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

Boris Aronov[1], Matias Korman[2], Simon Pratt[3], André van Renssen[4], and Marcel Roeloffzen[5]

1 **Tandon School of Engineering, New York University, New York, USA**
boris.aronov@nyu.edu
2 **Tohoku University, Sendai, Japan**
mati@dais.is.tohoku.ac.jp
3 **Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada**
Simon.Pratt@uwaterloo.ca
4 **National Institute of Informatics (NII), Tokyo, Japan; and JST, ERATO, Kawarabayashi Large Graph Project, Tokyo, Japan**
andre@nii.ac.jp
5 **National Institute of Informatics (NII), Tokyo, Japan; and JST, ERATO, Kawarabayashi Large Graph Project, Tokyo, Japan**
marcel@nii.ac.jp

Abstract

An s-workspace algorithm is an algorithm that has read-only access to the values of the input, write-only access to the output, and only uses $O(s)$ additional words of space. We give a randomized s-workspace algorithm for triangulating a simple polygon P of n vertices, for any $s \in O(n)$. The algorithm runs in $O(n^2/s + n(\log s)\log^5(n/s))$ expected time using $O(s)$ variables, for any $s \in O(n)$. In particular, when $s \in O(\frac{n}{\log n \log^5 \log n})$ the algorithm runs in $O(n^2/s)$ expected time.

1998 ACM Subject Classification I.3.5 Computational Geometry and Object Modeling

Keywords and phrases simple polygon, triangulation, shortest path, time-space trade-off

Digital Object Identifier 10.4230/LIPIcs.SWAT.2016.30

1 Introduction

Triangulation of a simple polygon, often used as a preprocessing step in computer graphics, is performed in a wide range of settings including on embedded systems like the Raspberry Pi or mobile phones. Such systems often run read-only filesystems for security reasons and have very limited working memory. An ideal triangulation algorithm for such an environment would allow for a trade-off in performance in time versus working space.

Computer science and specifically the field of Algorithms generally has two optimization goals; running time and memory size. In the 70's there was a strong focus on algorithms that required low memory as it was expensive. As memory became cheaper and more widely available this focus shifted towards optimizing algorithms for their running time, with memory mainly as a secondary constraint.

* Work on this paper by B. A. was supported in part by NSF Grants CCF-11-17336 and CCF-12-18791. M. K. was supported in part by the ELC project (MEXT KAKENHI No. 12H00855 and 15H02665). S. P. was supported in part by the Ontario Graduate Scholarship and The Natural Sciences and Engineering Research Council of Canada.

Nowadays, even though memory is cheap, there are other constraints that limit memory usage. First, there is a vast number of embedded devices that operate on batteries and have to remain small, which means they simply cannot contain a large memory. Second, some data may be read-only, due to hardware constraints (i.e., DVD/CDs can be written only once), or concurrency issues (i.e., to allow many processes to access the database at once).

These memory constraints can all be described in a simple way by the so-called *constrained-workspace* model (see Section 2 for details). Our input is read-only and potentially much larger than our working space, and the output we produce must we written to write-only memory. More precisely, we assume we have a read-only dataset of size n and a working space of size $O(s)$, for some user-specified parameter s. In this model, the aim is to design an algorithm whose running time decreases as s grows. Such algorithms are called *time-space trade-off* algorithms [11].

Previous Work

Several models of computation that consider space constraints have been studied in the past (we refer the interested reader to [9] for an overview). In the following we discuss the results related to triangulations. The concept of memory-constrained algorithms attracted renewed attention within the computational geometry community by the work of Asano *et al.* [4]. One of the algorithms presented in [4] was for triangulating a set of n points in the plane in $O(n^2)$ time using $O(1)$ variables. More recently, Korman *et al.* [10] introduced two different time-space trade-off algorithms for triangulating a point set: the first one computes an arbitrary triangulation in $O(n^2/s + n(\log n)\log s)$ time using $O(s)$ variables. The second is a randomized algorithm that computes the Delaunay triangulation of the given point set in expected $O((n^2/s)\log s + n(\log s)\log^* s)$ time within the same space bounds.

The above results address triangulating discrete point sets in the plane. The first algorithm for triangulating simple polygons was due to Asano *et al.* [2] (in fact, the algorithm works for slightly more general inputs: plane straight-line graphs). It runs in $O(n^2)$ time using $O(1)$ variables. The first time-space trade-off for triangulating polygons was provided by Barba *et al.* [5]. In their work, they describe a general time-space trade-off algorithm that in particular could be used to triangulate monotone polygons. An even faster algorithm (still for monotone polygons) was afterwards found by Asano and Kirkpatrick [3]: $O(n \log_s n)$ time using $O(s)$ variables. Despite extensive research on the problem, there was no known time-space trade-off algorithm for general simple polygons. It is worth noting that no lower bounds on the time-space trade-off are known for this problem either.

Results

This paper is structured as follows: In Section 2 we define our model, as well as the problems we study. Our main result on triangulating a simple polygon P with n vertices using only a limited amount of memory can be found in Section 3. Our algorithm achieves expected running time of $O(n^2/s + n(\log s)\log^5(n/s))$ using $O(s)$ variables, for any $s \in \Omega(\log n) \cap O(n)$. Note that for most values of s (i.e., when $s \in O(\frac{n}{(\log n)\log^5 \log n}))$ the algorithm runs in $O(n^2/s)$ expected time.

Our approach uses a recent result by Har-Peled [8] as a tool for subdividing P into smaller pieces and solving them recursively. A similar approach can be used for other problems. Indeed, in an extended version of this paper [1] we show how a similar approach can be used to compute the *shortest-path map* or *shortest-path tree* from any point $p \in P$, or simply to split P by $\Theta(s)$ pairwise disjoint diagonals into smaller subpolygons, each with $\Theta(n/s)$ vertices.

2 Preliminaries

In this paper, we utilize the *s-workspace* model of computation that is frequently used in the literature (see for example [2, 5, 6, 8]). In this model the input data is given in a read-only array or some similar structure. In our case, the input is a simple polygon P; let $v_1, v_2, \ldots, v_n$ be the vertices of P in clockwise order along the boundary of P. 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 passing through vertices v_i and v_j and the horizontal line passing through v_k) can be performed in constant time.

In addition to the read-only data, an s-workspace algorithm can use $O(s)$ variables during its execution, for some parameter s determined by the user. Implicit memory consumption (such as the stack space needed in recursive algorithms) must be taken into account when determining the size of a workspace. We assume that each variable or pointer is stored in a data word of $\Theta(\log n)$ bits. Thus, equivalently, we can say that an s-workspace algorithm uses $O(s \log n)$ bits of storage.

In this model we study the problem of computing a triangulation of a simple polygon P. 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. Once an output value is reported it cannot be afterwards accessed or modified.

In other memory-constrained triangulation algorithms [2, 3] the output is reported as a list of edges in no particular order (with no information on neighboring edges or faces). Moreover, it is not clear how to modify these algorithms to obtain such information. Our approach has the advantage that, in addition to the list of edges, we can report adjacency information as well. For example, we could report the triangulation in a doubly connected edge list (or any other similar format). More details on how we can report the triangulation are given in Section 3.4.

A vertex of a polygon is *reflex* if its interior angle is larger than 180°. Given two points $p, q \in P$, the *geodesic* (or *shortest path*) between them is the path of minimum length that connects p and q and that stays within P (viewing P as a closed set). The length of that path is the *geodesic distance* from p to q. It is well known that, for any two points of P, their geodesic π always exists and is unique. Such a path is a polygonal chain whose vertices (other than p and q) are reflex vertices of P. Thus, we often identify π with the ordered sequence of reflex vertices traversed by the path from p to q. When that sequence is empty (i.e., the geodesic consists of the straight segment pq) we say that p *sees* q (and vice versa).

Our algorithm relies in a recent result by Har-Peled [8] for computing geodesics under memory constraints. Specifically, it computes the geodesic between any two points in a simple polygon of n vertices in expected $O(n^2/s + n \log s \log^4(n/s))$ time using $O(s)$ words of space. Note that this path might not fit in memory, so the edges of the geodesic are reported one by one in order.

3 Algorithm

Let π be the geodesic connecting v_1 and $v_{\lfloor n/2 \rfloor}$. From a high-level perspective, the algorithm uses the approach of Har-Peled [8] to compute π. We will use the computed edges to subdivide P into smaller problems that can be solved recursively.

We start by introducing some definitions that will help in storing which portion of the polygon has already been triangulated. 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 $\lfloor n/2 \rfloor < i \leq n$. Top/bottom is the *type* of a vertex and all vertices (except for v_1 and $v_{\lfloor n/2 \rfloor}$) have exactly one type. A diagonal c is *alternating* if it connects a top and a bottom vertex (or one of its endpoints is either v_1 or $v_{\lfloor n/2 \rfloor}$), and *non-alternating* otherwise.

We will use diagonals to partition P into two parts. For simplicity of the exposition, given a diagonal d, we regard both components of $P \setminus d$ as closed (i.e., the diagonal belongs to both of them). Since any two consecutive vertices of P can see each other, the partition an edge of P is trivial, in the sense that one subpolygon is P and the other one is a line segment.

▶ **Observation 1.** *Let c be a diagonal of P not incident to v_1 or $v_{\lfloor n/2 \rfloor}$. Vertices 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 2.** *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, an alternating diagonal a_c records the part of the polygon has already been triangulated. More specically, we maintain the following invariant: the connected component of $P \setminus a_c$ not containing $v_{\lfloor n/2 \rfloor}$ has already been triangulated.

Ideally, a_c would be a segment of π (the geodesic connecting v_1 and $v_{\lfloor n/2 \rfloor}$), but this is not always possible. Instead, we guarantee that at least one of the endpoints of a_c is a vertex of π that has already been computed in the execution of 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 computation approach of Har-Peled. Our aim is to walk along π until we find a new alternating diagonal a_{new}. At that moment we pause the execution of the shortest-path algorithm, triangulate the subpolygons of P that have been created (and contain neither v_1 nor $v_{\lfloor n/2 \rfloor}$) recursively, update a_c to the newly found alternating diagonal, and then continue with the execution of the shortest-path algorithm.

Although our approach is intuitively simple, there are several technical difficulties that must be carefully considered. Ideally, the number of vertices we walked along π before finding an alternating diagonal is small and thus they can be stored explicitly. But if we do not find an alternating diagonal on π in just a few steps (indeed, it could even be that there is no alternating diagonal in π), we need to use other diagonals. We also need to make sure that the complexity of each recursive subproblem is reduced by a constant fraction, that we never exceed space bounds, and that no part of the triangulation is reported more than once.

Let v_c denote the endpoint of a_c that is on π and that is closest to $v_{\lfloor n/2 \rfloor}$. Recall that the subpolygon defined 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, path π is of the form $\pi = (v_1, \ldots, v_c = w_0, w_1, \ldots, w_k, \ldots, v_{\lfloor n/2 \rfloor})$ 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).

Consider the partition of P induced by a_c and this portion of π, see Figure 1. Let P_1 be the subpolygon induced by a_c that does not contain $v_{\lfloor n/2 \rfloor}$. Similarly, let $P_{\lfloor n/2 \rfloor}$ be the subpolygon that is induced by the alternating diagonal $w_{k-1}w_k$ and does not contain v_1[1].

[1] For simplicity of the exposition, the definition of P_1 assumes that $v_{\lfloor n/2 \rfloor}$ is not an endpoint of a_c (similarly, v_1 not an endpoint of $w_{k-1}w_k$ for the definition of $P_{\lfloor n/2 \rfloor}$). Each of these conditions is not satisfied once (i.e., with the first and last diagonals of π), and in those cases the polygons P_1 and $P_{\lfloor n/2 \rfloor}$ are not properly defined. Whenever this happens we have $k = 1$ and a single diagonal that splits P in

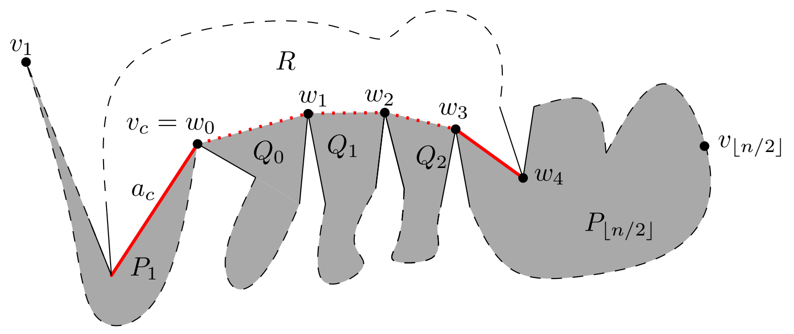

Figure 1 Partitioning P into subpolygons P_1, $P_{\lfloor n/2 \rfloor}$, R, Q_1, …, Q_{k-2}. The two alternating diagonals are marked by thick red lines.

For any $i < k - 1$ we define Q_i as 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, let R be the remaining component of P. Note that some of these subpolygons may be degenerate and consist only of a line segment (for example, when $w_i w_{i+1}$ is an edge of P).

▶ **Lemma 3.** *Each of the subpolygons R, Q_1, Q_2, …, Q_{k-2} has at most $\lceil n/2 \rceil + k$ vertices. Moreover, if $w_k = v_{\lfloor n/2 \rfloor}$, then the subpolygon $P_{\lfloor n/2 \rfloor}$ also has at most $\lceil n/2 \rceil$ vertices.*

Proof. Subpolygons Q_i are induced by non-alternating diagonals and cannot have more than $\lceil n/2 \rceil$ vertices, by Corollary 2. The proof for R follows by definition: the boundary of R (other than vertices $w_0, \ldots, w_k$) is defined by a contiguous portion of P consisting of only top vertices or only bottom vertices. Recall that there are at most $\lceil n/2 \rceil$ of them. Similarly, if $w_k = v_{\lfloor n/2 \rfloor}$, subpolygon $P_{\lfloor n/2 \rfloor}$ can only have vertices of one type (either only top or only bottom vertices), and thus the bound holds. This completes the proof of the Lemma. ◀

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

Handling the case of large k 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 directly. Given a parameter τ related to the workspace allowed for our algorithms, we say that the path is *long* when $k > \tau$. Initially we set $\tau = s$ but the value of this parameter will change as we descend the recursion tree. We say that the distance between two alternating diagonals is *long* whenever we have computed τ vertices of π besides v_c and they are all of the same type as v_c. That is, path π is of the form $\pi = (v_1, \ldots, v_c = w_0, w_1, \ldots, w_\tau, \ldots v_{\lfloor n/2 \rfloor})$ 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 look for a vertex u of P that together with w_τ forms an alternating diagonal. Once we have found this diagonal, we have at most $\tau + 2$ diagonals ($a_c, w_0 w_1, w_1 w_2, \ldots, w_{\tau-1} w_\tau$, and $u w_\tau$) partitioning P into at most $\tau + 3$ subpolygons once again: 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 $u w_\tau$ which does not contain v_1, Q_i is the part induced by $w_i w_{i+1}$, which contains neither v_1 nor $v_{\lfloor n/2 \rfloor}$, and R is the remaining component.

two. Thus, if $v_{\lfloor n/2 \rfloor} \in a_c$ (and thus P_1 is undefined), we simply define P_1 as the complement $P_{\lfloor n/2 \rfloor}$ (similarly, if $v_1 \in w_{k-1} w_k$, we define $P_{\lfloor n/2 \rfloor}$ as complement of P_1. If both subpolygons are undefined simultaneously we assign them arbitrarily.

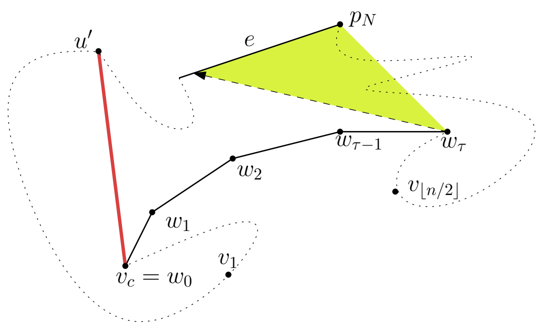

Figure 2 After we have walked τ steps of π we can find an alternating diagonal by shooting a ray from w_τ either towards u' or $w_{\tau-1}$ (whichever is higher). The upper endpoint p_N of the first edge e hit might not be visible. But in this case the reflex vertex of smallest angle inside the triangular zone must be visible.

▶ **Lemma 4.** *We can find a vertex u that together with w_τ forms an alternating diagonal in $O(n)$ time using $O(1)$ space. Moreover, each of the subpolygons R, Q_1, Q_2, …, $Q_{\tau-2}$ has at most $\lceil n/2 \rceil + \tau$ vertices.*

Proof. Proofs for the size of the subpolygons are identical to those of Lemma 3. Thus, we focus on how to compute u efficiently. Without loss of generality, we may assume that the edge $w_{\tau-1}w_\tau$ is horizontal. Recall that the chain $w_0, \ldots, w_\tau$ is in convex position, thus all of these vertices must lie on one side of the line $\ell_{\tau,\tau-1}$ through w_τ and $w_{\tau-1}$. Without loss of generality, we may assume that they all lie below $\ell_{\tau,\tau-1}$. Let u' be the endpoint of a_c other than v_c. If u' also lies below $\ell_{\tau,\tau-1}$, we shoot a ray from w_τ towards $w_{\tau-1}$. Otherwise, we shoot a ray from w_τ towards u'. Let e be the first edge that is properly intersected by the ray and let p_N be the endpoint of e of highest y-coordinate. Observe that p_N must be on or above $\ell_{\tau,\tau-1}$, see Figure 2.

Ideally, we would like to report p_N as the vertex u. However, point p_N need not be visible even when some portion of e is. Whenever this happens we can use the visibility properties of simple polygons: since e is partially visible, we know that the portion of P that obstructs visibility between w_τ and p_N must cross the segment from w_τ to p_N. In particular, there must be one or more reflex vertices in the triangle formed by w_τ, p_N, and the visible point of e (shaded region of Figure 2). Among those vertices, we know that the vertex r that maximizes the angle $\angle p_N w_\tau r$ must be visible (see Lemma 1 of [6]). Further note that r must be a top vertex: otherwise π would need to traverse through r to reach $v_{\lfloor n/2 \rfloor}$, and this would force π to do a reflex turn, which is impossible in a geodesic.

As described in Lemma 1 of [6], in order to find such a reflex vertex we need to scan the input polygon at most three times, each time storing a constant amount of information: once for finding the edge e and point p_N, once more to determine if p_N is visible, and a third time to find r if p_N is not visible. ◀

At 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 it reaches $v_{\lfloor n/2 \rfloor}$. In this case, in addition to triangulating R and the Q_i subpolygons as usual, we must also triangulate $P_{\lfloor n/2 \rfloor}$.

The algorithm in the deeper levels of recursion is almost identical. There are only three minor changes that need to be introduced. We need some base cases to end the recursion. Recall that τ denotes the amount of space available to the current level of recursion. Thus, 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 [7]. Similarly, if τ is small (say $\tau \leq 1$, we have run out of space and thus we triangulate P using a constant workspace algorithm [2]. In all other cases we continue with the recursive algorithm as usual.

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 geodesic must be computed). Recall that we have random access to the vertices of the input. Thus, once we know which vertex plays the role of v_1, we can find the vertex that satisfies the role of $v_{\lfloor n/2 \rfloor}$ in constant time as well.

In order to avoid exceeding the space bounds, at each level of the recursion we decrease the value of τ by a factor of $\kappa < 1$. The exact value of κ will be determined below. Pseudocode of the recursive algorithm can be found in the Appendix (Algorithm 1).

▶ **Theorem 5.** *Let P be a simple polygon of n vertices. We can compute a triangulation of P in $O(n^2/s + n(\log s)\log^5(n/s))$ expected time using $O(s)$ variables (for any $s \in O(n)$). In particular, when $s \in O(\frac{n}{\log n \log^5 \log n})$ the algorithm runs in $O(n^2/s)$ expected time.*

In the remainder of the section we prove correctness and both the time and space bounds for our algorithm.

3.1 Correctness

The current diagonal a_c properly records what portion of the polygon has already been triangulated. Thus, we never report an edge of the triangulation more than once. Hence, in order to show correctness of the algorithm, we must show that the recursion eventually terminates.

During the execution of the algorithm, we invoke recursion for polygons Q_i, R, and $P_{\lfloor n/2 \rfloor}$ (the latter one only when we have reached $v_{\lfloor n/2 \rfloor}$). By Lemma 3 all of these polygons have size at most $n/2 + \tau$. Since we only enter this level of recursion whenever $\tau \leq n/10$ (see lines 1-3 of Algorithm 1), overall the size of the problem decreases by a factor of $6/10$. That is, at each level of recursion the problem instances are reduced by a constant fraction. In particular, after $O(\log n)$ steps the subpolygons will be of constant size and will be solved without recursion.

At each level of recursion we use the shortest-path algorithm of Har-Peled. This algorithm needs random access in constant time to the vertices of the polygon. Thus, we must make sure that this property is preserved at all levels of recursion. A simple way to do so would be to explicitly store the polygon in memory at every recursive call, but this may exceed the space bounds of the algorithm.

Instead, we make sure that the subpolygon is described by $O(\tau)$ words. By construction, each subpolygon consists of a single chain of contiguous input vertices of P and at most τ additional *cut* vertices (vertices from the geodesics at higher levels). We can represent the portion of P by the indices of the first and last vertex of the chain and explicitly store the indices of all cut vertices. By an appropriate renaming of the indices within the subpolygon, we can make the vertices of the chain appear first, followed by the cut vertices. Thus, when we need to access the ith vertex of the subpolygon, we can check if i corresponds to a vertex of the chain or one of the cut vertices and identify the desired vertex in constant time, in either case.

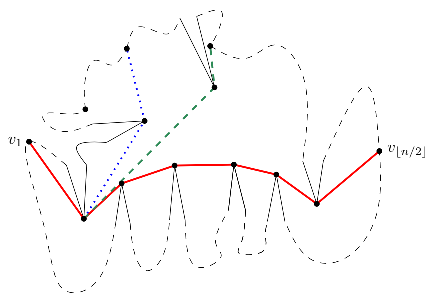

Figure 3 At different level of recursion the subproblems are formed by a consecutive chain of the input and a list of $O(s)$ cut vertices. The geodesics used to split the problem at first, second and third level are depicted in solid red, dashed green, and dotted blue, respectively.

Now, we must show that each recursive call satisfies this property. Clearly this holds for the top level of recursion, where the input polygon is simply P and no cut vertices are needed. At the next level of recursion each subproblem has up to τ cut vertices and a chain of contiguous input vertices. The way we make sure that this property is satisfied at lower levels of recursion is by a correct choice of v_1 (the vertex from which we start the path): at each level of recursion we build the next geodesic starting from either the first or last cut vertex. This might create additional cut vertices, but their position is immediately after or before the already existing cut vertices (see Figure 3). This way we certify that random access to the input polygon is possible at all levels of recursion.

3.2 Time Bounds

We use a two-parameter function $T(\eta, \tau)$ to bound the expected running time of the algorithm at all levels of recursion. The first parameter η represents the size of the problem. Specifically, for a polygon of n vertices we set $\eta = n - 2$, namely, the number of triangles to be reported. The second parameter τ gives the space bound for the algorithm. Initially, we have $\tau = s$, but this value decreases by a factor of κ at each level of recursion. Recall that τ is also the workspace limit for the shortest-path algorithm of Har-Peled that we invoke as part of our algorithm. In addition, τ is also used as the limit on the length of the geodesic we explore looking for an alternating diagonal.

When τ becomes really small (say $\tau \leq 10$) we have run out of allotted space. Thus, we triangulate the polygon using the constant workspace method of Asano *et al.* [2] that runs in $O(n^2)$ time. Similarly, if the space is large when compared to the instance size (say, $10\tau \geq \eta$) the polygon fits in the allowed workspace, hence we use Chazelle's algorithm [7] for triangulating it. In both cases we have $T(\eta, \tau) \leq c_\Delta \eta^2/\tau$ (for some constant $c_\Delta > 0$).

In other situations, we must partition the problem and solve it recursively. First we bound the time needed to compute the partitions. The main tool we use is computing the geodesic between v_1 and $v_{\lfloor n/2 \rfloor}$. This is done by the algorithm of Har-Peled [8] which takes $O(\eta^2/\tau + \eta(\log \tau)\log^4(\eta/\tau))$ expected time and uses $O(\tau)$ space. Recall that we pause and continue it often during the execution of our algorithm, but overall we only execute it once. Thus, the total time spent in shortest-path computation at one level is unchanged.

Another operation that we execute is FindAlternatingDiagonal (i.e., Lemma 4) which takes $O(\eta)$ time and $O(1)$ space. In the worst case, this operation is invoked once for every τ vertices of π. Since π cannot have more than n vertices, the overall time spent in this operation is bounded by $O(\eta^2/\tau)$. Thus, ignoring the time spent in recursion, the expected running time of the algorithm is $c_{\mathrm{HP}}(\eta^2/\tau + \eta(\log \tau)\log^4(\eta/\tau))$ for some constant

c_{HP}, which without loss of generality we assume to be at least c_Δ.

That is, for any value of η and τ we never spend more than $c_{\mathrm{HP}}(\eta^2/\tau + \eta(\log\tau)\log^4(\eta/\tau))$ time. To this value we must add the time spent in recursion. At each level we launch several subproblems, giving a recurrence of the form

$$T(\eta,\tau) \leq c_{\mathrm{HP}}(\eta^2/\tau + \eta(\log\tau)\log^4(\eta/\tau)) + \sum_j T(\eta_j, \kappa\tau).$$

Recall that the values η_j cannot be very large when compared to η. Indeed, each subproblem can have at most a constant fraction c of vertices of the original one (i.e., the way in which lines 1–4 of Algorithm 1 have been set, we have $c = 6/10$). Thus, each η_j satisfies $\eta_j \leq c(\eta+2) - 2 \leq c\eta$. Since every edge is reported exactly once, we also have $\sum_j \eta_j = \eta$.

We claim that for any $\tau, \eta > 0$ there exists a constant c_R so that $T(\eta,\tau) \leq c_R(\eta^2/\tau + \eta(\log\tau)\log^5(\eta/\tau))$. Indeed, when $\tau \leq 10$ or $10\tau \geq \eta$ we have $T(\eta,\tau) \leq c_\Delta\eta^2 \leq c_{\mathrm{HP}}\eta^2$. Otherwise, we use induction and obtain

$$\begin{aligned}
T(\eta,\tau) &\leq c_{\mathrm{HP}}(\eta^2/\tau + \eta(\log\tau)\log^4(\eta/\tau)) + \sum_j T(\eta_j,\tau\kappa)\\
&\leq c_{\mathrm{HP}}(\eta^2/\tau + \eta(\log\tau)\log^4(\eta/\tau)) + \frac{c_R}{\tau\kappa}\sum_j \eta_j^2 + c_R\sum_j \eta_j(\log\kappa\tau)\log^5(\frac{\eta_j}{\tau\kappa})\\
&\leq (c_{\mathrm{HP}}\frac{\eta^2}{\tau} + \frac{c_R}{\tau\kappa}\sum_j \eta_j^2) + c_{\mathrm{HP}}\eta(\log\tau)\log^4(\eta/\tau) + c_R\sum_j \eta_j(\log\tau)\log^5(\frac{\eta_j}{\tau\kappa})\\
&\leq (c_{\mathrm{HP}}\frac{\eta^2}{\tau} + \frac{c_R}{\tau\kappa}\sum_j \eta_j^2) + c_{\mathrm{HP}}\eta(\log\tau)\log^4(\eta/\tau) + c_R\sum_j \eta_j(\log\tau)\log^5(\frac{c\eta}{\tau\kappa})\\
&\leq (c_{\mathrm{HP}}\frac{\eta^2}{\tau} + \frac{c_R}{\tau\kappa}\sum_j \eta_j^2) + c_{\mathrm{HP}}\eta(\log\tau)\log^4(\eta/\tau) + c_R\eta(\log\tau)\log^5(\frac{c\eta}{\tau\kappa}).
\end{aligned}$$

The sum $\sum_j \eta_j^2$ is at most $\frac{n}{cn}(c\eta)^2 = c\eta^2$, since $\eta_j \leq c\eta$, yielding

$$\begin{aligned}
T(\eta,\tau) &\leq (c_{\mathrm{HP}}\frac{\eta^2}{\tau} + \frac{c_R c}{\kappa}\frac{\eta^2}{\tau}) + c_{\mathrm{HP}}\eta(\log\tau)\log^4(\eta/\tau) + c_R\eta(\log\tau)\log^5(\frac{c\eta}{\tau\kappa})\\
&\leq \frac{c_R\eta^2}{\tau} + c_{\mathrm{HP}}\eta(\log\tau)\log^4(\eta/\tau) + c_R\eta(\log\tau)\log^5(\frac{c\eta}{\tau\kappa}),
\end{aligned}$$

where the inequality $c_{\mathrm{HP}} + \frac{c}{\kappa}c_R \leq c_R$ holds for sufficiently large values of c_R and $\kappa < 1$ (say, $c_R = 10c_{\mathrm{HP}}$ and $\kappa = 9/10$). Now we focus on the second term of the inequation. We upper bound $\log^5(\frac{c\eta}{\tau\kappa})$ by $\log^4(\frac{\eta}{\tau})\log(\frac{c\eta}{\tau\kappa}) = \log^4(\frac{\eta}{\tau})(\log(\frac{\eta}{\tau}) - \log(\frac{\kappa}{c}))$ and substitute to obtain:

$$\begin{aligned}
T(\eta,\tau) &\leq \frac{c_R\eta^2}{\tau} + c_{\mathrm{HP}}\eta(\log\tau)\log^4(\eta/\tau) + c_R\eta(\log\tau)\log^4(\frac{\eta}{\tau})(\log(\frac{\eta}{\tau}) - \log(\frac{\kappa}{c}))\\
&\leq \frac{c_R\eta^2}{\tau} + (\eta(\log\tau)\log^4(\frac{\eta}{\tau}))(c_{\mathrm{HP}} + c_R\log(\frac{\eta}{\tau}) - c_R\log(\frac{\kappa}{c}))\\
&\leq \frac{c_R\eta^2}{\tau} + c_R(\eta(\log\tau)\log^5(\frac{\eta}{\tau})) = c_R(\eta^2/\tau + \eta(\log\tau)\log^5(\frac{\eta}{\tau})).
\end{aligned}$$

Again, the $c_{\mathrm{HP}} - c_R\log(\frac{\kappa}{c}) \leq 0$ inequality holds for sufficiently large values of c_R, that depend on c_{HP}, κ and c.

3.3 Space Bounds

We now show that the space bound holds. Recall that we stop recursion whenever the problem instance fits into memory or $\tau \leq 1$. Since the value of τ decreases by a constant factor at each level of recursion, we will never recurse for more than $\log_\kappa s = O(\log s)$ levels. Thus, the implicit memory consumption used in recursion does not exceed the space bounds.

Now we bound the size of the workspace needed by the algorithm at level i of the recursion (with the main algorithm invocation being level 0) by $O(s \cdot \kappa^i)$. Indeed, this is the threshold of space we receive as input (recall that initially we set $\tau = s$ and that at each level we reduce this value by a factor of κ). This threshold value is the amount of space for the shortest-path computation algorithm invoked at the current level, as well as limit on the number of vertices of π that are stored explicitly before invoking procedure FindAlternatingDiagional. Once we have found the new alternating diagonal, the vertices of π that were stored explicitly are used to generate the subproblems for the recursive calls.

The space used for storing the intermediate points can be reused after the recursive executions are finished, so overall we conclude that at the i-th level of recursion the algorithm never uses more than $O(s \cdot \kappa^i)$ space. Since we never have two simultaneously executing recursive calls at the same level, and $\kappa < 1$, the total amount of space used in the execution of the algorithm is bounded by

$$O(s) + O(s \cdot \kappa) + O(s \cdot \kappa^2) + \ldots = O(s).$$

3.4 Considerations on the Output

For simplicity in the explanation we assumed that in order to report the triangulation, reporting the edges suffices. However, we note that we can also report the triangulation in any other format, such as a list of adjacencies. That is, we can report the triangles generated, and for each one we additionally report the three boundary edges and the three triangles adjacent to it). Most of this is easy to do, since the triangles are reported at the bottom level of the recursion where the subpolygons fit in memory. Thus, for each triangle we can report their adjacencies as usual (using for example the indices of the vertices to identify the triangles). The only difficulty arises around the edges used to split the polygon into subpolygons. When we create the triangle on one side of such an edge we do not yet know which triangle will be created on the other side as this triangle resides in a different subpolygon. Hence, this triangle cannot report its adjacencies yet.

Instead we delay reporting triangles along these splitting edges until both triangles have been constructed. For this purpose we must slightly alter the triangulation invariant associated to a_c: subpolygon P_1 has been triangulated and all triangles have been reported *except* the triangle whose boundary is a_c. This triangle (along with its two neighbors in P_1) is stored explicitly in memory.

The algorithm proceeds, partitioning into subproblems $Q_1, \ldots Q_{k-2}$ and R as usual. Each subproblem Q_i returns a triangle that has not been reported yet along with its two adjacencies (or nothing if the corresponding subpolygon Q_i is empty). The neighbors of these triangles are in the subproblem R, so they are given to the recursive procedure of R. As soon as the missing neighbor is computed, we can report the stored triangle delete it from memory. Once R has finished we need to update a_c and v_c as usual. In addition, we must now store (and do not yet report) the triangle that is adjacent to a_c. The bottommost level of recursion triangulates as usual and stores the single triangle that has not been reported so it can be reported when processing R.

Overall, at each level of recursion we need to store as many triangles as subproblems generated. Moreover, once R has been recursively triangulated, this information need not be stored anymore. Recall that the number of subproblems generated is at most the space threshold. Thus, we conclude that the storage bounds are asymptotically unaffected.

Acknowledgements. The authors would like to thank Jean-François Baffier, Man-Kwun Chiu, Wolfgang Mulzer and Takeshi Tokuyama for valuable discussion in the creation of this paper.

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. URL: `http://arxiv.org/abs/1509.07669`.

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

5 L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira. Space–time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015. `doi:10.1007/s00453-014-9893-5`.

6 L. Barba, M. Korman, S. Langerman, and R. I. Silveira. Computing the visibility polygon using few variables. *Computational Geometry: Theory and Applications*, 47(9):918–926, 2013.

7 B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991. `doi:10.1007/BF02574703`.

8 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. `doi:10.4230/LIPIcs.SOCG.2015.111`.

9 M. Korman. Memory-constrained algorithms. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–7. Springer Berlin Heidelberg, 2015. `doi:10.1007/978-3-642-27848-8_586-1`.

10 M. Korman, W. Mulzer, M. Roeloffzen, A. v. Renssen, P. Seiferth, and Y. Stein. Time-space trade-offs for triangulations and voronoi diagrams. In *Proc. 14th Int. Conf. Algorithms and Data Structures (WADS)*, pages 482–494, 2015.

11 J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

A Algorithm Pseudocode

Algorithm 1: Pseudocode for Triangulate(P, v_1, τ) that, given a simple polygon P, a vertex v of P, and workspace capacity τ, computes a triangulation of P in $O(n^2/\tau)$ time using $O(\tau)$ variables.

```
 1: if $10\tau \geq n$ then (* The polygon fits into memory. *)
 2:     Triangulate $P$ using Chazelle's algorithm [7]
 3: else if $\tau \leq 10$ then (* We ran out of recursion space. *)
 4:     Triangulate $P$ using the constant workspace algorithm [2]
 5: else (* $P$ is large, we will use recursion. *)
 6:     $a_c \leftarrow v_1 v_1$
 7:     $v_c \leftarrow v_1$
 8:     walked $\leftarrow v_1$ (* Variable to keep track of how far we have walked on $\pi$. *)
 9:     while walked $\neq v_{\lfloor n/2 \rfloor}$ do
10:         $i \leftarrow 0$ (* $i$ counts the number of steps before finding an alternation edge *)
11:         repeat
12:             $i \leftarrow i + 1$
13:             $w_i \leftarrow$ next vertex of $\pi$
14:         until $i = \tau$ or type($v_c$) $\neq$ type($w_i$)
15:         if type($v_c$) $\neq$ type($w_i$) then
16:             $u' \leftarrow w_{i-1}$
17:             $a_{\text{new}} \leftarrow w_i w_{i-1}$
18:         else (* We walked too much. Use Lemma 4 to partition the problem. *)
19:             $u' \leftarrow$ FindAlternatingDiagonal($P, a_c, v_c, w_1, \ldots, w_\tau$)
20:             $a_{\text{new}} \leftarrow u' w_i$
21:         end if
22:         (* Now we triangulate the subpolygons. *)
23:         Triangulate($R, u', \tau \cdot \kappa$)
24:         for j=0 to i-2 do
25:             Triangulate($Q_j, w_j, \tau \cdot \kappa$)
26:         end for
27:         $a_c \leftarrow a_{\text{new}}$
28:         $v_c \leftarrow w_\tau$
29:         walked $\leftarrow w_\tau$
30:     end while
31:     (* We reached $v_{\lfloor n/2 \rfloor}$. All parts except $P_{\lfloor n/2 \rfloor}$ have been triangulated. *)
32:     Triangulate($P_{\lfloor n/2 \rfloor}, w_i, \tau \cdot \kappa$)
33: end if
```