Shortest Path in a Polygon using Sublinear Space*

Sariel Har-Peled

Department of Computer Science, University of Illinois
201 N. Goodwin Avenue, Urbana, IL, 61801, USA
sariel@illinois.edu

Abstract

We resolve an open problem due to Tetsuo Asano, showing how to compute the shortest path in a polygon, given in a read only memory, using sublinear space and subquadratic time. Specifically, given a simple polygon P with n vertices in a read only memory, and additional working memory of size m, the new algorithm computes the shortest path (in P) in $O(n^2/m)$ expected time, assuming $m = O(n/\log^2 n)$. This requires several new tools, which we believe to be of independent interest.

Specifically, we show that violator space problems, an abstraction of low dimensional linear-programming (and LP-type problems), can be solved using constant space and expected linear time, by modifying Seidel's linear programming algorithm and using pseudo-random sequences.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, I.1.2 Algorithm, I.3.5 Computational Geometry and Object Modeling

Keywords and phrases Shortest path, violator spaces, limited space.

Digital Object Identifier 10.4230/LIPIcs.SOCG.2015.111

1 Introduction

Space might not be the final frontier in the design of algorithms but it is an important constraint. Of special interest are algorithms that use sublinear space. Such algorithms arise naturally in streaming settings, or when the data set is massive, and only a few passes on the data are desirable. Another such setting is when one has a relatively weak embedded processor with limited high quality memory. For example, flash memory can withstand around 100,000 rewrites before starting to deteriorate. Specifically, imagine a hybrid system having a relatively large flash memory, with significantly smaller RAM. That is to a limited extent the setting in a typical smart-phone[1].

The model. The input is provided in a read only memory, and it is of size n. We have $O(m)$ available space which is a read/write space (i.e., the *work space*). We assume, as usual, that every memory cell is a word, and such a word is large enough to store a number or a pointer. We also assume that the input is given in a reasonable representation[2]. A survey of this computational model and related work is provided in the introduction of Asano *et al.* [1, 2].

The problem. We are given a simple polygon P with n vertices in the plane, and two points $s, t \in \mathsf{P}$ – all provided in a read-only memory. We also have $O(m)$ additional read-write memory (i.e., work space). The task is to compute the shortest path from s to t inside P.

* Work on this paper was partially supported by a NSF AF awards CCF-1421231, and CCF-1217462.

[1] For example, a typical smart-phone in 2014 have 2GB of RAM and 16GB of flash memory. I am sure these numbers would be laughable in a few years. So it goes.

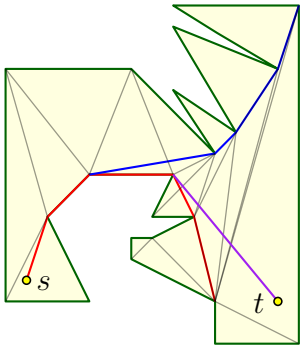
[2] In some rare cases, the "right" input representation can lead directly to sublinear time algorithms. See the work by Chazelle *et al.* [5].

31st International Symposium on Computational Geometry (SoCG'15).
Editors: Lars Arge and János Pach; pp. 111–125

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Asano *et al.* [1] showed how to solve this problem, in $O(n^2/m)$ time, using $O(m)$ space. The catch is that their solution requires quadratic time preprocessing. In a talk by Tetsuo Asano, given in a workshop in honor of his 65th birthday (during SoCG 2014), he posed the open problem of whether this quadratic preprocessing penalty can be avoided. This work provides a positive answer to this question.

If linear space is available. The standard algorithm [17] for computing the shortest path in a polygon, triangulates the polygon, (conceptually) computes the dual graph of the triangulation, which yields a tree, with a unique path between the triangles that contains the source s, and the target t. This path specifies the sequence of diagonals crossed by the shortest path, and it is now relatively easy to walk through this sequence of triangles and maintain the shortest paths from the source to the two endpoints of each diagonal. These paths share a prefix path, and then diverge into two concave chains (known together as a *funnel*). Once arriving to the destination, one computes the unique tangent from the destination t to one of these chains, and the (unique) path, formed by the prefix together with the tangent, defines the shortest path, which can be now extracted in linear time. See figure on the right for illustration.



Sketch of the new algorithm. The basic idea is to decompose the polygon into bigger pieces than triangles. Specifically, we break the polygon into canonical pieces each of size $O(m)$. To this end, we break the given polygon P into $\lceil n/m \rceil$ polygonal chains, each with at most m edges. We refer to such a chain as a *curve*. We next use the notion of *corridor decomposition*, introduced[3] by the author [14], to (conceptually) decompose the polygon into canonical pieces (i.e., corridors). Oversimplifying somewhat, each corridor is a polygon having portions of two of the input curves as floor and ceiling, and additional two diagonals of P as gates. It is relatively easy, using constant space and linear time, to figure out for such a diagonal if it separates the source from the destination. Now, start from the corridor containing the source, and figure out which of its two gates the shortest path goes through. We follow this gate to the next corridor and continue in this fashion till we reach the destination. Assuming that computing the next piece can be done in roughly linear time, this algorithm solves the shortest path problem in $O(n^2/m)$ time, as walking through a piece takes (roughly) linear time, and there are $O(n/m)$ pieces the shortest path might go through. (One also needs to keep track of the funnel being constructed during this walk, and prune parts of it away because of space considerations.)

Point-location queries in a canonical decomposition. To implement the above, we need a way to perform a point-location query in the corridor decomposition, without computing it explicitly. Here we are interested in any canonical decomposition that partition the underlying space into cells. Such a partition is induced by a set of objects, and every cell

[3] Many somewhat similar decomposition ideas can be found in the literature (for example, the decomposition of a polygon into monotone polygons so that the pieces, and thus the original polygon, can be triangulated [3]). Nevertheless, this specific decomposition scheme [14] is right for our nefarious purposes, but the author would not be surprised if it was known before. Well, at least this footnote is new!

is defined by a constant number of objects. Standard examples of such partitions are (i) vertical decomposition of segments in the plane, or (ii) bottom vertex triangulation of the Voronoi diagram of points in $\mathbb{R}^3$. Roughly speaking, any partition that complies with the Clarkson-Shor framework [8] is such a canonical decomposition.

If space and time were not a constraint, we could build the decomposition explicitly, Then a standard point-location query in the history DAG would yield the desired cell. Alternatively, one can perform this point-location query in the history DAG implicitly, without building the DAG before hand, but it is not obvious how to do so with limited space. Surprisingly, at least for the author, this task can be solved using techniques related to low-dimensional linear programming.

Violator spaces. Low dimensional linear programming can be solved in linear time [18]. Sharir and Welzl [23] introduced *LP-type* problems, which are an extension of linear programming. Intuitively, but somewhat incorrectly, one can think about LP-Type algorithms as solving low-dimensional convex programming, although Sharir and Welzl [23] used it to decide in linear time if a set of axis-parallel rectangles can be stabbed by three points (this is quite surprising as this problem has no convex programming flavor). LP-type problems have the same notions as linear programming of bases, and an objective function. The function scores such bases, and the purpose is to find the basis that does not violate any constraint and minimizes (or maximizes) this objective. A natural question is how to solve such problems if there is no scoring function of the bases.

This is captured by the notion of *violator spaces* [20, 21, 10, 11, 4]. The basic idea is that every subset of constraints is mapped to a unique basis, every basis has size at most δ (δ is the dimension of the problem, and is conceptually a constant), and certain conditions on consistency and monotonicity hold. Computing the basis of a violator space is not as easy as solving LP-type problems, because without a clear notion of progress, one can cycle through bases (which is not possible for LP-type problems). See Šavroň [21] for an example of such cycling. Nevertheless, Clarkson's algorithm [7] works for violator spaces [4].

We revisit the violator space framework, and show the following:

(A) Because of the cycling mentioned above, the standard version of Seidel's linear programming algorithm [22] does not work for violator spaces. However, it turns out that a variant of Seidel's algorithm does work for violator spaces.

(B) We demonstrate that violator spaces can be used to solve the problem of point-location in canonical decomposition. While in some cases this point-location problem can be stated as LP-type problem, stating it as a violator space problem seems to be more natural and elegant.

(C) The advantages of Seidel's algorithm is that except for constant work space, the only additional space it needs is to store the random permutations it uses. We show that one can use pseudo-random generators (PRGs) to generate the random permutation, so that there is no need to store it explicitly. This is of course well known – but the previous analysis [19] for linear programming implied only that the expected running time is $O(n \log^{\delta-1} n)$, where δ is the combinatorial dimension. Building on Mulmuley's work [19], we do a somewhat more careful analysis, showing that in this case one can use backward analysis on the random ordering of constraints generated, and as such the expected running time remains linear.

This implies that one can solve violator space problems (and thus, LP and LP-type problems) in constant dimension, using constant space, in expected linear time.

Paper organization. We present the new algorithm for computing the basis of violator spaces in Section 2. The adaptation of the algorithm to work with constant space is described in Section 2.3. We described corridor decomposition and its adaptation to our setting in Section 3. We present the shortest path algorithm in Section 4.

2 Violator spaces and constant space algorithms

First, we review the formal definition of *violator spaces* [20, 21, 10, 11, 4]. We then show that a variant of Seidel's algorithm for linear programming works for this abstract settings, and show how to adapt it to work with constant space and in expected linear time.

2.1 Formal definition of violator space

Before dwelling into the abstract framework, let us consider the following concrete example – hopefully it would help the reader in keeping track of the abstraction.

▶ **Example 1.** We have a set H of n segments in the plane, and we would like to compute the vertical trapezoid of $\mathcal{A}^{|}(H)$ that contains, say, the origin, where $\mathcal{A}^{|}(H)$ denote the vertical decomposition of the arrangement formed by the segments of H. Specifically, for a subset $X \subseteq H$, let $\tau(X)$ be the vertical trapezoid in $\mathcal{A}^{|}(X)$ that contains the origin. The vertical trapezoid $\tau(X)$ is defined by at most four segments, which are the *basis* of X. A segment $f \in H$ *violates* $\tau = \tau(X)$, if it intersects the interior of $\tau(X)$. The set of segments of H that intersects the interior of τ, denoted by $\mathrm{cl}(\tau)$ or $\mathrm{cl}(X)$, is the *conflict list* of τ.

Somewhat informally, violator space identifies a vertical trapezoid $\tau = \tau(X)$, by its conflict list $\mathrm{cl}(X)$, and not by its geometric realization (i.e., τ).

▶ **Definition 2.** A *violator space* is a pair $\mathcal{V} = (H, \mathsf{cl})$, where H is a finite set of *constraints*, and $\mathsf{cl} : 2^H \to 2^H$ is a function, such that:

- **Consistency**: For all $X \subseteq H$, we have that $\mathrm{cl}(X) \cap X = \emptyset$.
- **Locality**: For all $X \subseteq Y \subseteq H$, if $\mathrm{cl}(X) \cap Y = \emptyset$ then $\mathrm{cl}(X) = \mathrm{cl}(Y)$.
- **Monotonicity**: For all $X \subseteq Y \subseteq Z \subseteq H$, if $\mathrm{cl}(X) = \mathrm{cl}(Z)$ then $\mathrm{cl}(X) = \mathrm{cl}(Y) = \mathrm{cl}(Z)$.

A set $B \subseteq X \subseteq H$ is a *basis* of X, if $\mathrm{cl}(B) = \mathrm{cl}(X)$, and for any proper subset $B' \subset B$, we have that $\mathrm{cl}(B') \neq \mathrm{cl}(B)$. The *combinatorial dimension*, denoted by δ, is the maximum size of a basis.

Note that consistency and locality implies monotonicity. For the sake of concreteness, it would also be convenient to assume the following (this is strictly speaking not necessary for the algorithm).

▶ **Definition 3.** For any $X \subseteq H$ there is a unique *cell* $\tau(X)$ associated with it, where for any $X, Y \subseteq H$, we have that if $\mathrm{cl}(X) \neq \mathrm{cl}(Y)$ then $\tau(X) \neq \tau(Y)$. Consider any $X \subseteq H$, and any $f \in H$. For $\tau = \tau(X)$, the constraint f *violates* τ if $f \in \mathrm{cl}(X)$ (or alternatively, f *violates* X).

Finally, we assume that the following two basic operations are available:

- `violate`(f, B): Given a basis B (or its cell $\tau = \tau(B)$) and a constraint f, it returns true if f violates τ.

- `compBasis`(X): Given a set X with at most $(\delta+1)^2$ constraints, this procedure computes basis(X), where δ is the combinatorial dimension of the violator space[4]. For δ a constant, we assume that this takes constant time.

2.1.1 Examples

Linear programming as a violator space. Consider an instance I of linear programming in $\mathbb{R}^d$. Our interpretation is somewhat convoluted, but serves as a preparation for the next example. The instance I induces a polytope P in $\mathbb{R}^d$, which is the *feasible domain*. The vertices V of the polytope P induce a triangulation (assuming general position) of the sphere of directions, where a direction v belongs to a vertex p, if and only if p is an extreme vertex of P in the direction of v. Now, the objective function of I specifies a direction v_I, and in solving the LP, we are looking for the extreme vertex of P in this direction.

Put differently, every subset H of the constraints of I, defines a triangulation $\mathcal{T}(H)$ of the sphere of directions. So, let the cell of H, denoted by $\tau = \tau(H)$, be the spherical triangle in this decomposition that contains v_I. The basis of H is the subset of constraints that define $\tau(H)$. A constraint f of the LP violates τ if the vertex induced by the basis basis(H) (in the original space), is on the wrong side of f.

Thus solving the LP instance $I = (H, v_I)$ is no more than performing a point-location query in the spherical triangulation $\mathcal{T}(H)$, for the spherical triangle that contains v_I.

Doing point-location via violator spaces. Example 1 hints to a more general setup. So consider a space decomposition into canonical cells induced by a set of objects. For example, segments in the plane, with the canonical cells being the vertical trapezoids. More generally, consider any decomposition of a domain into simple canonical cells induced by objects, which complies with the Clarkson-Shor framework [8]. Examples of this include point-location in a (i) Delaunay triangulation, (ii) bottom vertex triangulation in an arrangement of hyperplanes, and (iii) many others.

▶ **Lemma 4.** *Consider a canonical decomposition of a domain into simple cells, induced by a set of objects, that complies with the Clarkson-Shor framework [8]. Then, performing a point-location query in such a domain is equivalent to computing a basis of a violator space.*

Proof. This follows readily from definition, see the full version [15] for details. ◀

It seems that for all of these point-location problems, one can solve them directly as LP-type problems. However, stating these problems as violator space problems is more natural as it avoids the need to explicitly define an artificial ordering over the bases, which can be quite tedious and not immediate.

2.2 The algorithm for computing the basis of a violator space

The input is a violator space $\mathcal{V} = (H, \mathsf{cl})$ with $n = |H|$ constraints, having combinatorial dimension δ.

[4] We consider basis(X) to be unique (that is, we assume implicitly that the input is in general position). This can be enforced by using lexicographical ordering, if necessary, among the defining bases always using the lexicographical minimum one.

```
solveVS(W, X):
    ⟨f_1, ..., f_m⟩: A random permutation of the constraints of X.
    B_0 ← compBasis(W)
    for i = 1 to m do
        if violate(f_i, B_{i−1}) then
            B_i ← solveVS(W ∪ B_{i−1} ∪ {f_i}, {f_1, ..., f_i})
        else
            B_i ← B_{i−1}
    return B_m
```

Figure 2.1 The algorithm for solving violator space problems. The parameter W is a set of $O(\delta^2)$ witness constraints, and X is a set of m constraints. The function return basis$(W \cup X)$. To solve a given violator space, defined implicitly by the set of constraints H, and the functions `violate` and `compBasis`, one calls `solveVS`$(\{\}, H)$.

2.2.1 Description of the algorithm

The algorithm is a variant of Seidel's algorithm [22] – it picks a random permutation of the constraints, and computes recursively in a randomized incremental fashion the basis of the solution for the first i constraints. Specifically, if the ith constraint violates the basis B_{i-1} computed for the first $i-1$ constraints, it calls recursively, adding the constraints of B_{i-1} and the ith constraint to the set of constraints that must be included whenever computing a basis (in the recursive calls). The resulting code is depicted in Figure 2.1.

The only difference with the original algorithm of Seidel, is that the recursive call gets the set $W \cup B_{i-1} \cup \{f_i\}$ instead of basis$(B_{i-1} \cup \{f_i\})$ (which is a smaller set). This modification is required because of the potential cycling between bases in a violator space.

2.2.2 The analysis

The key observation is that the depth of the recursion of `solveVS` is bounded by δ, where δ is the combinatorial dimension of the violator space. Indeed, if f_i violates a basis, the constraints added to the witness set W guarantee that any subsequent basis computed in the recursive call contains f_i, as testified by the following lemma.

▶ **Lemma 5.** *Consider any set $X \subseteq H$. Let $B = \text{basis}(X)$, and let f be a constraint in $H \setminus X$ that violates B. Then, for any subset Y such that $B \cup \{f\} \subseteq Y \subseteq X \cup \{f\}$, we have that $f \in \text{basis}(Y)$.*

Proof. Assume that this is false, and let Y be the bad set with $B' = \text{basis}(Y)$, such that $f \notin B'$. Since $f \in Y$, by consistency, $f \notin \text{cl}(Y)$, see Definition 2. By definition $\text{cl}(Y) = \text{cl}(B')$, which implies that $f \notin \text{cl}(B')$; that is, f does not violate B'.

Now, by monotonicity, we have $\text{cl}(Y) = \text{cl}(Y \setminus \{f\}) = \text{cl}(B')$. By assumption, $B \subseteq Y \setminus \{f\}$, which implies, again by monotonicity, as $B \subseteq Y \setminus \{f\} \subseteq X$, that $\text{cl}(X) = \text{cl}(Y \setminus \{f\}) = \text{cl}(B)$, as $B = \text{basis}(X)$. But that implies that $\text{cl}(B) = \text{cl}(Y) = \text{cl}(B')$. As $f \notin \text{cl}(Y)$, this implies that f does not violate B, which is a contradiction. ◀

▶ **Lemma 6.** *The depth of the recursion of `solveVS`, see Figure 2.1$_{p116}$, is at most δ, where δ is the combinatorial dimension of the given instance.*

Proof. Consider a sequence of k recursive calls, with $W_0 \subseteq W_1 \subseteq W_2 \subseteq \cdots \subseteq W_k$ as the different values of the parameter W of `solveVS`, where $W_0 = \emptyset$ is the value in the top-level call. Let f'_i, for $i = 1, \ldots, k$, be the constraint whose violation triggered the ith level call.

Observe that $f'_i \in W_i$, and as such all these constraints must be distinct (by consistency). Furthermore, we also included the basis B'_i, that f'_i violates, in the witness set W_i, which implies, by Lemma 5, that in any basis computation done inside this recursive call, it must be that $f'_i \in \text{basis}(W_j)$, for any $j \geq i$. As such, we have $f'_1, \ldots, f'_k \in \text{basis}(W_k)$. Since a basis can have at most δ elements, this is possible only if $k \leq \delta$, as claimed. ◀

▶ **Theorem 7.** *Given an instance of violator space $\mathcal{V} = (H, \mathsf{cl})$ with n constraints, and combinatorial dimension δ, the algorithm* **`solveVS`**$(\emptyset, H)$*, see Figure 2.1, computes* $\text{basis}(H)$*. The expected number of violation tests performed is bounded by $O(\delta^{\delta} n)$. Furthermore, the algorithm performs in expectation $O\Big((\delta \ln n)^{\delta}\Big)$ basis computations (on sets of constraints that contain at most $\delta(\delta+1)$ constraints).*

In particular, for constant combinatorial dimension δ, with violation test and basis computation that takes constant time, this algorithm runs in $O(n)$ expected time.

See the full version [15] for the proof of the above theorem.

2.3 Solving violator space problem with constant space and linear time

The key observation for turning **`solveVS`** into an algorithm that uses little space, is observing that the only thing we need to store (implicitly) is the random permutation used by **`solveVS`**.

2.3.1 Generating a random permutation using pseudo-random generators

To avoid storing the permutation, one can use pseudo-random techniques to compute the permutation on the fly. For our algorithm, we do not need a permutation - any random sequence that has uniform distribution over the constraints and is sufficiently long, would work.

▶ **Lemma 8.** *For any integer $\phi > 0$, a prime integer n, and an integer constant $c' \geq 12$, one can compute a random sequence of numbers $X_1, \ldots, X_{c'n} \in [\![n]\!] = \{1, \ldots, n\}$, such that:*
(A) *The probability of $X_i = j$ is $1/n$, for any $i \in [\![n]\!]$ and $j \in [\![c'n]\!]$.*
(B) *The sequence is ϕ-wise independent.*
(C) *Using $O(c'\phi)$ space, given an index i, one can compute X_i in $O(\phi)$ time.*

Proof. This is a standard pseudo-random generator (PRG) technique, described in detail by Mulmuley [19, p. 399]. We outline the idea. Randomly pick ϕ coefficients $\alpha_0, \ldots, \alpha_\phi \in \{0, \ldots, n-1\}$ (uniformly and independently), and consider the random polynomial $f_1(x) = \sum_{i=0}^{\phi} \alpha_i x^i$, and set $p(x) = (f(x) \mod n)$. Now, set $X_i = 1 + p(i)$, for $i = 1, \ldots, n$. It is easy to verify that the desired properties hold. To extent this sequence to be of the desired length, pick randomly c' such polynomials, and append their sequence together to get the desired longer sequence. It is easy to verify that the longer sequence is still ϕ-wise independent. ◀

The following lemma testifies that this PRG sequence, with good probability, contains the desired basis (as such, conceptually, we can think about it as being a permutation of $[\![n]\!]$).

▶ **Lemma 9.** *Let $B \subseteq [\![n]\!]$ be a specific set of δ numbers. For any integer $\phi \geq 8 + 2\delta$ and consider ϕ-wise independent random sequence of numbers $\mathcal{X} = \langle X_1, \ldots, X_{c'n} \rangle$, each uniformly distributed in $[\![n]\!]$, where c' is any constant $\geq 4(5 + \lceil \ln \delta \rceil)^2$. Then, the probability that the elements of B do not appear in $\mathcal{X}$ is bounded by, say, $1/20$.*

See the full version [15] for the proof of the above lemma.

▶ Remark. There are several low level technicalities that one needs to address in using such a PRG sequence instead of a truly random permutation:

(A) *Repeated numbers are not a problem*: the algorithm `solveVS` (see Figure 2.1$_{\text{p116}}$) ignores a constraint that is being inserted for the second time, since it can not violate the current basis.

(B) *Verifying the solution*: The sequence (of the indices) of the constraints used by the algorithm would be first $X_1, \ldots, X_{c'n}$. This sequence might miss some constraints that violates the computed solution.
As such, in the second stage, the algorithm check if any of the constraints $1, 2, \ldots, n$ violates the basis computed. If a violation was found, then the sequence generated failed, and the algorithm restart from scratch – resetting the PRG used in this level, regenerating the random keys used to initialize it, and rerun it to generate a new sequence.

(C) *Independence between levels*: We will use a different PRG for each level of the recursion of `solveVS`. Specifically, we generate the keys used in the PRG in the beginning of each recursive call. Since the depth of the recursion is δ, that would increase the space requirement by a factor of δ.

(D) *If the subproblem size is not a prime*: In a recursive call, the number of constraints given (i.e., m) might not be a prime. To this end, the algorithm can store (non-uniformly), a list of primes, such that for any m, there is a prime $m' \geq m$ that is at most twice bigger than m[5]. Then the algorithm generates the sequence modulo m', and ignores numbers that are larger than m. This implies that the sequence might contain invalid numbers, but such numbers are only a constant fraction of the sequence, so ignoring them does not change the running time analysis of our algorithm. (More precisely, this might cause the running time of the algorithm to deteriorate by a factor of $\exp(O(\delta))$, but as we consider δ to be a constant, this does not effect our analysis.)

One needs now to prove that backward analysis still works for our algorithm for violator spaces. The proof of the following lemma is implied by a careful tweaking of Mulmuley's analysis – we provide the details in the full version of the paper [15].

▶ **Lemma 10** (See [15]). *Consider a violator space $\mathcal{V} = (H, \mathsf{cl})$ with $n = |H|$, and combinatorial dimension δ. Let $i > 2\delta$, and let $\mathcal{X} = X_1, \ldots, X_{c'n}$ be a random sequence of constraints of H generated by ϕ-wise independent distribution (with each X_i having a uniform distribution), where $\phi > 6\delta + 9$ and $c' \geq 4(5 + \lceil \ln \delta \rceil)^2$ are constants. Then, for $i > 2\delta$, the probability that X_i violates $B = \text{basis}(X_1, \ldots, X_{i-1})$ is $O(1/i)$.*

2.3.2 The result

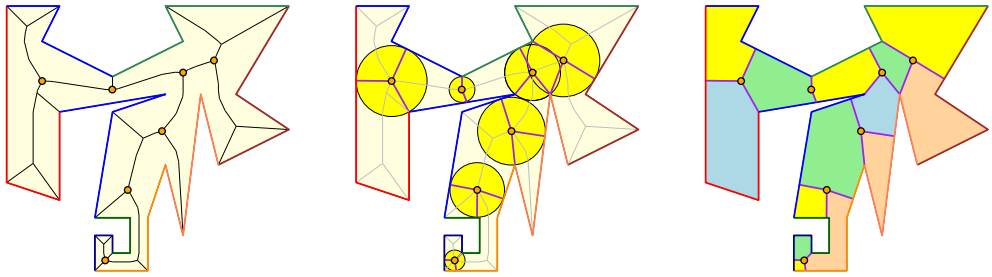
▶ **Theorem 11.** *Given an instance of violator space $\mathcal{V} = (H, \mathsf{cl})$ with n constraints, and combinatorial dimension δ, one can compute* $\text{basis}(H)$ *using $O\big(\delta^2 \log^2 \delta\big)$ space. For some constant $\zeta = O\big(\delta \log^2 \delta\big)$, we have that:*

(A) *The expected number of basis computations is $O\big((\zeta \ln n)^{\delta}\big)$, each done over $O(\delta^2)$ constraints.*

(B) *The expected number of violation tests performed is $O\big(\zeta^{\delta} n\big)$.*

(C) *The expected running time (ignoring the time to do the above operations) is $O\big(\zeta^{\delta} n\big)$.*

[5] That is, the program hard codes a list of such primes. The author wrote a program to compute such a list of primes, and used it to compute 50 primes that cover the range all the way to 10^{15} (the program run in a few seconds). However, it seems a bit redundant to include a list of such primes here. The interested reader can have a look here: `http://sarielhp.org/blog/?p=8700`.



(i) (ii) (iii)

Figure 3.1 An example of a corridor decomposition for a polygon: (i) Input curves, medial-axis and active vertices, (ii) their critical circles, and their spokes, and (iii) the resulting corridor decomposition.

Proof. The algorithm is described above. As for the analysis, it follows readily by plugging Lemma 10 into the proof of Theorem 7.

The only non-trivial technicality is to handle the case that the PRG sequence fails to contain the basis. Formally, abusing notations somewhat, consider a recursive call on the constraints indexed by $[\![n]\!]$, ad let B be the desired basis of the given subproblem. By Lemma 9, the probability that B is not contained in the generated PRG is bounded by $1/20$ – and in such a case the sequence has to be regenerated till success. As such, in expectation, this has a penalty factor of (say) 2 on the running time in each level. Overall, the analysis holds with the constants deteriorating by a factor of (at most) 2^{δ}. ◀

▶ Remark. Note, that the above pseudo-random generator technique is well known, but using it for linear programming by itself does not make too much sense. Indeed, pseudo-random generators are sometimes used as a way to reduce the randomness consumed by an algorithm. That in turn is used to derandomize the algorithm. However, for linear programming Megiddo's original algorithm was already linear time deterministic. Furthermore, Chazelle and Matoušek [6], using different techniques showed that one can even derandomize Clarkson's algorithm and get a linear running time with a better constant.

Similarly, using PRGs to reduce space of algorithms is by now a standard technique in streaming, see for example the work by Indyk [16], and references therein.

3 Corridor decomposition

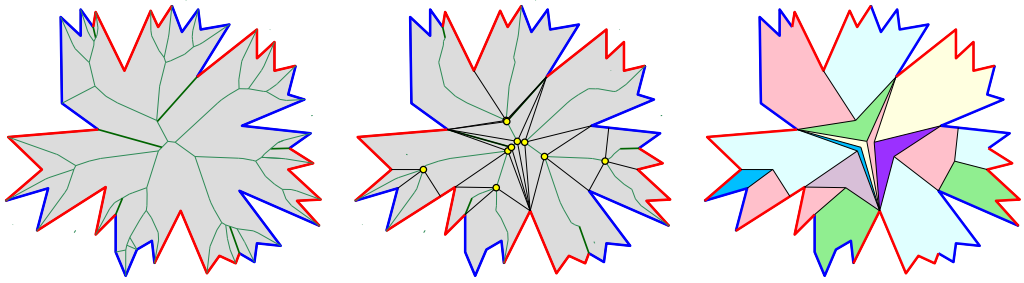
3.1 Construction

The decomposition here is similar to the decomposition described by the author in a recent work [14].

▶ **Definition 12** (Breaking a polygon into curves)**.** Let the polygon P have the vertices $v_1, \ldots v_n$ in counterclockwise order along its boundary. Let σ_i be the polygonal curve having the vertices $v_{(i-1)m+1}, v_{(i-1)m+2}, v_{(i-1)m+3}, \ldots, v_{im+1}$, for $i = 1, \ldots, \overline{\mathsf{n}} - 1$, where

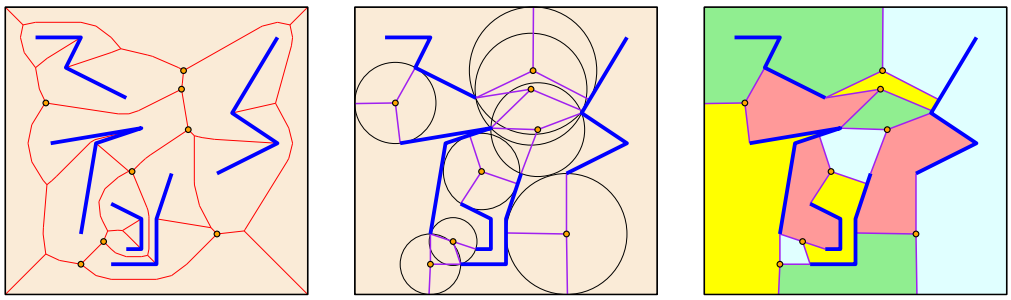
$$\overline{\mathsf{n}} = \lfloor (n-1)/m \rfloor + 1.$$

The last polygonal curve is $\sigma_{\overline{\mathsf{n}}} = v_{(\overline{\mathsf{n}}-1)m+1}, v_{(\overline{\mathsf{n}}-1)m+2}, \ldots v_n, v_1$. Note, that given P in a read only memory, one can encode any curve σ_i using $O(1)$ space. Let $\Gamma = \{\sigma_1, \ldots, \sigma_{\overline{\mathsf{n}}}\}$ be the resulting set of polygonal curves. From this point on, a *curve* refers to a polygonal curve generated by this process.



(i) (ii) (iii)

Figure 3.2 Another example of a corridor decomposition for a polygon: (i) Input polygon and its curves and its medial-axis (the thick lines are the angle bisectors for the obtuse angles where two curves meet), (ii) active vertices and their spokes (with a reduced medial axis), and iii) the resulting corridor decomposition.



(i) (ii) (iii)

Figure 3.3 Corridor decomposition for disjoint curves: (i) Input curves, the medial-axis, and active vertices, (ii) the critical circles, and their spokes, and (iii) the resulting corridor decomposition.

Corridor decomposition for the whole polygon. Next, consider the medial axis of P restricted to the interior of P. A vertex v of the medial axis corresponds to a disk D, that touches the boundary of P in two or three points (by general position assumption, not in any larger number of points). The medial axis has the topological structure of a tree.

To make things somewhat cleaner, we pretend that there is a little hole centered at every vertex of the polygon if it is the common endpoint of two curves. This results in a medial axis edge that comes out of the vertex as an angle bisector, both for an acute angle (where a medial-axis edge already exists), and for obtuse angles, see Figure 3.1 and Figure 3.2.

A vertex of the medial axis is *active* if its disk touches three different curves of Γ. It is easy to verify that there are $O(\overline{\mathsf{n}})$ active vertices. The segments connecting an active vertex to the three (or two) points of tangency of its empty disk with the boundary of P are its *spokes*. Introducing these spokes breaks the polygon into the desired *corridors*.

Corridor decomposition for a subset of the curves. For a subset $\Psi \subseteq \Gamma$, of total complexity t, one can apply a similar construction. Again, compute the medial axis of the curves of Ψ, by computing, in $O(\mathsf{t} \log \mathsf{t})$ time, the Voronoi diagram of the segments used by the curves [9], and extracting the medial axis (it is now a planar graph instead of a tree). Again, by considering the active vertices, building their associated spokes, results in a decomposition into corridors. For technical reasons, it is convenient to add a large bounding box, and restrict the construction to this domain, treating this *frame* as yet another input curve. Figure 3.3 depicts one such corridor decomposition.

Let $\mathcal{C}(\Psi)$ denote this resulting decomposition into corridors.

3.1.1 Properties of the resulting decomposition

Every corridor in the resulting decomposition $\mathcal{C}(\Psi)$ is defined by a constant number of input curves. Specifically, consider the set of all possible corridors; that is $\mathcal{F} = \bigcup_{\Upsilon \subseteq \Gamma} \mathcal{C}(\Upsilon)$. Next, consider any corridor $C \in \mathcal{F}$, then there is a unique *defining set* $D(C) \subseteq \Psi$ (of at most 4 curves). Similarly, such a corridor has *stopping set* (or *conflict list*) of C, denoted by $K(C)$.

Consider any subset $\mathcal{S} \subseteq \Gamma$. It is easy to verify that the following two conditions hold:

(i) For any $C \in \mathcal{C}(\mathcal{S})$, we have $D(C) \subseteq \mathcal{S}$ and $\mathcal{S} \cap K(C) = \emptyset$.

(ii) If $D(C) \subseteq \mathcal{S}$ and $K(C) \cap \mathcal{S} = \emptyset$, then $C \in \mathcal{C}(\mathcal{S})$.

Namely, the corridor decomposition complies with the technique of Clarkson-Shor [8] (see also [13, Chapter 8]).

3.2 Computing a specific corridor

Let p be a point in the plane, and let Γ be a set of $\overline{\mathsf{n}}$ interior disjoint curves (stored in a read only memory), where each curve is of complexity m. Let n be the total complexity of these curves (we assume that $n = \Theta(m\,\overline{\mathsf{n}})$). Our purpose here is to compute the corridor $C \in \mathcal{C}(\Gamma)$ that contains p. Formally, for a subset $\Psi \subseteq \Gamma$, we define the function $w(\Psi)$, to be the defining set of the corridor $C \in \mathcal{C}(\Psi)$ that contains p. Note, that such a defining set has cardinality at most $\delta = 4$.

Basic operations. We need to specify how to implement the two basic operations:

(A) **(Basis computation)** Given a set of $O(1)$ curves, we compute their medial axis, and extract the corridor containing p. This takes $O(m \log m)$ time.

(B) **(Violation test)** Given a corridor C, and a curve σ, both of complexity $O(m)$, we can check if σ violates the corridor by checking if an arbitrary vertex of σ is contained in C (this takes $O(m)$ time to check), and then check in $O(m)$ time, if any segment of σ intersects the doors of the corridor on its two sides. This takes $O(m)$ time.

▶ **Lemma 13.** *Given a polygon P with n vertices, stored in read only memory, and let m be a parameter. Let Γ be the set of $\overline{\mathsf{n}}$ curves resulting from breaking P into polygonal curves each with m vertices, as described in Definition 12. Then, given a query point p inside P, one can compute, in $O(n + m \log m \log^4 \overline{\mathsf{n}})$ expected time, the corridor of $\mathcal{C}(\Gamma)$ that contains p. This algorithm uses $O(1)$ additional space.*

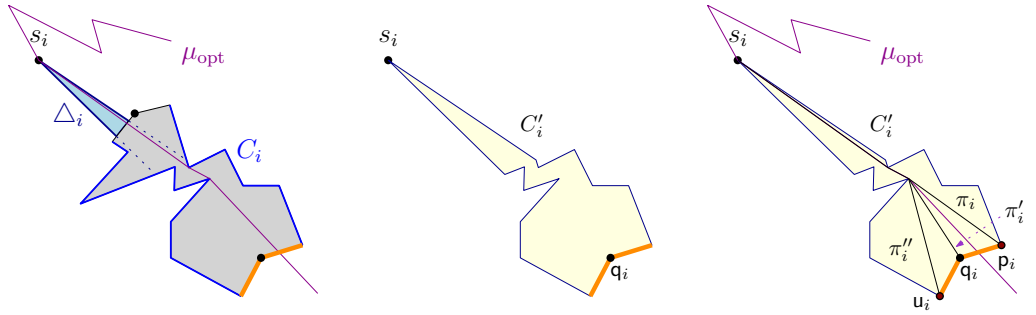
See the full version [15] for the proof of the above lemma.

4 Shortest path in a polygon in sublinear space

Let P be a simple polygon with n edges in the plane, and let s and t be two points in P, where s is the *source*, and t is the *target*. Our purpose here is to compute the shortest path between s and t inside P. The vertices of P are stored in (say) counterclockwise order in an array stored in a read only memory. Let m be a prespecified parameter that is (roughly) the amount of additional space available for the algorithm.

4.1 Updating the shortest path through a corridor

A corridor has two *doors* – a door is made of two segments, with a middle endpoint in the interior of the polygon, and the other endpoints on the boundary of the polygon. The rest of the boundary of the corridor is made out two chains from the original polygon.



(i) (ii) (iii)

Figure 4.1 (i) The state in the beginning of the ith iteration. (ii) The clipped polygon C'_i. (iii) The funnel created by the shortest paths from s_i to the two spoke endpoints.

Given two rays σ and σ', that share their source vertex v (which lies inside P), consider the polygon Q that starts at v, follows the ray σ till it hits the boundary of P, then trace the boundary of P in a counterclockwise direction till the first intersection of σ' with $\partial\mathsf{P}$, and then back to v. The polygon $\mathsf{Q} = \mathsf{P}\langle\sigma, \sigma'\rangle$ is the *clipped* polygon. See Figure 4.1.

A *geodesic* is the shortest path between two points (restricted to lie inside P). Two geodesics might have a common intersection, but they can cross at most once. Locally, inside a polygon, a geodesic is a straight segment. For our algorithm, we need some basic operations:

(A) `isPntIn`(p): Given a query point p, it decides if p is inside P. This is done by scanning the edges of P one by one, and counting how many edges crosses the vertical ray shooting from p downward. This operation takes linear time (in the number of vertices of P).

(B) `isInSubPoly`($\mathsf{p}, \sigma, \sigma'$): returns true if p is in the clipped polygon $\mathsf{P}\langle\sigma, \sigma'\rangle$. It is easy to verify that this can be implemented to work in linear time and constant space.

Using vertical and horizontal rays shot from s, one can decide, in $O(n)$ time, which quadrant around s is locally used by the shortest path from s to t. Assume that this path is in the positive quadrant. It would be useful to think about geodesics starting at s as being sorted angularly. Specifically, if τ and τ' are two geodesic starting at s, then τ is to the *left* of τ', if the first edge of τ is counterclockwise to the first edge of τ'. If the prefix of τ and τ' is non-empty, we apply the same test to the last common point of the two paths. Let $\tau \prec \tau'$ denote that τ is to the left of τ'.

In particular, if the endpoint of the rays σ, σ' is the source vertex s, and the geodesic between s and t lies in $\mathsf{P}\langle\sigma, \sigma'\rangle$, then given a third ray π lying between σ and σ', the shortest path between s and t in P must lie completely either in $\mathsf{P}\langle\sigma, \pi\rangle$ or $\mathsf{P}\langle\pi, \sigma'\rangle$, and this can be tested by a single call to `isInSubPoly` for checking if t is in $\mathsf{P}\langle\pi, \sigma'\rangle$.

4.1.1 Limiting the search space

▶ **Lemma 14.** *Let P, s and t be as above, and μ be the shortest path from s to t in P. Let pq be the last edge in the shortest path τ from s to q, where q is in P. Then, one can decide in $O(n)$ time, and using $O(1)$ space, if $\mu \prec \tau$, where n is the number of vertices of P.*

See the full version [15] for the proof of the above lemma.

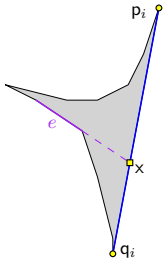


Figure 4.2 Funnel reduction.

4.1.2 Walking through a corridor

In the beginning of the ith iteration of the algorithm it would maintain the following quantities (depicted in Figure 4.1 (i)):

(A) s_i: the current source (it lies on the optimal shortest path μ_{opt} between s and t).

(B) C_i: The current corridor.

(C) $\triangle_i$: A triangle having s_i as one of its vertices, and its two other vertices lie on a spoke of C_i. The shortest path μ_{opt} passes through s_i, and enters C_i through the base of $\triangle_i$, and then exists the corridor through one of its "exit" spokes.

The task at hand is to trace the shortest path through C_i, in order to compute where the shortest path leaves the corridor.

▶ **Lemma 15.** *Tracing the shortest path μ_{opt} through a single corridor takes $O\bigl(n \log m + m \log m \log^4 \overline{n}\bigr)$ expected time, using $O(m)$ space.*

Proof. We use the above notation. The algorithm glues together $\triangle_i$ to C_i to get a new polygon. Next, it clips the new polygon by extending the two edges of $\triangle_i$ from s_i. Let C_i' denote the resulting polygon, depicted in Figure 4.1 (ii). Let the three vertices of C_i' forming the two "exit" spokes be $\mathsf{p}_i, \mathsf{q}_i, \mathsf{u}_i$. Next, the algorithm computes the shortest path from s_i to the three vertices $\mathsf{p}_i, \mathsf{q}_i, \mathsf{u}_i$ inside C_i', and let π_i, π_i', π_i'' be these paths, respectively (this takes $O(|C_i'|) = O(m)$ time [12]). Using Lemma 14 the algorithm decides if $\pi_i \prec \mu_{\text{opt}} \prec \pi_i'$ or $\pi_i' \prec \mu_{\text{opt}} \prec \pi_i''$. We refer to a prefix path (that is part of the desired shortest path) followed by the two concave chains as a *funnel* – see Figure 4.1 (iii) and Figure 4.2 for an example.

Assume that $\pi_i \prec \mu_{\text{opt}} \prec \pi_i'$, and let F_i be the funnel created by these two shortest paths, where $\mathsf{p}_i\mathsf{q}_i$ is the base of the funnel. If the space bounded by the funnel is a triangle, then the algorithm sets its top vertex as s_{i+1}, the funnel triangle is $\triangle_{i+1}$, and the algorithm computes the corridor on the other side of $\mathsf{p}_i\mathsf{q}_i$ using the algorithm of Lemma 13, set it as C_{i+1}, and continues the execution of the algorithm to the next iteration.

So the problem is when funnel chains are "complicated" concave polygons (with at most $O(m)$ vertices), see Figure 4.2. As long as the funnel F_i is not a triangle, pick a middle edge e on one side of the funnel, and extend it till it hits the edge $\mathsf{p}_i\mathsf{p}_{i+1}$, at a point x. This breaks F_i into two funnels, and using the algorithm of Lemma 14 on the edge e, decide which of these two funnels contains the shortest path μ_{opt}, and replace F_i by this funnel. Repeat this process till F_i becomes a triangle. Once this happens, the algorithm continues to the next iteration as described above. Clearly, this funnel "reduction" requires $O(\log m)$ calls to the algorithm of Lemma 14.

Note, that the algorithm "forgets" the portion of the funnel that is common to both paths as it moves from C_i to C_{i+1}. This polygonal path is a part of the shortest path μ_{opt}

computed by the algorithm, and it can be output at this stage, before moving to the next corridor C_{i+1}.

In the end of the iteration, this algorithm computes the next corridor C_{i+1} by calling the algorithm of Lemma 13. ◀

4.2 The algorithm

The overall algorithm works by first computing the corridor C_1 containing the source $s_1 = s$ using Lemma 13. The algorithm now iteratively applies Lemma 15 till arriving to the corridor containing t, where the remaining shortest path can be readily computed. Since every corridor gets visited only once by this walk, we get the following result.

▶ **Theorem 16.** *Given a simple polygon* P *with* n *vertices (stored in a read only memory), a start vertex* s*, a target vertex* t*, and a space parameter* m*, one can compute the length of the shortest path from* s *to* t *(and output it), using* $O(m)$ *additional space, in* $O\bigl(n^2/m\bigr)$ *expected time, if* $m = O(n/\log^2 n)$*. Otherwise, it is* $O\Bigl(\frac{n^2}{m} + n \log m \log^4 \overline{\mathsf{n}}\Bigr)$*.*

Proof. The algorithm is described above, and let $\overline{\mathsf{n}} = \lfloor (n-1)/m \rfloor + 1$. There are $O(\overline{\mathsf{n}})$ corridors, and this bounds the number of iterations of the algorithm. As such, the overall expected running time is $O\bigl(\overline{\mathsf{n}}\bigl(n \log m + m \log m \log^4 \overline{\mathsf{n}}\bigr)\bigr) = O\Bigl(\frac{n^2}{m} \log m + n \log m \log^4 \overline{\mathsf{n}}\Bigr)$.

To get a better running time, observe that the extra log factor (on the first term), is rising out of the funnel reduction $O(\log m)$ queries inside each corridor, done in the algorithm of Lemma 15. If instead of reducing a funnel all the way to constant size, we reduce it to have say, at most $\lceil m/4 \rceil$ edges (triggered by the event that the funnel has at least $m/2$ edges), then at each invocation of Lemma 15, only a constant number of such queries would be performed. One has to adapt the algorithm such that instead of a triangle entering a new corridor, it is a funnel. The adaptation is straightforward, and we omit the easy details. The improved running time is $O\Bigl(\frac{n^2}{m} + n \log m \log^4 \overline{\mathsf{n}}\Bigr)$. ◀

5 Conclusions

The most interesting open problem remaining from our work, is whether one can improve the running time for computing the shortest path in a polygon with $O(m)$ space to be faster than $O(n^2/m)$.

Acknowledgments. The author became aware of the low-space shortest path problem during Tetsuo Asano talk in the Workshop in honor of his 65th birthday during SoCG 2014. The author thanks him for the talk, and the subsequent discussions. The author also thanks Pankaj Agarwal, Chandra Chekuri, Jeff Erickson and Bernd Gärtner for useful discussions. The authors also thanks the anonymous referees for their detailed comments, and their patience with the numerous typos in the submitted version.

References

1 T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *Comput. Geom. Theory Appl.*, 46(8):959–969, 2013.

2 T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Reprint of: Memory-constrained algorithms for simple polygons. *Comput. Geom. Theory Appl.*, 47(3):469–479, 2014.

3 M. de Berg, O. Cheong, M. van Kreveld, and M. H. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Santa Clara, CA, USA, 3rd edition, 2008.

4 Y. Brise and B. Gärtner. Clarkson's algorithm for violator spaces. *Comput. Geom. Theory Appl.*, 44(2):70–81, 2011.

5 B. Chazelle, D. Liu, and A. Magen. Sublinear geometric algorithms. *SIAM J. Comput.*, 35(3):627–646, 2005.

6 B. Chazelle and J. Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. *J. Algorithms*, 21:579–597, 1996.

7 K. L. Clarkson. Las Vegas algorithms for linear and integer programming. *J. Assoc. Comput. Mach.*, 42:488–499, 1995.

8 K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

9 S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

10 B. Gärtner, J. Matoušek, L. Rüst, and P. Šavroň. Violator spaces: Structure and algorithms. In *Proc. 14th Annu. European Sympos. Algorithms* (ESA), pages 387–398, 2006.

11 B. Gärtner, J. Matoušek, L. Rüst, and P. Šavroň. Violator spaces: Structure and algorithms. *Discrete Appl. Math.*, 156(11):2124–2141, 2008.

12 L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.*, 39(2):126–152, October 1989.

13 S. Har-Peled. *Geometric Approximation Algorithms*, volume 173 of *Mathematical Surveys and Monographs*. Amer. Math. Soc., Boston, MA, USA, 2011.

14 S. Har-Peled. Quasi-polynomial time approximation scheme for sparse subsets of polygons. In *Proc. 30th Annu. Sympos. Comput. Geom.* (SoCG), pages 120–129, 2014.

15 S. Har-Peled. Shortest path in a polygon using sublinear space. *CoRR*, abs/1412.0779, 2014.

16 P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. Assoc. Comput. Mach.*, 53(3):307–323, 2006.

17 D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14:393–410, 1984.

18 N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. Assoc. Comput. Mach.*, 31:114–127, 1984.

19 K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.

20 L. Y. Rüst. *The P-Matrix Linear Complementarity Problem – Generalizations and Specializations*. PhD thesis, ETH, 2007. Diss. ETH No. 17387.

21 P. Šavroň. *Abstract models of optimization problems*. PhD thesis, Charles University, 2007. `http://kam.mff.cuni.cz/~xofon/thesis/diplomka.pdf`.

22 R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.

23 M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proc. 9th Sympos. Theoret. Aspects Comput. Sci.*, volume 577 of *Lect. Notes in Comp. Sci.*, pages 569–579, London, UK, 1992. Springer-Verlag.