume that, given an index i, in constant time we can access the coordinates of the vertex v_i. We also assume that the usual word RAM operations (say, given i, j, k, finding the intersection point of the line v_iv_j and the horizontal line through v_k) can be performed in constant time.

Besides read-only data, an s-workspace algorithm can use only $O(s)$ variables for internal computation. Implicit memory consumption (such as the stack space needed in recursive algorithms) is included in the size of the workspace. We assume that each variable or pointer is stored in a data word of $\Theta(\log n)$ bits. Thus an s-workspace algorithm uses $O(s \log n)$ bits of storage.

We study the problem of triangulating a simple n-gon P in this model. A *triangulation* of P is a maximal crossing-free straight-line graph whose vertices are the vertices of P and whose edges lie inside P. Unless s is very large, the triangulation cannot be stored explicitly. Thus, the goal is to report a triangulation of P in a write-only data structure. After a value is reported, it cannot be accessed or modified.

3 Algorithm

Let π be the geodesic between v_1 and $v_{\lfloor n/2 \rfloor}$. At high level, our algorithm runs the procedure of Har-Peled [5] to incrementally compute π in expected $O(n^2/s)$ time. We use the edges of π to subdivide P into smaller problems that can be solved recursively.

Vertices v_1 and $v_{\lfloor n/2 \rfloor}$ split the boundary of P into two chains. We say v_i is a *top* vertex if $1 < i < \lfloor n/2 \rfloor$ and a *bottom* vertex if $\lfloor n/2 \rfloor < i \leq n$; c is an *alternating* diagonal if it connects a top and a bottom vertex. We will use alternating diagonals to partition P into two parts. For simplicity of exposition, given a diagonal d, we regard both components of $P \setminus d$ as closed sets (i.e., the diagonal belongs to both components).

Observation 1. *Let c be a diagonal of P not adjacent to v_1 or $v_{\lfloor n/2 \rfloor}$. Then v_1 and $v_{\lfloor n/2 \rfloor}$ belong to different components of $P \setminus c$ if and only if c is an alternating diagonal.*

Corollary 1. *Let c be a non-alternating diagonal of P. The component of $P \setminus c$ that contains neither v_1 nor $v_{\lfloor n/2 \rfloor}$ has at most $\lceil n/2 \rceil$ vertices.*

While triangulating the polygon, we maintain an alternating diagonal a_c, such that the connected component of $P \setminus a_c$ not containing $v_{\lfloor n/2 \rfloor}$ has already been triangulated, and at least one endpoint of a_c is a vertex of π that has already been computed by the shortest-path algorithm.

With these definitions in place, we can give an intuitive description of our algorithm: we start by setting a_c as the degenerate diagonal from v_1 to v_1. We then use the shortest-path algorithm of Har-Peled to walk along π until we find a new alternating diagonal a_{new}. At that moment we pause the traversal of π, recursively triangulate the subpolygons of P that have been created (and contain neither v_1 nor $v_{\lfloor n/2 \rfloor}$), update a_c to the newly found alternating diagonal, and then resume the shortest-path algorithm.

Although our approach is intuitively simple, we must address several technical difficulties. Ideally, the set of diagonals we walked along π is small enough to be stored explicitly. However, if we do not find an alternating diagonal in just a few steps (indeed, π may contain none!), we need to use other diagonals. We also need to make sure that the size of each recursive subproblem is reduced by a constant

*Work on this paper by B.A. has been partially supported by NSF Grants CCF-11-17336 and CCF-12-18791. M.K. was supported in part by the ELC project (MEXT KAKENHI No. 24106008). S.P. was supported in part by the Ontario Graduate Scholarship and The Natural Sciences and Engineering Research Council of Canada.

†Polytechnic School of Engineering, New York University, Brooklyn, USA. boris.aronov@nyu.edu

‡Tohoku University, Sendai, Japan. mati@dais.is.tohoku.ac.jp

§Cheriton School of Computer Science, University of Waterloo, Canada. Simon.Pratt@uwaterloo.ca

¶National Institute of Informatics (NII), Tokyo, Japan. {andre, marcel}@nii.ac.jp

‖JST, ERATO, Kawarabayashi Large Graph Project.

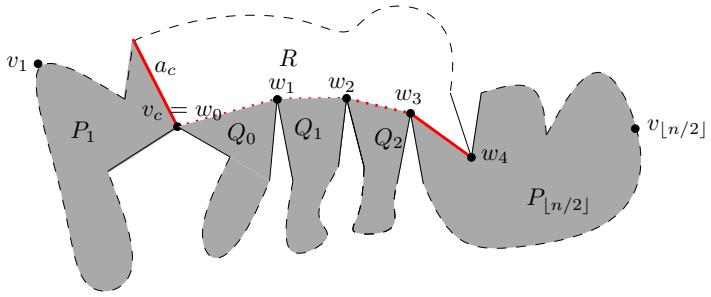

Figure 1: Partitioning P into P_1, $P_{\lfloor n/2 \rfloor}$, R, Q_1, …, Q_{k-2}. Two alternating diagonals are marked by thick red lines.

fraction, that we never exceed space bounds, and that no part of the triangulation is reported more than once.

Let v_c denote the closest to $v_{\lfloor n/2 \rfloor}$ endpoint of a_c that lies on π. Note that the subpolygon induced by a_c containing v_1 has already been triangulated. Let $w_0, \ldots, w_k$ be the portion of π up to the next alternating diagonal. That is, $\pi = (v_1, \ldots, v_c = w_0, w_1, \ldots, w_k, \ldots, v_{\lfloor n/2 \rfloor})$, where $w_1, \ldots, w_{k-1}$ are of the same type as v_c, and w_k is of different type (or $w_k = v_{\lfloor n/2 \rfloor}$ if all vertices between v_c and $v_{\lfloor n/2 \rfloor}$ are of the same type).

The path $w_0 w_1 \ldots w_k$ partitions P into subpolygons: P_1 is the subpolygon induced by a_c that does not contain $v_{\lfloor n/2 \rfloor}$. Similarly, $P_{\lfloor n/2 \rfloor}$ is the subpolygon induced by the alternating diagonal $w_{k-1}w_k$ that does not contain v_1. For any $i < k-1$, Q_i is the subpolygon induced by the non-alternating diagonal $w_i w_{i+1}$ that contains neither v_1 nor $v_{\lfloor n/2 \rfloor}$. Finally, R is the remaining component of P, see Figure 1. Note that some of these subpolygons may degenerate to a line segment (for example, when $w_i w_{i+1}$ is an edge of the boundary of P).

Lemma 1. *$R, Q_1, Q_2, \ldots, Q_{k-2}$ have at most $\lceil n/2 \rceil + k$ vertices each. If $w_k = v_{\lfloor n/2 \rfloor}$, $P_{\lfloor n/2 \rfloor}$ has at most $\lceil n/2 \rceil$ vertices.*

This result allows us to treat the easy case of our algorithm. When k is small (say, a constant number of vertices), we pause the shortest-path computation, explicitly store all vertices w_i, recursively triangulate R and Q_i, $i = 1, \ldots, k-2$), update a_c to the edge $w_{k-1}w_k$, and resume computing π.

Handling the case where k is large is more involved. Note that we do not know the value of k until we find the next alternating diagonal, but we need not compute it explicitly. We fix a parameter $\tau = s\kappa^i$, where κ is a suitably chosen absolute constant, and i is the current recursion level. We say that the distance between two consecutive alternating diagonals is *long* whenever we have encountered τ consecutive vertices of π besides v_c, all of the same type as v_c. That is, $\pi = (v_1, \ldots, v_c = w_0, w_1, \ldots, w_\tau, \ldots, v_{\lfloor n/2 \rfloor})$ and vertices $w_0, w_1, \ldots, w_\tau$ are all of the same type. In particular, the vertices $w_0, \ldots, w_\tau$ must form a convex chain (see Figure 1). Rather than continue walking along π, we identify a vertex u of P such that uw_τ is an alternating diagonal. We then partition P into $\tau - 2$ subpolygons using the diagonals $a_c, w_0w_1, w_1w_2, \ldots, w_{\tau-1}w_\tau$, and uw_τ, similarly to the easy case: P_1 is the part induced by a_c which does not contain $v_{\lfloor n/2 \rfloor}$, $P_{\lfloor n/2 \rfloor}$ is the part induced by uw_τ which does not contain v_1, Q_i is the part induced by the edge $w_i w_{i+1}$, which contains neither v_1 nor $v_{\lfloor n/2 \rfloor}$, and R is the remaining component.

Lemma 2. *There is a vertex u such that uw_τ is an alternating diagonal. It can be found in $O(n)$ time using $O(1)$ space. Moreover, $R, Q_1, Q_2, \ldots, Q_{\tau-2}$ has at most $\lceil n/2 \rceil + \tau$ vertices each.*

At a high level, our algorithm walks from v_1 to $v_{\lfloor n/2 \rfloor}$. We stop after walking τ steps or when we find an alternating diagonal (whichever comes first). This generates several subproblems of smaller complexity that are solved recursively. Once the recursion is done we update a_c to keep track of the portion of P that has been triangulated, and continue walking along π. The walking process ends when the walk reaches $v_{\lfloor n/2 \rfloor}$. In this case, in addition to triangulating R and the subpolygons Q_i as usual, we must also triangulate $P_{\lfloor n/2 \rfloor}$.

The algorithm at deeper levels of recursion is almost identical. There are only three minor changes that need to be introduced. First, we compare the size of the polygon to $\tau = s\kappa^i$ rather than s. Recall that τ denotes the amount of space available to the current instance of the algorithm. If τ is comparable to n (say, $10\tau \geq n$), then the whole polygon fits into memory and can be triangulated in linear time [4]. Otherwise, we continue with the recursive approach.

For ease in handling the subproblems, at each step we also indicate the vertex that fulfils the role of v_1 (i.e., one of the vertices from which the shortest path must be computed). Recall that we have random access to the vertices of the input. Thus, once we know which vertex takes the role of v_1, we can also find the vertex that plays the role of $v_{\lfloor n/2 \rfloor}$ in constant time as well.

Theorem 1. *Let P be a simple polygon of n vertices. A triangulation of P can be computed in $O(n^2/s)$ expected time using $O(s)$ variables, for any $s \in \Omega(\log n) \cap O(n)$.*

References

[1] B. Aronov, M. Korman, S. Pratt, A. van Renssen, and M. Roeloffzen. Time-space trade-offs for triangulating a simple polygon. *CoRR*, abs/1509.07669, 2015.

[2] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, 46(8):959–969, 2013.

[3] T. Asano and D. Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *Proc. 13th Int. Conf. Algorithms and Data Structures (WADS)*, pages 61–72, 2013.

[4] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991.

[5] S. Har-Peled. Shortest path in a polygon using sublinear space. In *Proceedings of the 31st International Symposium on Compututational Geometry (SoCG)*, pages 111–125, 2015.