HEURISTICS FOR THE GENERATION OF RANDOM POLYGONS

Diplomarbeit
zur Erlangung des Diplomingenieurgrades
an der Naturwissenschaftlichen Fakultät
der Universität Salzburg

eingereicht von
Thomas AUER

Salzburg, 25. Juli 1996

Contents

List of Tables

List of Figures

Abstract

We consider the problem of randomly generating simple polygons on a given set of points. This problem is of considerable importance in the practical evaluation of algorithms that operate on polygons, where it is necessary to check the correctness and to determine the actual CPU-consumption of an algorithm experimentally.

We first present an approach by O'Rourke and Virmani (1991). Next, we discuss the algorithm of Zhu, Sundaram, Snoeyink and Mitchell (1996) for the uniformly random generation of x-monotone polygons. In addition, we devise a solution for the uniformly random generation of star-shaped polygons. Since no polynomial-time solution for the uniformly random generation of polygons is known, we present and analyze five heuristics. All heuristics have been implemented and tested, and we give experimental results, comparing the numbers of polygons generated by the heuristics to the actual numbers of all simple polygons on given sets of points. We also report on applicability of the heuristics and on the quality of the polygons generated. Three heuristics turned out to be applicable for practical purposes, both as far as the CPU-consumption and as far as the quality of the polygons generated is concerned.

Acknowledgments

First of all I want to thank Martin Held: This work would not have been possible without his continuing help and support. In the various discussions with him, many details became clear and many problems arising while working on the thesis were solved. Further, I am grateful to Christian Linhart, Joseph Mitchell and Karel Zikan for interesting discussions on the topic of this thesis.

I thank Stephan Wegenkittl and the other members of the University of Salzburg's pLab research group for helpful discussions on the generation of pseudo random numbers.

Chapter 1

Introduction

1.1 Problem Definition

In this thesis the random generation of polygons in two dimensional space is studied. Since there exists no widely accepted definition of a random polygon in the plane[1], we focus on the well-defined problem of generating a random simple[2] polygon on a given set $\mathcal{S}$ of n points. Note that there exist $\frac{(n-1)!}{2}$ different polygons on $\mathcal{S}$, where "different" means different in shape: Since we want to count each polygon exactly once, we fix the first vertex, and thus have $(n-1)!$ possibilities to arrange the remaining $(n-1)$ vertices. Further, each polygon appears twice (with vertices in clockwise respectively counterclockwise order), but we want to consider it only once. Clearly, as depicted in Fig. 1.1, not all of these polygons are simple: Only four (cf. Fig. 1.1a – Fig. 1.1d) of the $\frac{(5-1)!}{2} = 12$ polygons that exist for the depicted set of five points are simple, whereas the remaining eight polygons (cf. Fig. 1.1e – Fig. 1.1l) exhibit self-intersections.

From a theoretical point of view, the problem of random polygon generation is defined as follows: Given $\mathcal{S}$, we would like to generate a simple polygon on $\mathcal{S}$ at random with a uniform distribution. (I.e., if there exist k simple polygons on $\mathcal{S}$, we want to create any of these polygons with a probability of $\frac{1}{k}$.)

Consider two different pairs of points and the two edges defined by these pairs. It is easy to observe that the number of simple polygons containing one of these edges may differ. This is illustrated in Fig. 1.2, where the interesting edge is represented by a dashed line: For the first edge there exist exactly two simple polygons (cf. Fig. 1.2a – Fig. 1.2b) which contain it, whereas for the second edge three simple polygons (cf. Fig. 1.2c – Fig. 1.2e) containing it exist.

Unfortunately, no algorithm running in polynomial time is known that generates random simple polygons with a uniform distribution. This motivated us to pursue heuristics for generating random simple polygons.

---

[1] In [OV91], the authors give two measures for the randomness of a polygon, but they admit that these measures are not totally adequate for characterizing the randomness of polygons.

[2] A simple polygon is a polygon without self-intersections. A formal definition is given in Section 1.5.

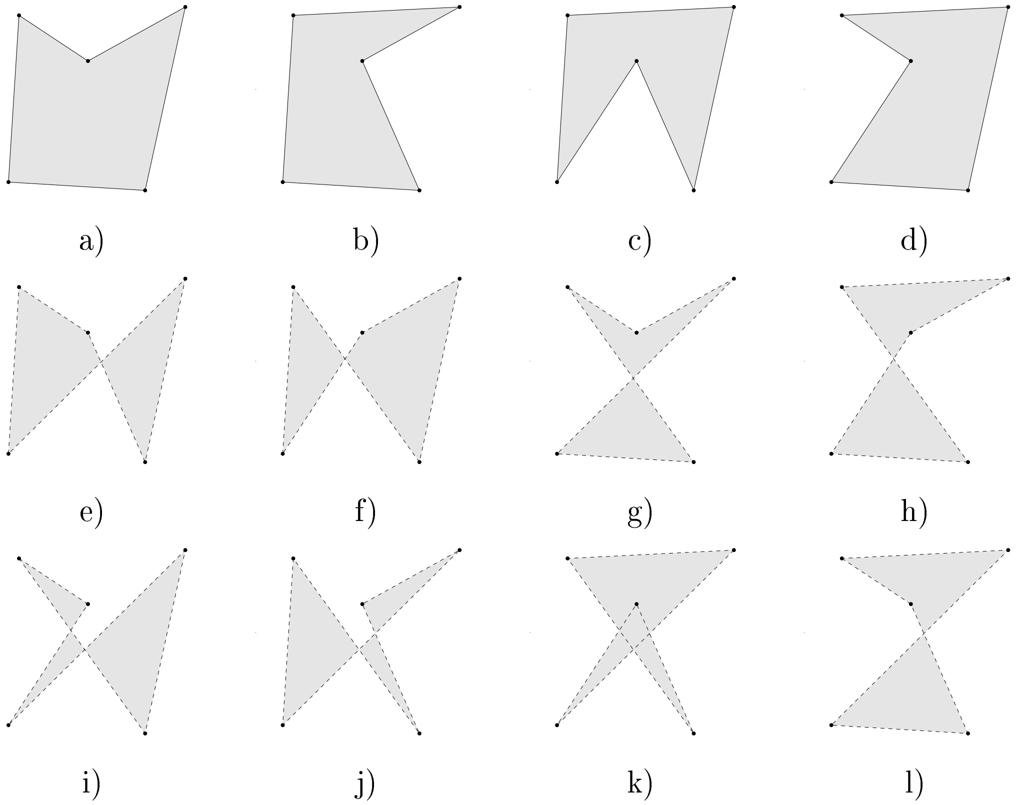


Figure 1.1: A set $\mathcal{S}$ with five points and its polygons.

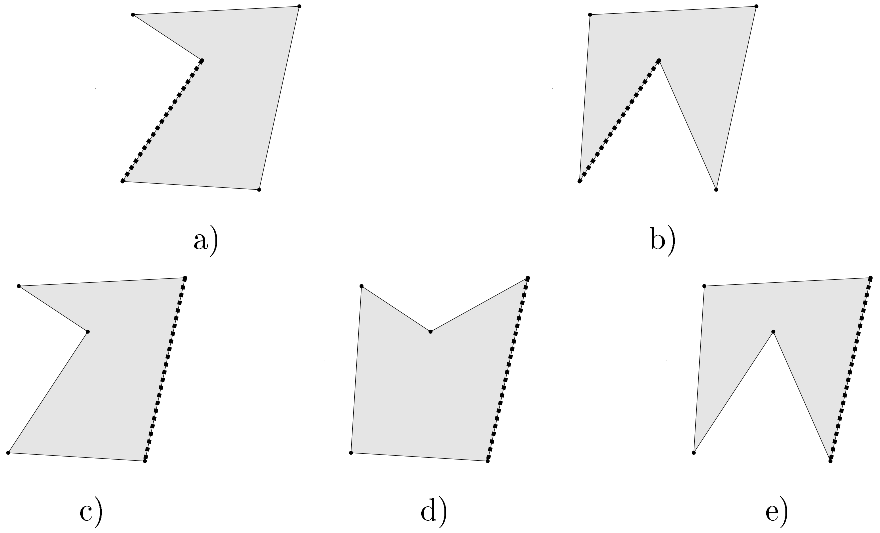


Figure 1.2: Two edges belonging to a different number of simple polygons.

1.2 Motivation

The generation of random simple polygons has two main areas of application:

a) the empirical verification of an algorithm operating on polygons,

b) the practical testing of its CPU-consumption.

Consider an algorithm which works on polygonal input data. Typically, one would like to test this algorithm in a way such that bugs are found. Obviously, the best way for testing an algorithm is to perform a white-box test, i.e., to generate test data that covers all possible branches of the algorithm. However, this may not always be possible: Either the source of the algorithm is not available for testing, or the number of required inputs is far too large for testing all the branches of the algorithm. In both cases one has to rely on a different testing strategy. (I.e., testing without considering the details or different branches of the algorithm.) Thus, one cannot guarantee that testing will cover all possible branches of the algorithm and all that can be done is to test whether the algorithm handles all classes of input data correctly. Therefore, data that covers all possible classes (of inputs) would be needed for efficient testing. Since such a set of data may become both too large and too hard to define for practical purposes, what one might do is to use randomly generated data that has a high probability to cover all the different classes of inputs.

The second application of random polygons is testing the CPU-time consumption of an algorithm. Ideally, one would like to test an algorithm with data of practical relevance. However, in practice it is next to impossible to obtain a sufficiently large amount of data needed for testing. Furthermore, companies tend to be reluctant to provide classified data, even if a non-disclosure agreement has been signed. Thus, since practical data may not be available for testing, it is natural to test the algorithm on randomly created input data.

Beside being a topic of practical relevance, the random generation of simple polygons is a question of theoretical interest of its own: No polynomial-time algorithm is known for enumerating or even counting the number of simple polygons on a given point set. Also, no polynomial-time algorithm is known for the random generation of a simple polygon without the vertices being fixed a priori. A necessary prerequisite would be a measure for the randomness of polygons, which is yet to be devised.

1.3 Related Work

For the reasons mentioned above, the random generation of geometric objects has received some attention by researchers: Epstein and Sack [ES92] presented an $\mathcal{O}(n^4)$ algorithm for generating triangulations of simple polygons at random. They use dynamic programming to count all the triangulations that contain a triangle with edge (v_i, v_{i+k}) for increasing values of k. (I.e., they consider the triangles $\triangle(v_i, v_j, v_{i+k})$ for suitable v_j with $i < j < i+k$.) This number is the product of the number of triangulations that contain (v_i, v_j) and the number of triangulations containing (v_j, v_{i+k}). Note that these numbers have already been computed, since $|j-i| < k$ and $|i+k-j| < k$. This enumeration of all triangulations can be done in $\mathcal{O}(n^3)$. Then this enumeration is used to generate a triangulation uniformly at random: Start with an ear[3] $\triangle(v_{e-1}, v_e, v_{e+1})$, and either "cut" the ear (i.e., insert edge (v_{e-1}, v_{e+1}) into the triangulation) or select an edge that intersects (v_{e-1}, v_{e+1}).

Devroye, Epstein and Sack [DES93] studied the random generation of intervals and hyperrectangles. They consider the problem of generating a random hyperrectangle in a unit hypercube such that each point of the hypercube has probability p of being covered. For random subintervals of $[0, 1]$, various methods based either on the distribution of the length or on the distribution of the midpoint are compared. Further, it is shown that no solution exits for subintervals with a fixed length.

Atkinson and Sack [AS94b] studied the uniform generation of forests of restricted height. A k-way tree is either empty or consists of a root node and an ordered sequence of k subtrees, each of which is a k-way tree. Each tree node consists of some data field and k pointers to other tree nodes. A k-way forest is an ordered sequence of non-empty k-way trees. First they present an algorithm, based on recurrence, for counting all k-way forests. This algorithm runs in $\mathcal{O}(n^3 \cdot h)$, where h denotes the height of the tree. Then this counting algorithm is used to select one k-way forest uniformly at random.

An algorithm running in at most $\mathcal{O}(n^2)$ time for the random generation of x-monotone polygons was described by Zhu et al. [ZSSM96]. We will discuss this algorithm in detail in Section 2.2 because it is part of our implementation. In addition, these authors presented an algorithm that generates a convex polygon whose vertices are a subset of a given set of n points: This is done by selecting some vertex v_i and then counting the number of convex polygons such that v_i is the "lowest"[4] vertex in the polygon.

An interesting approach for the generation of random polygons in the plane (but not on a given set of points) was researched by O'Rourke and Virmani [OV91]. It is discussed in detail in Chapter 2 because it also is part of our implementation.

1.4 Survey

In this section, we give an overview over the thesis. The thesis consists of four parts: The first part (Chapter 1) is the introduction, which contains (in addition to the subsequent survey of the algorithms studied) the basic concepts used throughout the thesis and the notation used. In the section on basic concepts (Section 1.5), we included only fundamental concepts and concepts that are used for more than one algorithm. Advanced concepts that are used only for one algorithm are given when they are needed, i.e., when explaining the algorithm.

The second part (Chapter 2) of the thesis gives the description of the algorithms we analyzed and implemented. The algorithms comprise several different classes: First, we have Bouncing Vertices[5], which is different from all the other algorithms

---

[3] Three consecutive vertices v_{e-1}, v_e and v_{e+1} form an ear of a polygon if (v_{e-1}, v_{e+1}) is a diagonal of the polygon.

[4] The vertex with the minimum y-coordinate.

[5] This is the algorithm by O'Rourke and Virmani citeORVi91.

as it generates polygons by moving vertices. For two subproblems of the original problem[6], exact solutions exist: For the enumeration (and random generation) of x-monotone polygons Zhu et al. [ZSSM96] devised an algorithm. Next, we present our solution for the random generation of star-shaped polygons. Since our exact algorithms for star-shaped polygons are not feasible in practice, we present a fast heuristic, Quick Star. Finally, we studied five heuristics for the random generation of a simple polygon. (Four of these heuristics have been devised by the author.) In Subsection 1.4.1, we will give a short overview over these algorithms and outline their basics.

In Chapter 3, we describe the implementation[7] of these algorithms in detail. We used the language C for the implementation. For the generation of random numbers we used the linear congruential random number generator *rand48*, which belongs to the standard C library. In addition to a test bed, we enhanced our implementation by a graphical user interface based on the Forms Library by Zhao and Overmars [ZO95].

In order to keep the implementation simple and still be able to handle data without the underlying assumption of general position, our implementation differs slightly from the description given above. However, these differences can be expected not to affect the test results. See Chapter 3 for details.

In Chapter 4, we present the results we obtained with our implementation. We ran three different series of experiments:

1. We recorded the CPU-consumption of our algorithms.
2. We obtained experimental bounds on the numbers of star-shaped and simple polygons in terms of the cardinality of the point set.
3. We evaluated the number of polygons generated by our algorithms in order to assess the quality and practical applicability of these heuristics.

In addition, we determined experimental bounds for parameters which have a great influence on the actual running time of the algorithms. Note that we implemented but did not test the algorithm by O'Rourke and Virmani [OV91] and the algorithm by Zhu et al. [ZSSM96].

Finally, in Chapter 5, we summarize the test results and evaluate the algorithms presented. Further, we list open problems related to the generation of random polygons.

Note that the bibliography given at the end of this thesis constitutes a literature survey which includes more publications than are referenced in the thesis.

---

[6] To compute a random polygon on a given set of vertices.

[7] The implementation is available for ftp'ing at ftp.cosy.sbg.ac.at in the directory /pub/people/tom/mth as file source.tar.gz.

1.4.1 Algorithms

Bouncing Vertices

Bouncing Vertices, suggested by O'Rourke and Virmani [OV91], generates a random polygon by starting with some (random) simple polygon (we will use a star-shaped polygon) and then moving the vertices of the polygon at random while maintaining simplicity. (I.e., the vertices may not be moved in a manner such that self-intersections occur.) This is done by testing the two edges incident upon vertex v with all other edges for intersection when moving v. While an intersection occurs, another random move is computed and tested for feasibility.

x-Monotone Polygons

The algorithm by Zhu et al. [ZSSM96] generates x-monotone polygons uniformly at random. It has a time complexity of $\mathcal{O}(n^2)$ and it is based on the following approach: Each x-monotone polygon can be considered as the union of two x-monotone chains, where the top chain lies above the bottom chain. Thus, Zhu et al. scan forward along the x-axis, and for each vertex they count the number of polygons which contain the current vertex in the top chain and the number of polygons which contain it in the bottom chain. This is done by recurrence: They show that a polygon on $\mathcal{S}_k$ containing edge (s_i, s_k) on the top chain (with $i < k - 1$) can be simplified to a polygon on $\mathcal{S}_{i+1}$ with (s_i, s_{i+1}) on the top chain. Further, if (s_i, s_k) is an edge of the top chain then (s_{k-1}, s_k) must be on the bottom chain. Thus, the number of all polygons with (s_{k-1}, s_k) on the bottom chain is the sum over all polygons that have edge (s_i, s_{i+1}) on their top chain, for suitable points s_i.

All suitable points s_i that may form an edge on the top chain must be visible from s_k with respect to the monotone chain $(s_1, s_2, \ldots, s_k)$. Thus, all visible points are found with the help of a shortest path tree rooted at s_k. All the points with distance 1 from s_k are exactly all the points that are visible. This tree is incrementally constructed and updated. Having determined the number N_n of x-monotone polygons, they pick a random integer $x \in \{1, \ldots, N_n\}$ and scan backward to generate the polygon corresponding to this number.

Enumerating Star-Shaped Polygons

We start with explaining how to enumerate all star-shaped polygons on $\mathcal{S}$. Obviously, a star-shaped polygon $\mathcal{P}$ is fixed once its kernel has been specified: For every point p that lies within the kernel, the order of the polygon's vertices is fixed because the vertices appear in sorted order around p. Also, this order is identical for every point interior to the kernel. Therefore, the kernels of two distinct star-shaped polygons share at most one edge, and the set of all kernels forms a valid partition of the convex hull[8] $\mathcal{CH}(\mathcal{S})$.

Thus, for enumerating all star-shaped polygons, it is sufficient to compute this partition of the convex hull. Necessarily, the arrangement induced by all lines

---

[8] Trivially, all kernels are confined to the convex hull.

$\ell(s_i, s_j)$, where $1 \leq i < j \leq n$, contains all the kernels. In general, several adjacent cells of the arrangement will belong to one kernel. The number of all lines $\ell(s_i, s_j)$ is bound by $\mathcal{O}(n^2)$, and therefore the arrangement contains at most $\mathcal{O}(n^4)$ cells, i.e., at most $\mathcal{O}(n^4)$ kernels. It can be computed in $\mathcal{O}(n^4)$. All the kernels can be constructed from the arrangement in time linear in its size by the following depth-first search:

1. Choose one cell and mark it as visited.
2. Create the star-shaped polygon $\mathcal{P}$ defined by this "active" cell.
3. For every edge e on the boundary of the active cell, do the following: If the neighboring cell bounded by e is part of the same kernel, then check its edges using $\mathcal{P}$. Otherwise, (i.e., if the neighboring cell belongs to another kernel) invert the order of the vertices that lie on the supporting line of e, and proceed with this other cell (and with the modified polygon).

Note that the neighboring cell belongs to the same kernel if and only if the two vertices defining e do not appear in consecutive order in $\mathcal{P}$. This algorithm is dubbed Star Arrange.

Since the method outlined above consumes $\mathcal{O}(n^4)$ space[9] independent of the actual number of star-shaped polygons, we also investigated the following method dubbed Star Universe which has a space complexity that depends on the output size: For each line $\ell(s_i, s_j)$, we compute the intersections with all other lines (defined by pairs of points of $\mathcal{S}$) and sort them according to their intersection parameters[10]. Since all kernels lie within $\mathcal{CH}(\mathcal{S})$, we can restrict $\ell(s_i, s_j)$ to the portion that lies within $\mathcal{CH}(\mathcal{S})$. For each intersection point p we keep track of the line ℓ which generated it. For the first[11] star-shaped polygon $\mathcal{P}$, we compute the midpoint of the first two intersections and sort the points of $\mathcal{S}$ around this midpoint. Then we process each intersection point p, starting with the second one, as follows: If the line ℓ associated with p coincides with an edge of $\mathcal{P}$ then we swap the points defining ℓ, thus updating $\mathcal{P}$. Otherwise, we skip p.

Algorithm Star Universe can be implemented with a time complexity of $\mathcal{O}(n^5)$ as opposed to $\mathcal{O}(n^4)$ for Star Arrange. However, the space requirement is reduced from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^2+k)$ space[12], where k again denotes the number of star-shaped polygons on $\mathcal{S}$.

---

[9] If all k resulting polygons are to be stored, the space complexity goes up to $\mathcal{O}(n^4 + n \cdot k)$.

[10] I.e., sort them according to their x-coordinate (or according to their y-coordinate if the line is vertical).

[11] Note that if one of the edges of $\mathcal{P}$ coincides with ℓ, we actually get two polygons.

[12] When applied to 50 points the process size of the arrangement-based code grew up to 236MB (and the program execution ended up in frequent swapping) whereas Star Universe consumed only the modest amount of 16MB of main memory. Due to swapping, the arrangement-based code was significantly slower than Star Universe, too.

Quick Star

Every point p which lies within $\mathcal{CH}(\mathcal{S})$ defines a star-shaped polygon. (If p lies on an edge or coincides with a vertex of the arrangement, up to four[13] star-shaped polygons are defined.) Therefore, the following simple method dubbed Quick Star generates every possible star-shaped polygon on $\mathcal{S}$ with positive probability: Choose a random point p within $\mathcal{CH}(\mathcal{S})$, and sort the points of $\mathcal{S}$ around p. Clearly, this approach requires $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

Steady Growth

Steady Growth is the first of the five heuristics we studied for the random generation of simple polygons. As initialization, Steady Growth randomly selects two points s_1, s_2. Let $\mathcal{S}_3 := \mathcal{S} \setminus \{s_1, s_2\}$. During the i-th phase (with $3 \leq i \leq n$), we

1. Choose one point $s_i \in \mathcal{S}_i$ at random such that no remaining point of $\mathcal{S}_{i+1} := \mathcal{S}_i \setminus \{s_i\}$ lies within $\mathcal{CH}(\mathcal{P}_{i-1} \cup \{s_i\})$.
2. Find an edge (v_k, v_{k+1}) of $\mathcal{P}_{i-1}$ that is completely visible from s_i, and replace it with the edges (v_k, s_i) and (s_i, v_{k+1}).

Note that a point $s_i \in \mathcal{S}_i$ which is suitable for Step 1 always exists. (For example take the point that lies closest to $\mathcal{CH}(\mathcal{P}_{i-1})$.)

It is less straightforward to guarantee that a suitable edge always exists in Step 2: A point p which lies outside a general polygon $\mathcal{P}$ need not see any edge of $\mathcal{P}$ completely, and the same holds for a point p within $\mathcal{P}$. However, if p lies outside $\mathcal{CH}(\mathcal{P})$, there must exist at least one edge which is completely visible from p. This can be shown by induction.

By using Steady Growth, one can compute a simple polygon in at most $\mathcal{O}(n^2)$ time, since all that has to be done during each phase is to compute all edges which are completely visible. (This can be done in $\mathcal{O}(n)$ time, cf. Joe and Simpson [JS87].) Note that selecting a suitable point s_i in Step 1 can be carried out in linear time, too. Unfortunately, Steady Growth does not generate every possible polygon on $\mathcal{S}$.

Space Partitioning

Space Partitioning recursively partitions $\mathcal{S}$ into subsets which have disjoint convex hulls. Let $\mathcal{S}'$ be a such a subset of $\mathcal{S}$. (Thus, $\mathcal{CH}(\mathcal{S}')$ does not contain any point of $\mathcal{S} \setminus \mathcal{S}'$.) When generating a polygon $\mathcal{P}$ we will guarantee that the intersection of $\mathcal{P}$ with $\mathcal{CH}(\mathcal{S}')$ consists of one single chain. The first point of this chain is denoted by s'_f, and its last point by s'_l. Note that both s'_f and s'_l are located on the boundary of $\mathcal{CH}(\mathcal{S}')$.

During the initial phase of the algorithm, we choose $s_f, s_l \in \mathcal{S}$ at random. Then the remaining points of $\mathcal{S}$ are partitioned into a left and a right set by the line

---

[13] Recall that we assume $\mathcal{S}$ to be in general position.

$\ell(s_f, s_l)$. For the general recursive call of the algorithm, consider a subset $\mathcal{S}'$ generated by this recursive subdivision, and let s'_f be its first point and s'_l its last point. (Recall that $\mathcal{CH}(\mathcal{S}')$ does not contain any other point of $\mathcal{S}$.) If s'_f and s'_l are the only points of $\mathcal{S}'$, then the line segment $\overline{s'_f s'_l}$ is output and the recursion is terminated. Otherwise, in order to split $\mathcal{S}'$ into two subsets $\mathcal{S}''$ and $\mathcal{S}'''$, we

1. Pick a point $s' \in \mathcal{S}'$ at random.
2. Select a random line ℓ through s' such that ℓ separates s'_f and s'_l. The line ℓ splits $\mathcal{S}'$ into two subsets $\mathcal{S}''$ and $\mathcal{S}'''$, where $\mathcal{S}''$ has s'_f as its first and s' as its last point. Similarly, $\mathcal{S}'''$ has s' as its first and s'_l as its last point.

In the worst case, this algorithm executes $\mathcal{O}(n)$ recursive calls, which results in $\mathcal{O}(n^2)$ time. However, if the recursive subdivision is somewhat balanced then we get a time behavior of roughly $\mathcal{O}(n \log n)$. Unfortunately, Space Partitioning does not generate every possible polygon on $\mathcal{S}$.

Permute & Reject

For Permute & Reject, we create a permutation of $\mathcal{S}$ and check whether this permutation corresponds to a simple polygon. If the polygon is simple then it is output; otherwise a new polygon is generated.

The actual running time of this method mainly depends on how many polygons need to be generated in order to encounter a simple polygon. Clearly, Permute & Reject produces all possible polygons with a uniform distribution.

2-Opt Moves

This approach was devised by Zhu et al. [ZSSM96]. First, it generates a random permutation of $\mathcal{S}$, which again is regarded as the initial polygon $\mathcal{P}$. Any self-intersections of $\mathcal{P}$ are removed by applying so-called 2-opt moves. Every 2-opt move replaces a pair of intersecting edges (v_i, v_{i+1}), (v_j, v_{j+1}) with the edges (v_i, v_j) and (v_{i+1}, v_{j+1}). In our application, in each phase of the algorithm one pair of intersecting edges is chosen at random and the intersection is removed.

Van Leeuwen and Schoone [vLS82] showed that at most $\mathcal{O}(n^3)$ many 2-opt moves need to be applied in order to obtain a simple polygon. Since we need a linear amount of time for removing an intersection, an overall time complexity of $\mathcal{O}(n^4)$ can be achieved.

2-Opt Moves will produce all possible polygons, but not with a uniform distribution: As explained in Zhu et al. [ZSSM96], there exist polygons which are obtained by a unique series of 2-opt moves, whereas other polygons may be obtained by several different series of 2-opt moves.

Incremental Construction & Backtracking

Finally, we studied an approach based on exhaustive search and backtracking which is akin to the work by Shuffelt and Berliner [SB94]. We start with a polygonal chain consisting of one randomly chosen point, and we randomly add one point after the other to it as long as the resulting chain remains simple. Backtracking has to be applied when a non-simple chain is encountered. Clearly, the main crux is to avoid any extensive backtracking.

In order to reduce backtracking, we keep an inventory of those edges which still are usable for completing the polygon. (Edges which are no longer usable get marked.) Initially, all edges of the complete graph on $\mathcal{S}$ are usable. When adding point s, and thus using some edge e, all the edges that intersect e are marked because they are no longer usable for completing the polygon. Furthermore, if a point is adjacent[14] to two other points that both have only two incident unmarked edges, we mark all the other edges incident upon that point. Clearly, backtracking is necessary if any of the following conditions is violated:

1. Each point that does not yet belong to the polygonal chain under construction has at least two incident unmarked edges. (Otherwise, it is impossible to add this point and still complete the polygon.)
2. At most one point adjacent to the point last added has only two incident unmarked edges.
3. Points that lie on the boundary of $\mathcal{CH}(\mathcal{S})$ appear in the polygonal chain in the same relative order as on the hull.

These conditions are checked in the same order as stated.

This algorithm produces every possible simple polygon with positive probability. Clearly, its efficiency depends on the amount of backtracking necessary.

1.4.2 Experimental Results

CPU-Time Consumption

We measured the CPU-time consumption of each algorithm (i.e., the algorithms for star-shaped polygons and the heuristics for the generation of simple polygons) when applied to random point sets (within the unit square) of the following cardinalities: 10, 25, 50, 100, 200, 300, 400 and 500. For each of these cardinalities we generated three independent sets. Our algorithms had to compute 50 polygons on each of these sets.

As expected, Star Universe and Star Arrange are only feasible for input sets with a small cardinality: Our attempts to run Star Universe on 100 points had to be aborted due to lack of main memory[15], and Star Arrange was not able to produce results on

---

[14] Two points are called "adjacent" if they are linked by an unmarked edge.

[15] 192MB of main memory and adequate swap space did not suffice.

50 points. Quick Star, however, seems well suited for larger point sets: computing a star-shaped polygon on 500 points takes roughly 72 milliseconds.

Two of the algorithms for the generation of simple polygons are not applicable to anything but extremely small point sets: In more than three weeks of running time we were not able to generate results for 25 points when using Permute & Reject or Incremental Construction & Backtracking. Clearly, those two algorithms are not suited for practical purposes.

Among the remaining four methods, Space Partitioning is significantly faster than the two other algorithms. Roughly, Space Partitioning takes about 62 milliseconds to compute a simple polygon on 500 points, whereas Steady Growth consumes about 23 seconds and 2-Opt Moves takes about 15 seconds. Steady Growth II, however, takes only 2.7 seconds. Thus, from a performance point of view, Space Partitioning is the candidate of choice. Note that Quick Star and Space Partitioning consume about the same amount of CPU time.

Number of Polygons

By using a modified version of Incremental Construction & Backtracking, we determined the number of simple polygons on groups of ten sets with 10 respectively 15 random points. For star-shaped polygons, we enumerated all polygons for groups of ten sets with 10, 15, 20, 25 and 50 points each by means of Star Universe. In our tests, all polygons which describe the same geometric figure were counted exactly once.

The gigantic number[16] of simple polygons on comparatively small sets of points also constitutes a practical problem when implementing any algorithm which is based on an enumeration of all polygons: integer overflows are very likely to occur[17], unless one switches to variable-length integer arithmetic. Also, the conventional random number generators (as contained in the standard libraries) can no longer be applied without modifications because they do not generate sufficiently large random numbers.

Quality Assessment

For each algorithm, we started with experimentally determining the ratio of the number of polygons generated and the total number of possible polygons.

For star-shaped polygons we have results for sets with 20 and 25 points. When generating 100,000 star-shaped polygons on 20 points with Quick Star, the mean percentage of polygons hit at least once was 91.439 with a minimum of 89.250 and a maximum of 94.679. When generating 10,000 polygons on 20 points we got 65.361 as mean, 59.141 as minimum and 69.241 as maximum. For 25 points and 100,000 polygons generated, we got a mean of 84.045, a minimum of 80.461 and

---

[16] E.g., more than 200,000 simple polygons on sets with 15 points.

[17] We experienced integer overflows when running our implementation of the algorithm by Zhu et al. [ZSSM96] for 1,000 points.

a maximum of 87.634, whereas the corresponding numbers for 10,000 polygons are 51.769, 44.830, and 55.085. Since Quick Star is capable of producing all possible star-shaped polygons it does not come as a big surprise that the hit rate goes up as the number of polygons generated is increased.

In the case of simple polygons, we tested 10 groups of sets with 10, 15 and 100 random points each. For sets with 10 or 15 points, we have results for 10,000 and 100,000 polygons generated on each set. For sets with 100 points we have results for 100,000 polygons generated.

Among the methods for simple polygons, one method is significantly worse than the others: Incremental Construction & Backtracking hits less than half of all possible simple polygons at least once when generating 10,000 polygons on 10 points. As could be expected, Permute&Reject exhibits an optimal hit rate of 100%. For the three algorithms with a modest CPU-consumption, the results are good for 2-Opt Moves, acceptable for Steady Growth (II), but rather poor for Space Partitioning. Note that the results improved when generating 100,000 polygons instead of 10,000 polygons: 2-Opt moves generates almost all polygons, Steady Growth (II) lies around or above 90 percent, and Space Partitioning generates about between 80 and 90 percent of all possible polygons. In both tests the distribution of the polygons turned out to be highly non-uniform, though.

However, when generating 100,000 polygons on 100 points all three algorithms 2-Opt Moves, Steady Growth (II) and Space Partitioning behaved optimally: They all generated exactly 100,000 different polygons! It is likely that this result is due to the fact that there exists an enormous number of simple polygons on 100 points. (We encountered sets with 20 points which already allowed more than three million simple polygons.) On one hand, our tests suggest that a user of any of these three algorithms need not worry about repeatedly generating the same "random" polygons when dealing with 100 or more points. On the other hand, the distribution of the polygons generated should not be expected to be (close to) uniform. (Any further statistical analysis of the distributions of the polygons generated had to be abandoned due to hardware constraints imposed on the CPU-time consumption and the available main memory and disk space.)

1.4.3 Conclusion

We present five heuristics for the random generation of simple polygons. Three of these heuristics, namely 2-Opt Moves, Steady Growth (II) and Space Partitioning, are suited for practical purposes. However, we experienced a clear trade-off between the quality of the heuristic and the running time. Thus, when the CPU-consumption is not at a premium, one can afford to generate a large variety of polygons by 2-Opt Moves. In order to achieve maximum speed Space Partitioning would be the method of choice. Steady Growth (II) is slightly faster than 2-Opt Moves but generates a less rich set of polygons.

It is very likely that the same trade-offs also hold true for the generation of random polygons on sets of 100 or more points. However, the class of simple polygons on such point sets is rich enough and the power of 2-Opt Moves, Steady Growth

(II) and Space Partitioning is large enough that any of them can be expected to yield fairly good results. In particular, it is quite unlikely that a simple polygon will be generated repeatedly by any of these three heuristics. In our experimental analysis, constraints imposed by our computing equipment on the available CPU time, memory, and on the disk space clearly turned out to be the limiting factor.

For the random generation of star-shaped polygons we present a fast heuristic, Quick Star. It has about the same characteristics as Space Partitioning. Unfortunately, the enumeration algorithms Star Arrange and Star Universe do not work for sets with more than, say, 50 points since they require far too much space.

1.5 Basic Concepts and Notation

In this section, we will outline the basic concepts and notation that are used in the thesis. Basic concepts are concepts that are important for more than one algorithm, e.g., we cannot deal with polygon generation without having defined what a polygon is. More advanced concepts that are used only for single algorithms, such as arrangements for Star Arrange, will be introduced later when needed. The definitions were mainly taken from the textbook by Preparata and Shamos [PS85].

As mentioned above, we restrict ourselves to studying points in two-dimensional Euclidean space. A *point*, denoted by p, is specified by a pair of coordinates (x, y) (Additional points will be denoted by q and r). We consider only points that lie within the *unit square* $[0,1]^2$. This is sufficient since we can apply a continuous, strictly monotone bijection between $[0,1]$ and $\mathbb{R}$ coordinate-wise. Thus, the study of $[0,1]^2$ immediately yields results for $\mathbb{R}^2$. The set of n points forming the polygon's vertex set is denoted by $\mathcal{S}$, and its members (points) are given as s_1 through s_n. When describing an algorithm, the points are normally numbered according to the order in which the algorithm adds them to the polygon. When tracing an algorithm, however, the points will always be sorted lexicographically, i.e., first according to their x-coordinates, and second according to their y-coordinates for points with the same x-coordinate. A two-dimensional vector $\mathfrak{w} = \binom{x}{y}$ is determined by its x-coordinate and its y-coordinate. Note that each point corresponds to a vector from the origin to its location.

Given two distinct points p and q, the linear combination

$$\alpha p + (1-\alpha) q \qquad (\alpha \in \mathbb{R}) \tag{1.1}$$

describes a *line*. It is denoted by $\ell(p, q)$. By adding the constraint $0 \leq \alpha \leq 1$, we obtain the *line segment* between p and q, which is denoted by $\overline{pq}$.

A region $\mathcal{R}$ is *convex* if, for any two points p and q in $\mathcal{R}$, the segment $\overline{pq}$ is entirely contained in $\mathcal{R}$. In Fig. 1.3, the region in Fig. 1.3a is convex, since it will contain the segment $\overline{pq}$ for any choice of p and q within $\mathcal{R}$. However, the region in Fig. 1.3b is not convex: For our choice of p and q, the line segment $\overline{pq}$ is not entirely contained in $\mathcal{R}$.

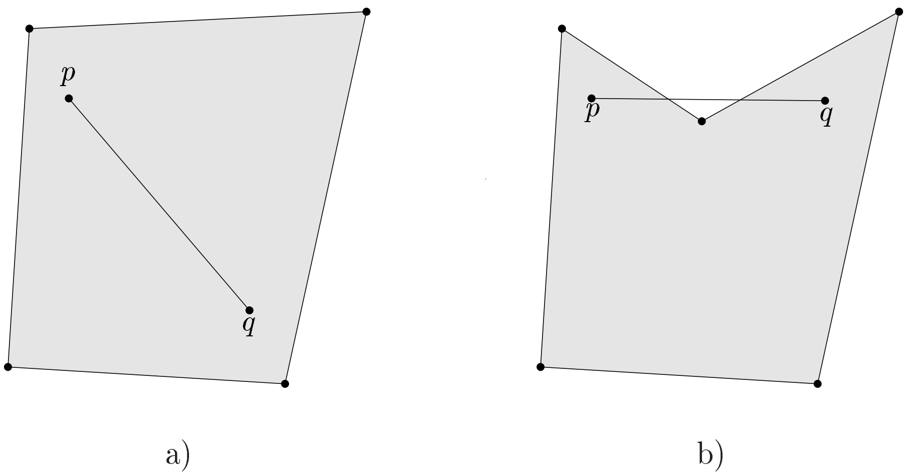


Figure 1.3: A convex and a non-convex region.

A *polygon* $\mathcal{Q}$ is defined by a finite set of line segments such that every segment extreme is shared by exactly two edges and no subset of edges has the same property. The first property of this definition guarantees that there does not exist a point in which more than 2 segments coincide (a curve violating this condition is the curve of Fig. 1.4a), whereas the second condition ensures that the polygon is connected (cf. the curve of Fig. 1.4b). A polygon according to this definition is given in Fig. 1.4c. As usual, segments are called *edges* and their extremes are called *vertices* of the polygon. The vertices are denoted by v; the edge e defined by vertices v_i and v_j is denoted by (v_i, v_j). A polygon with n vertices[18] is called an n-gon. Throughout the thesis, $\mathcal{Q}$ denotes a general polygon in $[0,1]^2$, whereas $\mathcal{P}$ denotes a polygon on $\mathcal{S}$; in our algorithms, the polygon obtained after executing phase k of the algorithm is denoted by $\mathcal{P}_k$.

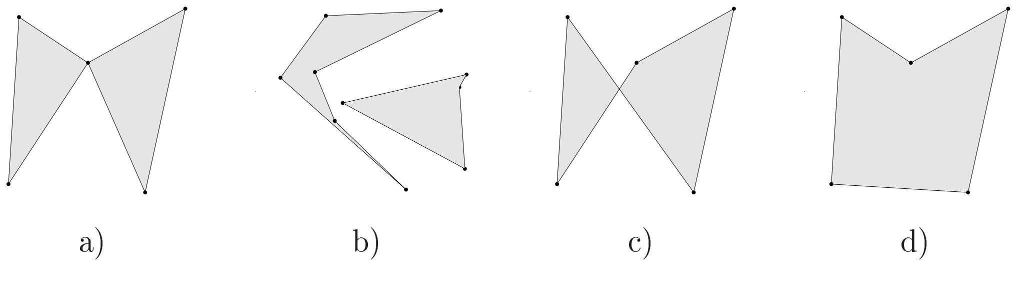


Figure 1.4: Illustration of the definition of a polygon.

A polygon is called *simple* if there is no pair of non-consecutive edges sharing a point. (The polygon of Fig. 1.4d is simple.) By the Jordan curve theorem, cf. [PS85], a simple polygon partitions the plane into two disjoint regions, the bounded interior and the unbounded exterior. We always consider the bounded interior of the polygon to be part of the polygon. Thus, a polygon consists of a set of vertices,

---

[18] Note that the number of vertices is the same as the number of edges.

a set of edges and its (bounded) interior region. The *boundary* of $\mathcal{P}$ consists of the set of vertices and the set of edges; it is denoted by $\partial\mathcal{P}$. Throughout the thesis, we assume that all polygons are specified in *counterclockwise* (CCW) order, i.e., the interior of a polygon always lies to the left of each oriented edge (v_i, v_{i+1}), cf. Fig. 1.5. (Note, however, that the algorithms may also generate a polygon in clockwise (CW) orientation nevertheless.) A *triangle* is a simple polygon with exactly three vertices. The triangle with vertices p,q and r is denoted by $\triangle(p, q, r)$.

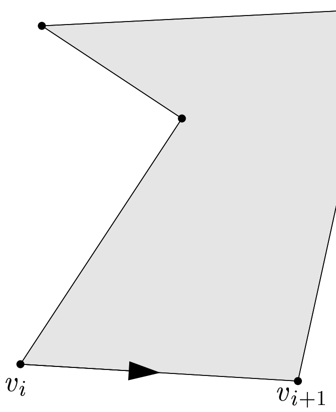


Figure 1.5: A polygon in CCW order.

A simple polygon $\mathcal{Q}$ is *convex* if it forms a convex set. A simple polygon $\mathcal{Q}$ is *star-shaped* if there exists a point p not external to $\mathcal{Q}$ such that for all points q of $\mathcal{Q}$ the line segment $\overline{qp}$ lies entirely within $\mathcal{Q}$. Note that each convex polygon is also star-shaped. The locus of the points p having the above property is the *kernel* of $\mathcal{Q}$. The kernel of a star-shaped polygon is always convex, cf. [PS85]. A star-shaped polygon and its kernel (with p lying in the kernel) are depicted in Fig. 1.6.

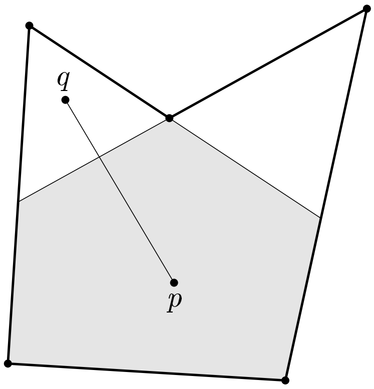


Figure 1.6: A star-shaped polygon and its kernel.

The *convex hull* of a set of points $\mathcal{S}$, $\mathcal{CH}(\mathcal{S})$, is the the smallest convex region[19] containing $\mathcal{S}$. Equivalently, $\mathcal{CH}(\mathcal{S})$ is the intersection of all convex regions containing $\mathcal{S}$. Yet another equivalent definition states that $\mathcal{CH}(\mathcal{S})$ equals the union of all triangles determined by the points of $\mathcal{S}$. For a point set $\mathcal{S}$, the convex hull $\mathcal{CH}(\mathcal{S})$ is a convex polygon, whose boundary is denoted by $\partial\mathcal{CH}(\mathcal{S})$. Similar to $\mathcal{CH}(\mathcal{S})$, the convex hull of a polygon $\mathcal{CH}(\mathcal{P})$ is the convex hull of its vertex set. Fig. 1.7 shows a set $\mathcal{S}$ of 10 points and $\mathcal{CH}(\mathcal{S})$.

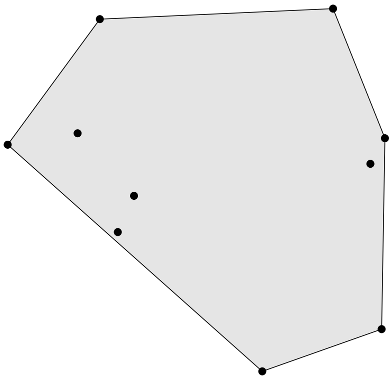


Figure 1.7: A set $\mathcal{S}$ and its convex hull $\mathcal{CH}(\mathcal{S})$.

A *polygonal chain*[20] $\mathcal{C} = (v_1, \dots, v_k)$ is defined by a vertex set $\{v_1, \dots, v_k\}$ and an edge set $\{(v_i, v_{i+1}) : i = 1, \dots, k-1\}$. A polygonal chain $\mathcal{C}$ is said to be *strictly monotone*[21] with respect to a straight line ℓ if a line ℓ' orthogonal to ℓ intersects $\mathcal{C}$ in at most one point. (I.e., $\ell' \cap \mathcal{C}$ is either empty or a single point.) A chain $\mathcal{C}$ is *monotone*, if $\ell' \cap \mathcal{C}$ has at most one connected component, i.e., if it is either empty, a single point or a single line segment.

A simple polygon is said to be *monotone* if its boundary can be decomposed into two chains which are monotone with respect to the same straight line. In the special case of x-monotone[22] polygons, these two chains are called *top* and *bottom chain*, where the top chain lies "above"[23] the bottom chain. The bottom chain is denoted by $\mathcal{C}_b$ and the top chain by $\mathcal{C}_t$. Fig. 1.8 shows an x-monotone polygon with its top and bottom chains.

---

[19] In the textbook by Preparata and Shamos [PS85], the convex hull is defined as the *boundary* of the smallest convex region containing $\mathcal{S}$.

[20] In Preparata and Shamos [PS85], the definition of a polygonal chain is based on the notion of a planar straight line graph (PSLG). Since we do not use the concept of a planar straight line graph in this thesis, we gave an equivalent but different definition.

[21] The distinction between *strictly monotone* and *monotone* was taken from the textbook by O'Rourke [O'R94]. The concept used by Preparata and Shamos [PS85] is what we call *strictly monotone.*

[22] I.e., polygons which are monotone with respect to the x-axis of the coordinate system.

[23] I.e., it lies in positive y-direction with respect to the bottom chain.

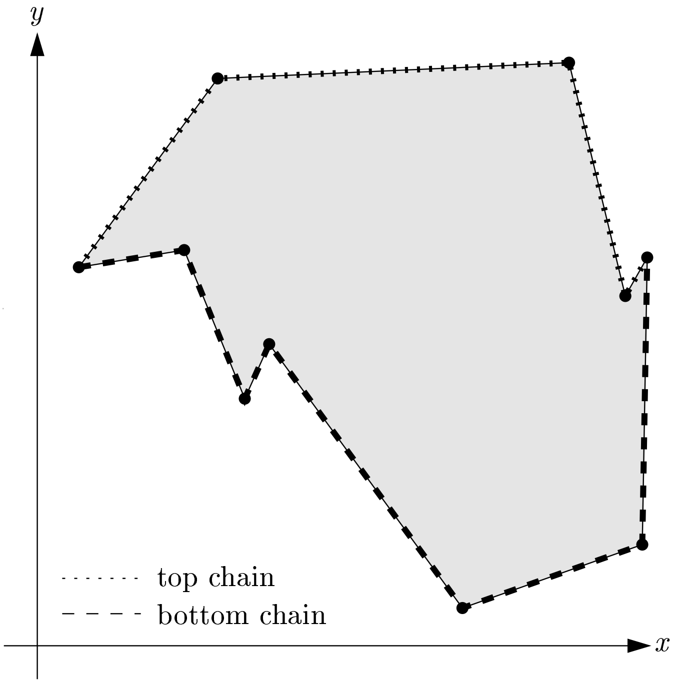
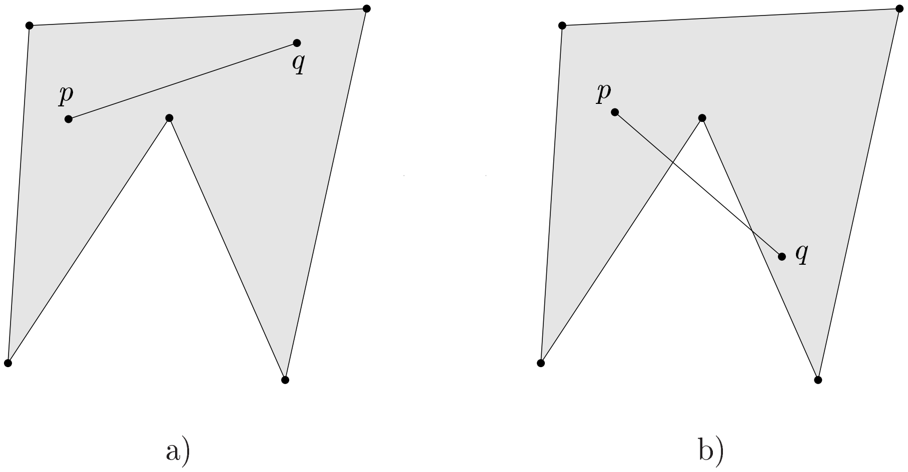


Figure 1.8: An x-monotone polygon and its top and bottom chain.

Two points p and q are *mutually visible* with respect to some polygon $\mathcal{Q}$, if the line segment $\overline{pq}$ does not intersect its boundary $\partial\mathcal{Q}$. Note that p and q may be either interior or exterior to $\mathcal{Q}$. In Fig. 1.9a, the points are mutually visible, whereas the points in Fig. 1.9b are not mutually visible. Similarly, a line segment $\overline{qr}$ is *completely visible*[24] from p if every point on $\overline{qr}$ is visible from p.



a) b)

Figure 1.9: Visibility of two points p and q with respect to a polygon $\mathcal{Q}$.

We assume that all point sets are in general position, i.e., that no three points

---

[24] Partial visibility is defined in a similar way.

are collinear and no four points are cocircular.

1.5.1 Terms of Complexity

When analyzing the complexity of an algorithm, we use a concept introduced by Knuth [Knu76]: $\mathcal{O}(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants c and n_0 with $|g(n)| \leq c \cdot f(n)$ for all $n \geq n_0$. Thus, $\mathcal{O}(f(n))$ is used to indicate functions whose asymptotic growth is equal to or less than $f(n)$ times some constant c. Therefore, this terminology can be used to describe upper bounds.

$\Omega(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants c and n_0 with $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$. Since $\Omega(f(n))$ indicates functions at least as large as some constant c times $f(n)$, it is the concept analogous to $\mathcal{O}(f(n))$ for describing lower bounds.

Finally, $\Theta(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants c_1, c_2 and n_0 with $c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$. Thus, it indicates functions of the same order as $f(n)$. Hence, it is normally used for "optimal" algorithms.

Unless stated otherwise, all the complexities given throughout the thesis are worst-case complexities: Worst-case complexity is the maximum of a measure of performance of a given algorithm over all possible input data of a given size. We will discuss both *time* and *space* complexity.

When stating an algorithm in pseudo code, the parameters given to the algorithm may be modified in the algorithm, but the scope of this modification is the algorithm. If we modify a parameter and want that this modification is visible to the calling process, the parameter is preceded by "*VAR*" in the procedure declaration[25].

1.6 Sample Point Sets

Throughout the thesis, we will use two point sets to illustrate (and trace) the algorithms. The first sample set, denoted by $\mathcal{S}1$, is depicted in Fig. 1.10. It consists of five points; their coordinates are given in Table 1.1. It will be used for showing general principles and short traces.

The second sample set which is denoted by $\mathcal{S}2$, consists of ten points which were created at random. Sample Set $\mathcal{S}2$ is depicted in Fig. 1.11 and the coordinates of its points are given in Table 1.2. We will use this set for detailed traces of the algorithms presented in this thesis.

---

[25] Note that this notation is similar to the concept for passing parameters used in programming languages such as Pascal or Modula.

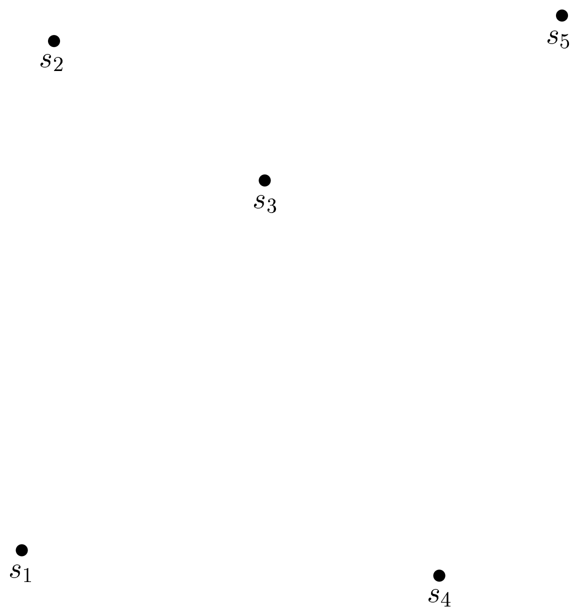


Figure 1.10: Sample Set $\mathcal{S}1$.

	x-coordinate	y-coordinate
s_1	0.120807	0.129406
s_2	0.226119	0.843143
s_3	0.478462	0.673851
s_4	0.735386	0.092030
s_5	0.916021	0.916660

Table 1.1: Coordinates of the points of Sample Set $\mathcal{S}1$.

	x-coordinate	y-coordinate
s_1	0.110391	0.606917
s_2	0.265468	0.632202
s_3	0.314795	0.885048
s_4	0.354639	0.413326
s_5	0.390602	0.493621
s_6	0.675076	0.104805
s_7	0.832020	0.908315
s_8	0.914863	0.564527
s_9	0.939811	0.198284
s_{10}	0.947087	0.621259

Table 1.2: Coordinates of the points of Sample Set $\mathcal{S}2$.

Figure 1.11: Sample Set $\mathcal{S}2$.

Chapter 2

Algorithms

2.1 Bouncing Vertices

The algorithm Bouncing Vertices is due to O'Rourke and Virmani [OV91]. It differs from all the other algorithms in this thesis as it generates a random polygon by moving the vertices of the polygon, whereas all the other algorithms operate on a fixed vertex set. The algorithm works as follows:

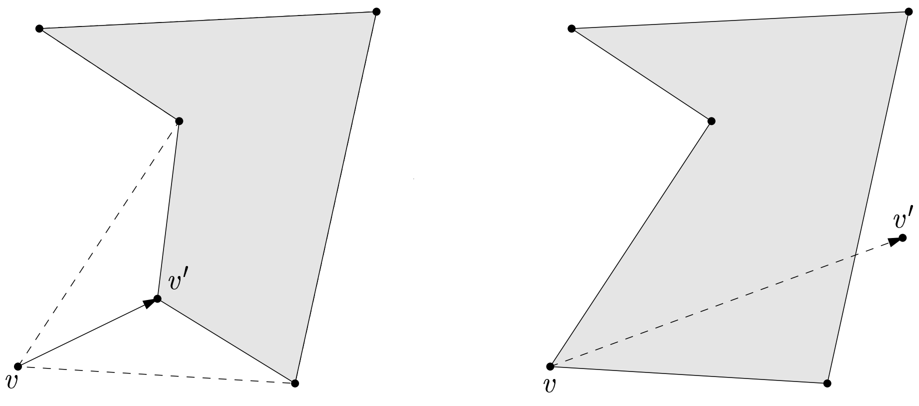
1. Generate an arbitrary simple polygon on the given point set $\mathcal{S}$ [1].
2. Take each vertex and do the following:
    - Choose a random vector $\mathfrak{w}$ and let $v' = v + \mathfrak{w}$.
    - The resulting vertex v' must still be within $[0,1]^2$, and the resulting polygon must be simple.
    - If one of these conditions is violated, choose another random vector $\mathfrak{w}$. (This update of a vertex is depicted in Fig. 2.1: Fig. 2.1a shows a valid move and Fig. 2.1b shows an invalid move where self-intersections occur.)
3. Repeat Step 2 as many times as considered necessary.

We chose to generate the initial polygon by performing the sorting step of Graham's convex-hull algorithm: Take the point with the minimum x-coordinate (if there are several points with minimum x-coordinate, take the one among them with the smallest y-coordinate) and sort all the remaining points in angular order around this point. Obviously, this yields a simple, star-shaped polygon.

Let k be the number of phases specified by the user. In this context, "phase" denotes one iteration of the polygon, whereas in the rest of the thesis, a "phase" includes all the steps necessary to add one point to the polygon. Then we can state Algorithm BouncingVertices as follows:

---

[1] Clearly, we do not need a set of input points for Bouncing Vertices, only the desired number of vertices. However, since all the other algorithms operate on a given set of input points, we also require a set of input points with which we start the algorithm.



a) b)

Figure 2.1: Moving vertex v.

```
Algorithm BouncingVertices($\mathcal{S}$,k)
Sort the points of $\mathcal{S}$ around $s_1$ and init the polygon.
$n := |\mathcal{S}|$.
for $j = 1$ to $k$ do
   for $i = 1$ to $n$ do
      repeat
         Select a random vector $\mathfrak{w}$.
         Replace vertex $s_i$ with $s'_i := s_i + \mathfrak{w}$.
      until $s'_i \in [0,1]^2$ and the new polygon is simple.
```

2.1.1 Complexity of the Algorithm

First, we need to initialize the polygon, which requires sorting the points of $\mathcal{S}$ around point s_1. Clearly, this can be done in $\mathcal{O}(n \log n)$ time. For each vertex moved, two tests are necessary:

- First, the resulting vertex v' must be tested whether it lies within the unit square $[0,1]^2$. Clearly, this can be done in constant time.
- Second, one must test whether the resulting polygon is simple. Let v_i be the vertex that has been moved. Then, the test for simplicity requires testing whether the two new edges (v_{i-1}, v'_i) and (v'_i, v_{i+1}) intersect any of the remaining $n-2$ edges. With a straightforward approach, this can be done in linear time.

We may have to generate several random vectors until a suitable one is encountered. Naturally, a huge number of vectors to be tried will dramatically slow down the algorithm. Therefore, we let $\vartheta(n)$ denote a function which reflects the number of vectors that have to be generated until a valid one is encountered. Thus, we have a complexity of $\mathcal{O}(n^2 + n \cdot \vartheta(n))$ per phase, and with k phases the algorithm needs $\mathcal{O}(k(n^2 + n \cdot \vartheta(n)))$ time. The space requirement of the algorithm is bound by $\mathcal{O}(n)$, because it is necessary to store the polygon, but no additional space (except for one additional vertex and a vector) is needed.

2.1.2 Trace of the Algorithm

We illustrate how Bouncing Vertices works when applied to the points of $\mathcal{S}1$. Note that we depicted only one phase of the algorithm. In the first figure of our trace, Fig. 2.2, the initial polygon after the sorting is depicted. In the figures illustrating the trace, the polygon resulting from moving vertex v_i is filled with a dark gray, whereas the polygon as it was before the move is outlined. Points obtained as (unsuccessful) candidates for moving the vertex are also shown, and the edges implied by them are dashed. (All figures were obtained by actually using our implementation of the algorithm.)

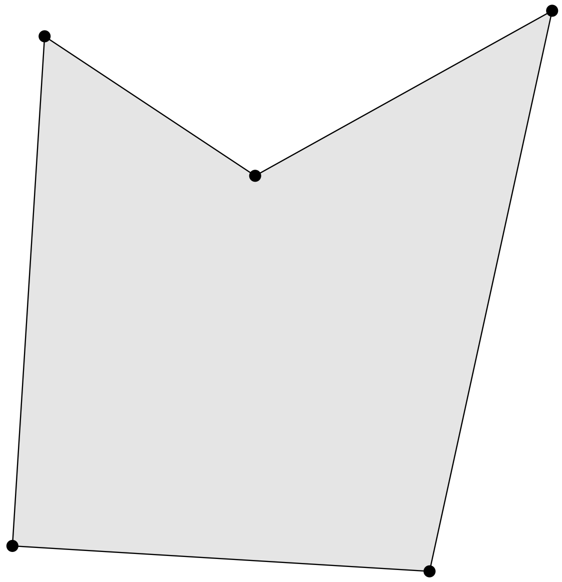


Figure 2.2: The initial polygon after the sorting.

For the first step, we have to move vertex v_1. As can be seen in Fig. 2.3, the first random move is successful.

For the next vertex, v_2, the algorithm does not succeed as fast as for v_1: Only the third random move yields a polygon that does not self-intersect, cf. Fig. 2.4. We have depicted the two unsuccessful moves as well: The dashed lines are the edges that would have been inserted when using one of the points generated by an unsuccessful move. As can be seen in Fig. 2.4, both unsuccessful moves would have caused self-intersections. Vertex v_3 again is successfully iterated by the first random move. This is shown in Fig. 2.5.

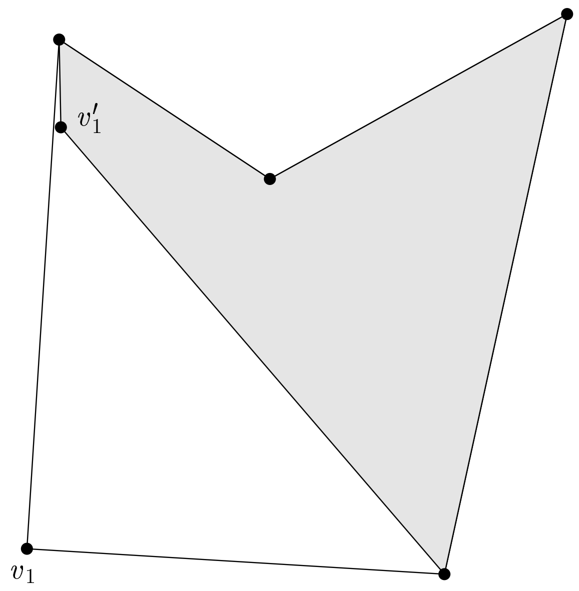


Figure 2.3: Moving vertex v_1.

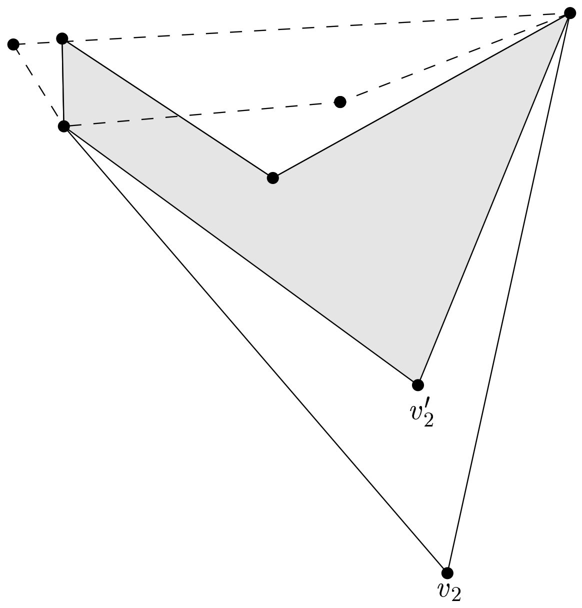


Figure 2.4: Moving vertex v_2.

For vertex v_4, the situation is similar to vertex v_2: Again, two unsuccessful attempts to move the point are made until the third random move succeeds. The two unsuccessful attempts and the (successful) vertex v_4' are depicted in Fig. 2.6.

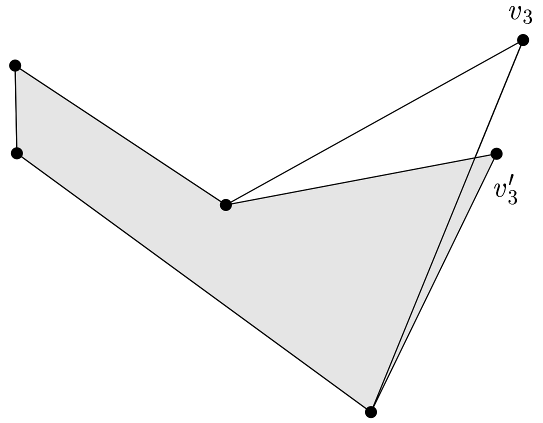


Figure 2.5: Moving vertex v_3.

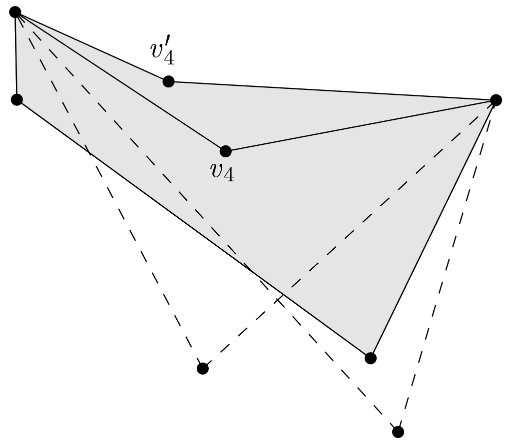


Figure 2.6: Moving vertex v_4.

Finally, the only vertex not moved already is v_5. In order to move it, five random moves have to be computed, the first four of which fail. But eventually the fifth random move yields a valid vertex v_5'. In Fig. 2.8, the resulting polygon after one phase of Bouncing Vertices is depicted.

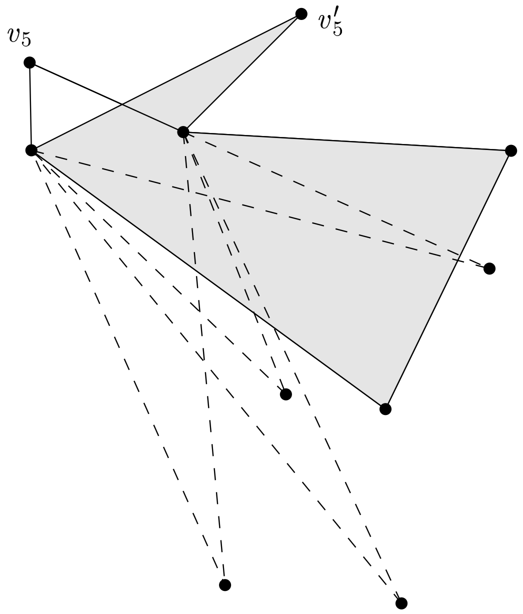


Figure 2.7: Moving vertex v_5.

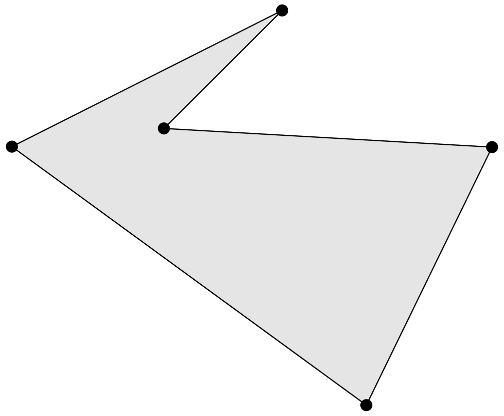


Figure 2.8: The final polygon after the first phase.

2.2 x-Monotone Polygons

The algorithm by Zhu et al. [ZSSM96] for the random generation of x-monotone polygons is based on counting all x-monotone polygons on a given set $\mathcal{S}$ of vertices.

In this section, we will present their results but omit the proofs. For the proofs, please refer to the original work by Zhu et al. [ZSSM96]. (Our notation follows the notation by Zhu et al. [ZSSM96] as close as possible.) The algorithm determines for each point s all the polygons where s lies on the top chain and all polygons where s lies on the bottom chain. Throughout this section, we assume that the points in $\mathcal{S}$ are lexicographically sorted, i.e., the first point s_1 of $\mathcal{S}$ is the one with minimum x-coordinate. Let $\mathcal{S}_k$ denote the set of the first k points in $\mathcal{S}$, i.e., $\mathcal{S}_k := \{s_1, \ldots, s_k\}$.

Clearly, the first point s_1 and the last point s_k of any x-monotone polygon on $\mathcal{S}_k$ must appear on both the bottom and the top chain. Since the point s_{k-1} must lie on either the top or the bottom chain, either the top or the bottom chain must contain edge (s_{k-1}, s_k). Let $\mathcal{T}_k$ be the set of x-monotone polygons that have edge (s_{k-1}, s_k) on their top chain and $\mathcal{B}_k$ the set of x-monotone polygons that have (s_{k-1}, s_k) on their bottom chain. Further, let $T_k := |\mathcal{T}_k|$ and $B_k := |\mathcal{B}_k|$. (In the original paper this numbers are denoted by $T(k) := |\mathcal{T}(k)|$ and $B(k) := |\mathcal{B}(k)|$, respectively.) As shown by Zhu et al., the following holds:

Lemma 2.1 For any point set $\mathcal{S}_k$ with $k > 2$, the number N_k of monotone polygons[2] with vertex set $\mathcal{S}_k$ is given by

$$N_k = T_k + B_k. \tag{2.1}$$

In addition to the concept of visibility introduced in Section 1.5, the authors use the concept of *above-visible* and *below-visible*: A point s_i is above visible from s_k if $i < (k-1)$ and s_i is above the line $\ell(s_j, s_k)$ for all points s_j with $i < j < k$. The concept of above-visibility is illustrated in Fig. 2.9: For point s_7, point s_5 is above-visible because it lies above $\ell(s_6, s_7)$. Similarly, s_3 is above-visible, since it lies above the lines $\ell(s_4, s_7)$, $\ell(s_5, s_7)$ and $\ell(s_6, s_7)$. In the figure, the visibility lines are drawn solid, whereas the lines to be checked (for determining visibility) are drawn dashed. Similarly, s_i is below-visible from s_k if $i < (k-1)$ and s_i is below $\ell(s_j, s_k)$ for all points s_j with $i < j < k$. The set of all points that are above-visible from s_k is denoted by $\mathcal{V}_k^T$ and the set of below-visible points is denoted by $\mathcal{V}_k^B$. (In the original paper these sets are denoted by $V_T(k)$ and $V_B(k)$, respectively.) In Fig. 2.9, the set of below-visible points of s_7 is empty since no point in $\mathcal{S}_7$ lies below line $\ell(s_6, s_7)$.

An edge (s_i, s_k) may lie on the bottom chain if s_i is below visible from s_k, because all the points s_j with $i < j < k$ must lie on the top chain if (s_i, s_k) is on the bottom chain. Further, if s_i and s_k lie on the bottom chain and all points s_j with $i < j < k$ lie on the top chain, then the path from s_i to s_{i+1} is fixed and it is treated as an edge (s_i, s_{i+1}) which lies on the bottom chain. Thus, the following lemma holds.

Lemma 2.2 The number of polygons in $\mathcal{T}_k$ that contain edge (s_i, s_k), for $s_i \in \mathcal{V}_k^B$, is B_{i+1}. The number of polygons in $\mathcal{B}_k$ that contain edge (s_i, s_k), for $s_i \in \mathcal{V}_k^T$, is T_{i+1}.

---

[2] In the work of Zhu et al. [ZSSM96] this number is denoted by $N(k)$.

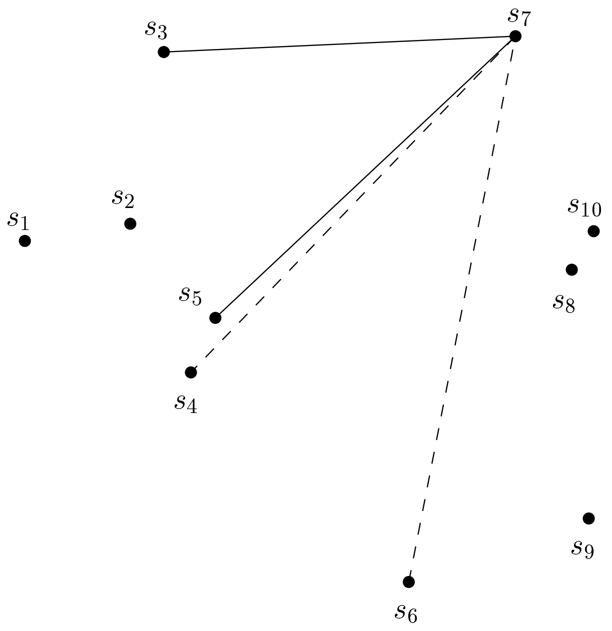


Figure 2.9: Above and below-visible points from s_7.

This leads to the following theorem.

Theorem 2.1 For any point set $\mathcal{S}_k$, with $k > 2$,

$$T_k = \sum_{s_i \in \mathcal{V}_k^B} B_{i+1}, \tag{2.2}$$

and

$$B_k = \sum_{s_i \in \mathcal{V}_k^T} T_{i+1}. \tag{2.3}$$

Thus, Zhu et al. start with $T_2 = B_2 = 1$ and then use recurrences to determine T_i and B_i for $i := 3$ to n.

For the algorithm, it is necessary to determine the sets of above-visible and below-visible points. $\mathcal{V}_k^T$ and $\mathcal{V}_k^B$ are computed by applying a simplified version of the algorithm by Hershberger [Her89]. For computing the set of above-visible points, they use the following algorithm: First, they consider the monotone chain $\mathcal{C}_k$ with vertices $\mathcal{C}_k = (s_1, s_2, \ldots, s_k)$ and compute the shortest paths in the plane above $\mathcal{S}_k$ from s_k to each s_i with $i \leq k$. In this way, they obtain the shortest path tree rooted at s_k. The above-visible set $\mathcal{V}_k^T$ is exactly the set of children of s_k in the tree.

Then, they incrementally compute $\mathcal{V}_k^T$ from $\mathcal{V}_{k-1}^T$ by computing the tree rooted at s_k from the tree rooted at s_{k-1}. This is done by algorithm Algorithm MakeTop.

Algorithm MakeTop(i, k, *VAR lastSib*)

```
while $s_{i.upc} \neq nil$ and $s_k$ is above $\ell(s_{i.upc}, s_i)$ do
    MakeTop($i.upc, k, lastSib$).
    $i.upc := i.upc.sib$.
$lastSib.sib := i$.
$lastSib := i$.
```

As a data structure they use a binary tree where each node has a pointer to its uppermost child (referenced to as *node.upc*) and a pointer to its next sibling (referenced as *node.sib*). Further, *lastsib* is used to keep track of the last node updated.

For computing the above-visible points for s_k, the algorithm is called with parameters $k-1$, k and the tree for s_{k-1}. Initially, the tree contains a single record that has no child and no sibling.

Having presented an efficient method for computing the visibility sets, they give the algorithm for computing an x-monotone polygon. It extends a subsequence from right to left in order to generate a polygon. Before the algorithm is invoked, the number of polygons must have been determined.

Algorithm xmonotonePolygon($\mathcal{S}$)

```
Pick $x \in [1, N_n]$ at random.
Add $s_n$ to topChain.
Add $s_n$ to bottomChain.
if $x \leq T_n$ then
    Add $s_{n-1}$ to topChain.
    generateTop($n, x$).
else
    Add $s_{n-1}$ to bottomChain.
    generateBottom($n, x$).
if the last element of topChain $\neq s_1$ then
    Add $s_1$ to topChain.
if the last element of bottomChain $\neq s_1$ then
    Add $s_1$ to bottomChain.
```

Algorithm generateBottom completes a polygon sequence in which s_{k-1} is on the bottom chain and s_k is on the top chain. (I.e., (s_{k-1}, s_k) lies on the bottom chain in the completion polygon.) For generating the polygon at random, they select a random integer $x \in \{1, \ldots, N_n\}$ and determine which partial sum is greater than or equal to x. Since the computation of the sum is started with the high indices, each point is considered at most once.

2.2.1 Complexity of the Algorithm

The algorithm consists of two parts: First, the number of x-monotone polygons is computed and then this is used for randomly generating a polygon. The first part requires $\mathcal{O}(n)$ space, since a linear number of values for B and T is stored, and

Algorithm generateTop(k, x)

```
if $k \leq 2$ then
    return.
$sum := 0$.
$i := k - 1$.
$curTop := k - 1$.
repeat
    $i := i - 1$.
    if $s_i$ below $\ell(s_{curTop}, s_k)$ then
        $curTop := i$.
        $sum := sum + B_{i+1}$.
until $x \leq sum$.
Add $s_i$ to $bottomChain$.
Add $s_{k-2}, s_{k-3}, \ldots, s_{i+1}$ to $topChain$.
$k := i + 1$.
$x := x - (sum - B_{i+1})$.
generateBottom$(k, x)$.
```

Algorithm generateBottom(k, x)

```
if $k \leq 2$ then
    return.
$sum := 0$.
$i := k - 1$.
$curBottom := k - 1$.
repeat
    $i := i - 1$.
    if $s_i$ above $\ell(s_{curBottom}, s_k)$ then
        $curBottom := i$.
        $sum := sum + T_{i+1}$.
until $x \leq sum$.
Add $s_i$ to $topChain$.
Add $s_{k-2}, s_{k-3}, \ldots, s_{i+1}$ to $bottomChain$.
$k := i + 1$.
$x := x - (sum - T_{i+1})$.
generateTop$(k, x)$.
```

in each phase the visibility tree is of linear size. The time complexity is bound by $\mathcal{O}(K)$ with $n < K < n^2$, where K denotes the number of edges in the visibility graph of the x-monotone chain.

The second part of the algorithm, the generation of the x-monotone polygon, runs in linear time and with linear space. Thus, in total, Zhu et al. compute a x-monotone polygon in $\mathcal{O}(K)$ time, thereby using linear space.

2.2.2 Trace of the Algorithm

We illustrate the algorithm on the points of set $\mathcal{S}2$. The vertex visibilities and the values for $\mathcal{V}^T$ and $\mathcal{V}^B$ are given in Table 2.1. Further, the vertex visibilities are shown in Fig. 2.10, where above-visible points are connected by dashed lines and below-visible points by dotted lines. We assume that the random number chosen is 1 at the start of Algorithm xmonotonePolygon. Since $T_{10} = 0$, the else-branch is executed. Thus, s_{n-1} is added to *bottomChain*, and generateBottom is called with $n = 10$ and $x = 1$.

	$\mathcal{V}_i^T$	$\mathcal{V}_i^B$	T_i	B_i
s_1	-	-	-	-
s_2	-	-	1	1
s_3	s_1	-	0	1
s_4	-	s_1,s_2	2	0
s_5	s_3	-	0	2
s_6	s_3	s_1,s_4	3	2
s_7	s_3,s_5	-	0	5
s_8	-	s_6	5	0
s_9	-	s_6,s_7	5	0
s_{10}	s_7,s_8	-	0	10

Table 2.1: $\mathcal{V}^T$, $\mathcal{V}^B$, T and B for the points of $\mathcal{S}2$.

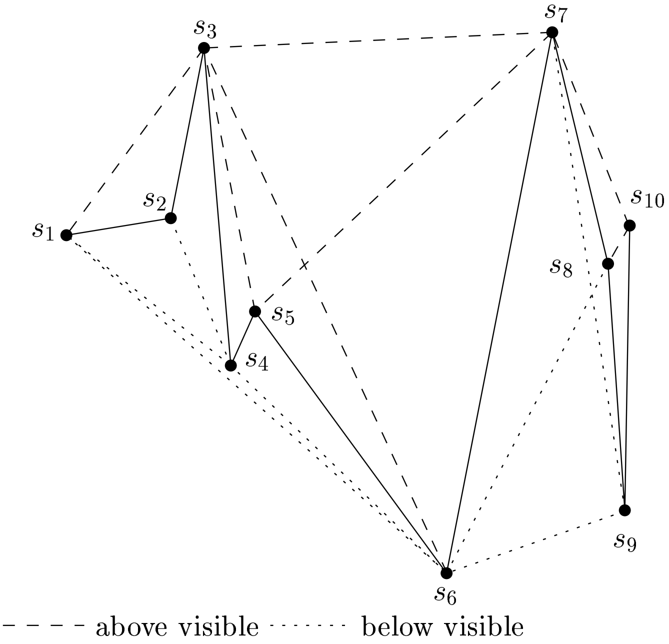


Figure 2.10: Vertex visibilities for the points of $\mathcal{S}2$.

In generateBottom, i and *curBottom* are set to 9, then i is decremented. Since point s_8 lies above $\ell(s_9, s_{10})$, *sum* is incremented by $T_9 = 5$. With $x = 1$, the repeat loop is terminated and s_8 is added to *topChain*. This step is illustrated in Fig. 2.11.

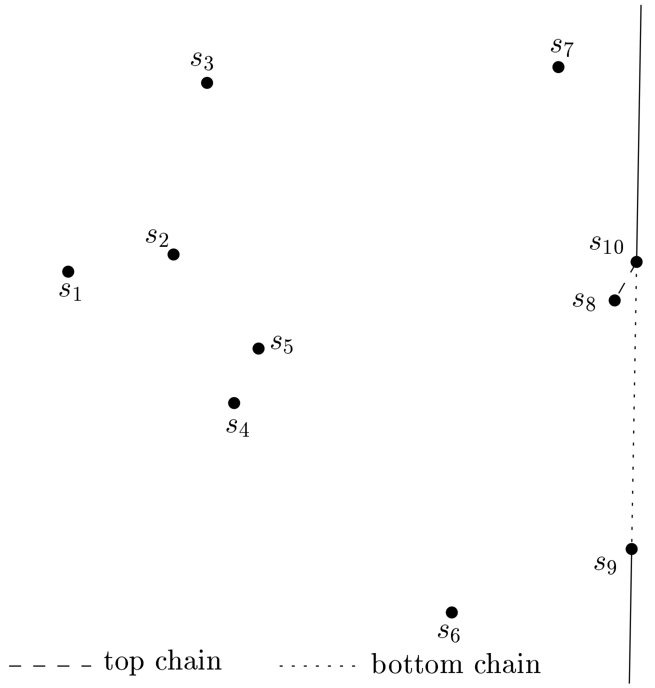


Figure 2.11: Point s_8 is added to *topChain* (Phase 1).

Finally, generateBottom calls generateTop with $k = 9$ and $x = 1$. Point s_7 lies below line $\ell(s_1, s_9)$, and $B_8 = 0$. Therefore, *curTop* is set to 7 and *sum* remains 0. Next, s_6 is tested against line $\ell(s_7, s_9)$. As a result we have *curTop* $= 6$ and *sum* $= 5$. As a consequence, s_7 is added to *topChain* and s_6 to *bottomChain* (cf. Fig. 2.12). Then, generateBottom is called with $k = 7$ and $x = 1$.

In generateBottom, s_5 lies above line $\ell(s_6, s_7)$ and since $T_6 = 3$, the repeat loop is terminated and as a result s_5 is added to *topChain* (refer to Fig. 2.13).

The next step is the execution of generateTop with parameters $k = 6$ and $x = 1$. Since s_4 lies below line $\ell(s_5, s_6)$ and $B_5 = 2$, s_4 is added to *bottomChain*, cf. Fig. 2.14.

Next, generateTop calls generateBottom with $k = 5$ and $x = 1$. Since s_3 lies above line $\ell(s_4, s_5)$ and $T_4 = 2$, s_3 is added to *topChain* (cf. Fig. 2.15).

The last step causes the execution of generateTop with $k = 4$ and $x = 1$. Point s_2 lies below line $\ell(s_3, s_4)$ and since $B_3 = 1$, the point s_2 is added to *bottomChain* (Fig. 2.16).

Finally, generateBottom is called with $k = 3$ and $x = 1$. Point s_1 lies above line $\ell(s_2, s_3)$ and $T_2 = 1$. Thus, s_1 is added to *topChain* (cf. Fig. 2.17).

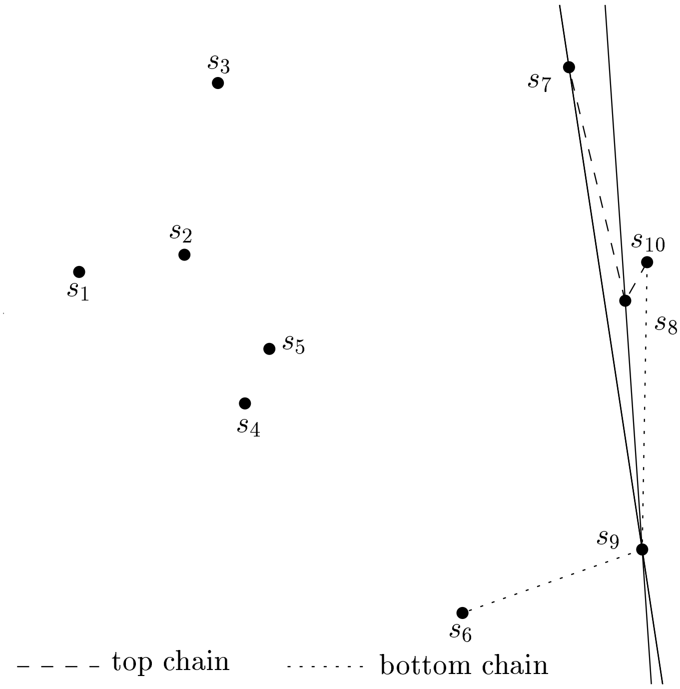


Figure 2.12: Point s_6 is added to bottomChain, point s_7 to *topChain* (Phase 2).

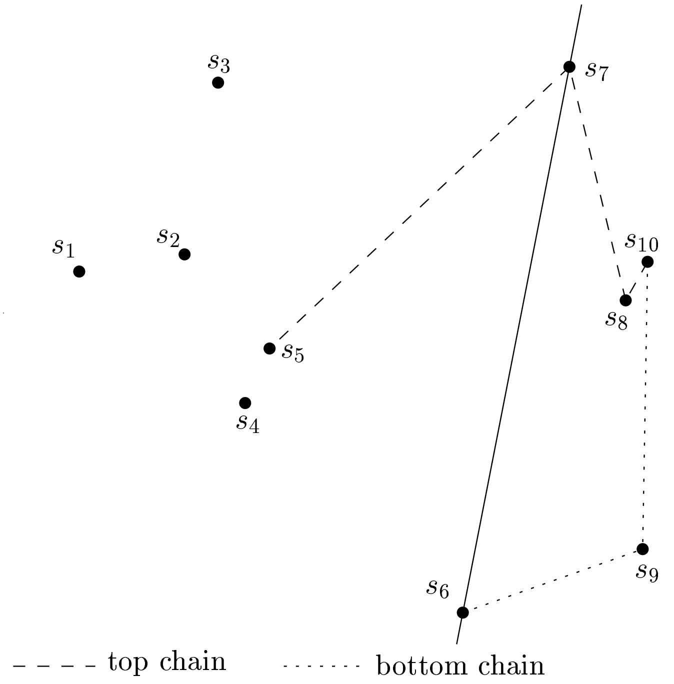


Figure 2.13: Point s_5 is added to *topChain* (Phase 3).

After the completion of the recursion, s_1 is added to *bottomChain* in Algorithm xmonotonePolygon, and the polygon is complete. The resulting polygon is depicted in Fig. 2.18.

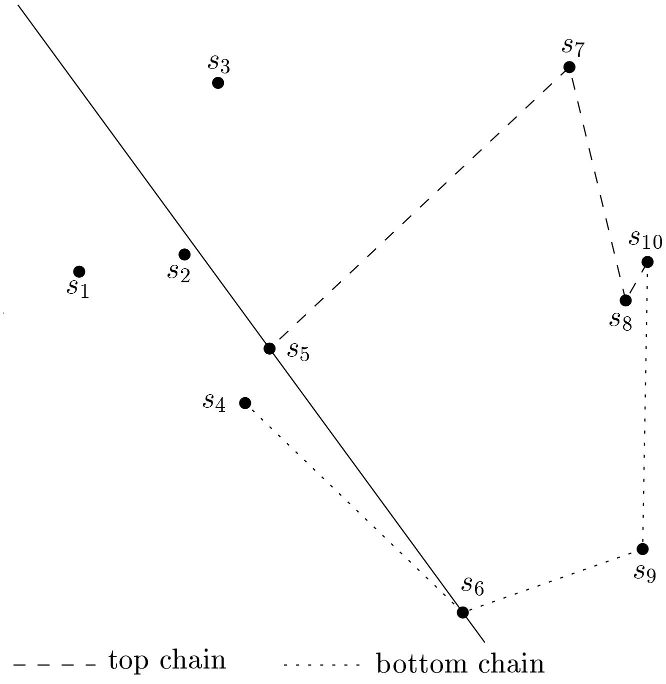


Figure 2.14: s_4 is added to *bottomChain* (Phase 4).

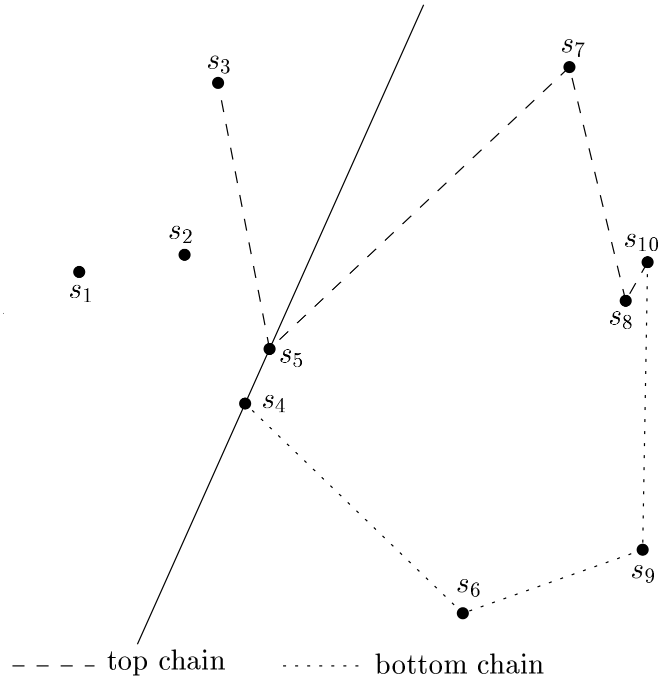


Figure 2.15: Point s_3 is added to *topChain* (Phase 5).

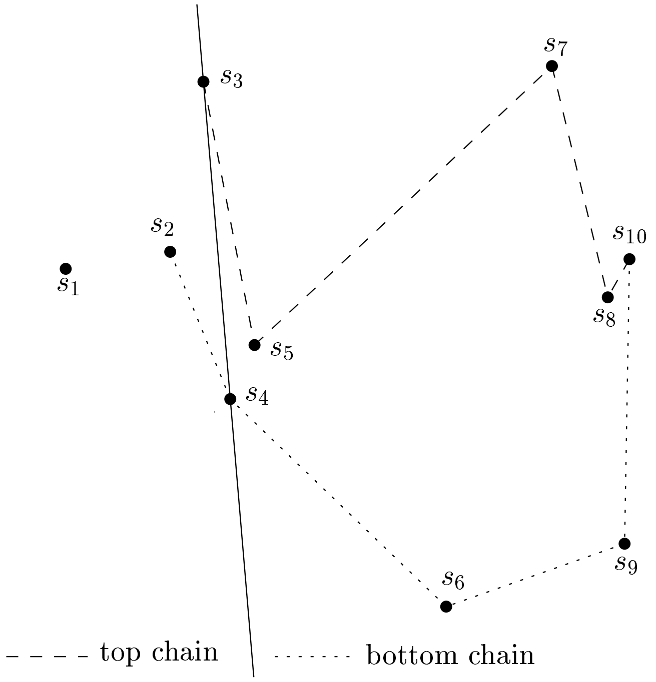


Figure 2.16: Point s_2 is added to *bottomChain* (Phase 6).

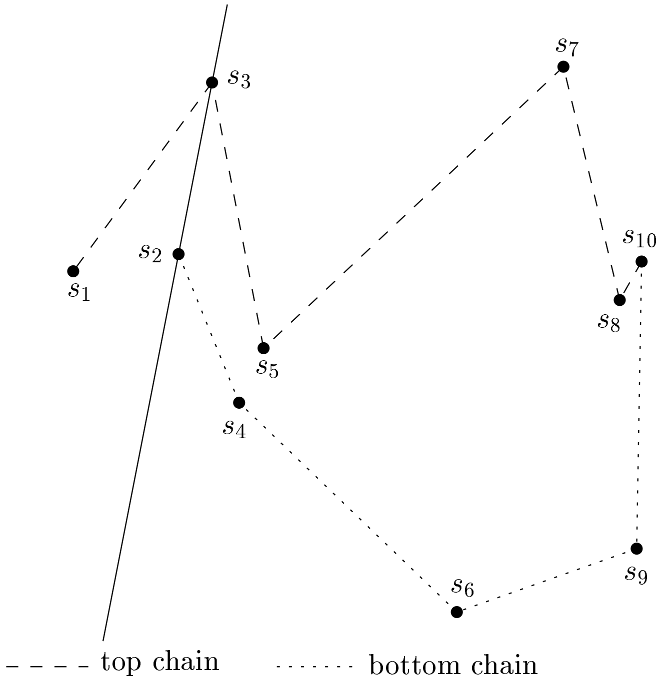


Figure 2.17: Point s_1 is added to *topChain* (Phase 7).

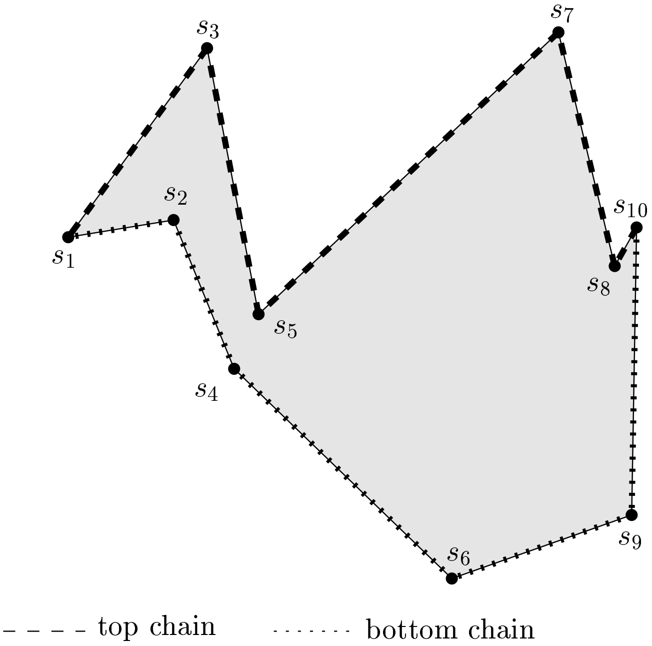


Figure 2.18: Result of x-Monotone Polygons.

2.3 Star-Shaped Polygons

2.3.1 Arrangements and Kernels

Since our method for generating star-shaped polygons is based on the concept of arrangements in the plane, we start with taking a closer look at arrangements. An *arrangement* is a collection of infinite lines "arranged" in the plane. These lines induce a partition of the plane into convex regions (called *cells*), segments (between line crossings), and vertices (where lines meet), cf. O'Rourke [O'R94]. The arrangement induced by all the lines defined by the points of set $\mathcal{S}1$ is depicted in Fig. 2.19.

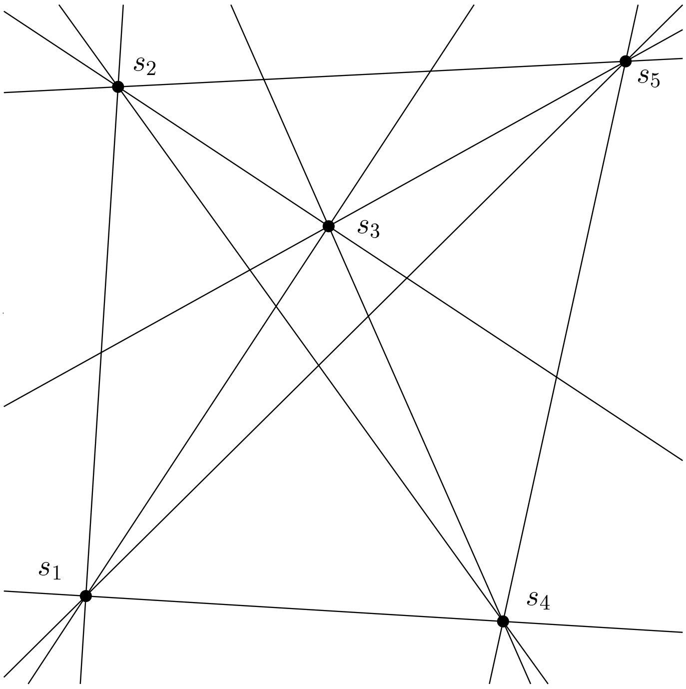


Figure 2.19: The arrangement induced by the lines of $\mathcal{S}1$.

Next, we will show that computing the arrangement of all the lines on a given point set also yields all the kernels. For this purpose, we have to state some properties of kernels. First, as stated in Preparata and Shamos [PS85], the kernel of a star-shaped polygon with n vertices can be computed as the intersection of n half-planes, where each edge of the star-shaped polygon $\mathcal{P}$ determines one half-plane. It follows immediately that the kernel forms a convex region. This is based on the fact that the intersection of convex sets such as half-planes is necessarily convex, cf. Preparata and Shamos [PS85]. Further, due to the definition of the kernel all the vertices of $\mathcal{P}$ are visible from p for each point p in the kernel. Based on this observation, we can now state the following lemma.

Lemma 2.3 Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two different star-shaped polygons. Let $\mathcal{K}_1$ be the kernel of $\mathcal{P}_1$ and let $\mathcal{K}_2$ be the kernel of $\mathcal{P}_2$. Then the intersection of $\mathcal{K}_1$ and $\mathcal{K}_2$ is either empty or consists of a single edge or a single vertex.

Proof: First, we will show that the order of the vertices in $\mathcal{P}$ is identical (up to shifts) to the order obtained when sorting the vertices in angular order around a point p in the kernel. Let v_i and v_{i+1} be two consecutive vertices in the circular order around s. Without loss of generality we assume that there exists a vertex v_k, such that v_k is the next vertex after v_i in the polygon. If the edge (v_i, v_k) lies in front of v_{i+1} then v_{i+1} is not visible. Otherwise, at least one point q on (v_i, v_k) is not visible from p, cf. Fig. 2.20. However, in both cases we have a point of $\mathcal{P}$ which is not visible from p. Therefore, p does not belong to the kernel of $\mathcal{P}$.

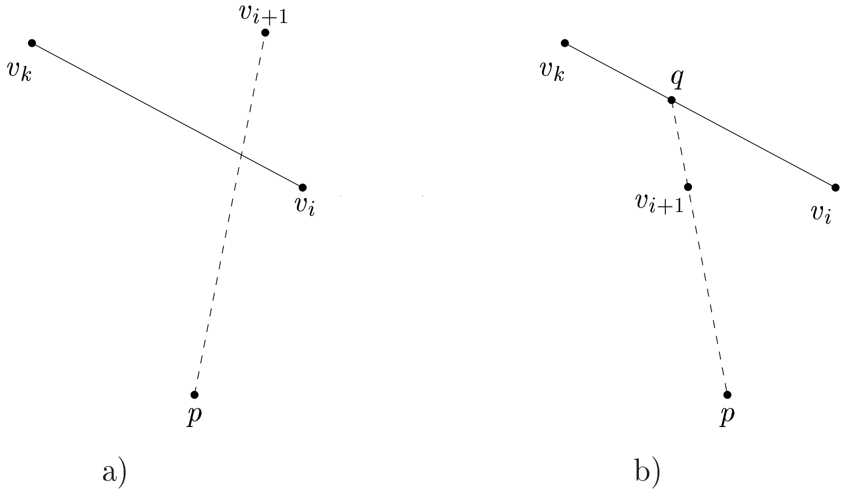


Figure 2.20: Visibility of vertices from a point p in the kernel.

Now, let us consider two points $p_1 \in \mathcal{K}_1$ and $p_2 \in \mathcal{K}_2$ that do not lie on the boundary of the kernels. Clearly, the order of the vertices of the polygon is fixed for both p_1 and p_2. Due to the assumption that $\mathcal{P}_1 \neq \mathcal{P}_2$, the order of vertices of $\mathcal{P}_1$ with respect to p_1 differs from the order of the vertices of $\mathcal{P}_2$ with respect to p_2. Thus, $p_1 \notin \mathcal{K}_2$ and $p_2 \notin \mathcal{K}_1$. Further, let us consider a point p lying on the boundary of the kernel of a star-shaped polygon $\mathcal{P}$: Clearly, the edge p lies on is defined by a line supporting an edge of $\mathcal{P}$. Thus, the order of vertices that define this edge is not fixed with respect to p and p might belong to at most two kernels of the arrangement. (If p is a vertex of the arrangement, it belongs to at most four kernels.) Based on this, $\mathcal{K}_1$ and $\mathcal{K}_2$ may share one or more edges; since $\mathcal{K}_1$ and $\mathcal{K}_2$ both are convex, they share at most one edge. □

Further, it is obvious that all kernels must lie within the convex hull of $\mathcal{S}$: For each point $p \in \mathcal{K}$ all the other points q of a polygon are visible, i.e., the segment $\overline{pq}$ lies within the polygon, which, in turn, is confined to $\mathcal{CH}(\mathcal{S})$. Also, every point within $\mathcal{CH}(\mathcal{S})$ belongs to at least one kernel[3]. Thus, the kernels of all star-shaped polygons on $\mathcal{S}$ form a valid partition of the convex hull $\mathcal{CH}(\mathcal{S})$. We will now see how this partition is related to the arrangement of all lines on $\mathcal{S}$.

---

[3] The maximum number of kernels one point belongs to is four: consider the intersection of the supporting lines of two edges of $\mathcal{P}$.

Lemma 2.4 Let $\mathcal{A}$ be the arrangement induced by all the lines defined by any two points of $\mathcal{S}$. Then each cell $\mathcal{C}$ of $\mathcal{A}$ which lies within the convex hull $\mathcal{CH}(\mathcal{S})$ is part of exactly one kernel.

Proof: Consider a cell $\mathcal{C}$ of $\mathcal{A}$. First, it is clear that every point $p \in \mathcal{C}$ belongs to at least one kernel. (Otherwise, we would get a contradiction to the fact that the kernels form a partition of the convex hull.) Thus, we have to show that there exists no cell that overlaps with more than one kernel. Let us assume that $\mathcal{C}$ is part of the adjacent[4] kernels $\mathcal{K}_1$ and $\mathcal{K}_2$, and let e be the edge shared by $\mathcal{K}_1$ and $\mathcal{K}_2$. Clearly, the edge e lies on the supporting line of one edge of the polygon defined by $\mathcal{K}_1$, and thus the supporting line is a line through two vertices of the polygon, which are also points of $\mathcal{S}$. Since $\mathcal{C}$ cannot lie on both sides of this line we get a contradiction to the fact that $\mathcal{A}$ is the arrangement of *all* the lines defined by any two points in $\mathcal{S}$. Therefore, every cell is part of exactly one kernel. $\square$

Next, we consider two consecutive vertices v_i and v_{i+1} of the polygon, and the line $\ell = \ell(v_i, v_{i+1})$. Clearly, the relative angular order with respect to a point p depends on whether p lies left or right of ℓ. Let us now consider two adjacent kernels: They are separated by a line defined by two consecutive vertices. Clearly, no other line through any two consecutive vertices is passed when moving from one kernel into an adjacent kernel. Thus, two adjacent kernels differ only in the relative order of the two vertices that define the supporting line of their common edge.

Due to the properties stated above, the arrangement of all lines also includes the kernels of all star-shaped polygons on $\mathcal{S}$. In Fig. 2.21, we depicted the arrangement induced by set $\mathcal{S}1$, and the kernels of all star-shaped polygons on $\mathcal{S}1$. We restricted the arrangement to the convex hull of set $\mathcal{S}1$. On $\mathcal{S}1$, there exist four star-shaped polygons, and their kernels consist of the following cells (we numbered them in the figure): The kernel of the polygon depicted in Fig. 2.22a consists of cells 1 to 6. The kernel of the polygon in Fig. 2.22b consists of cells 7 to 9. Next, cell 10 forms the kernel of the polygon depicted in Fig. 2.22c, and finally, the kernel of the polygon in Fig. 2.22d is formed by cells 11, 12 and 13.

We will now present two algorithms that compute all the kernels. Clearly, we will get a random star-shaped polygon on $\mathcal{S}$ if we select one of the kernels computed at random. Thus, we present the algorithms and describe only how we compute all the kernels, since the generation of a random polygon with a uniform distribution is straightforward once the kernels have been computed.

2.3.2 Star Arrange

The first algorithm for computing a star-shaped polygon on $\mathcal{S}$, Star Arrange is directly based on computing the arrangement of all lines. Having computed the arrangement, we obtain all the kernels with the following depth-first search:

---

[4] Note that the kernels which overlap with $\mathcal{C}$ always are pairwise adjacent since $\mathcal{C}$ is convex. Thus, if $\mathcal{C}$ overlaps with more than one kernel then we can always find two kernels which are adjacent and which overlap with $\mathcal{C}$.

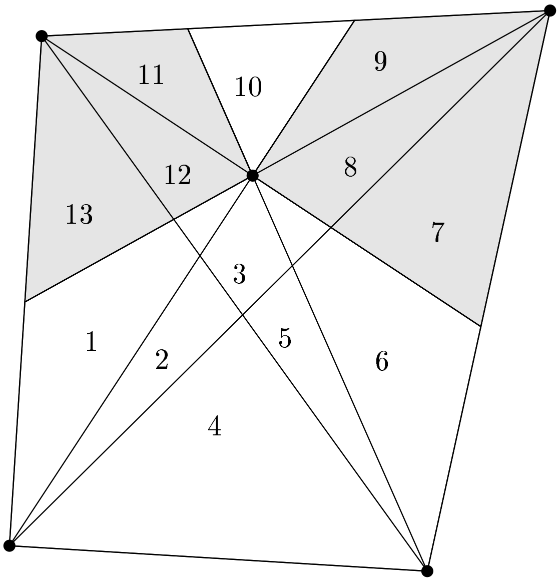


Figure 2.21: An arrangement, and the kernels implied by the arrangement.

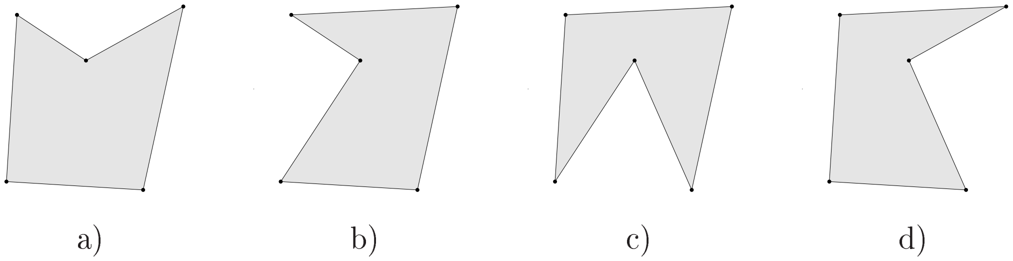


Figure 2.22: The star-shaped polygons on $\mathcal{S}1$.

1. Choose one cell and mark it as visited.
2. Create the star-shaped polygon $\mathcal{P}$ defined by this "active" cell.
3. For every edge e on the boundary of the active cell, do the following: If the neighboring cell sharing e with the current cell is part of the same kernel, then proceed with the edges of the neighboring cell using $\mathcal{P}$. Otherwise, (i.e., if the neighboring cell belongs to another kernel) invert the order of the vertices that lie on the supporting line of e, and proceed with this other cell (and the modified polygon).

Note that the neighboring cell belongs to the same kernel if and only if the two vertices defining e do not appear in consecutive order in $\mathcal{P}$.

2.3.3 Star Universe

Since tests (cf. Chapter 4) showed that algorithm Star Arrange is not suited for generating star-shaped polygons in practice[5], we developed the following method whose space-consumption is output-sensitive. This method is dubbed Star Universe. Instead of computing the arrangement of all lines, we scan along each line and keep track of the current polygon to the left and to the right of the line. (Note that the "left" and "right" polygons might be identical.) When encountering the intersection with another line, we check whether the line causing this intersection is defined by two consecutive vertices of the current polygon(s). If it is defined by two consecutive vertices v_i and v_{i+1} then we revert the relative order of these two vertices and proceed with the next intersection point.

Due to the fact that all kernels lie within $\mathcal{CH}(\mathcal{S})$, we consider only that portion of each line which lies completely within the convex hull. For computing the first "left" and "right" polygons, we take the midpoint of the first[6] intersection point of the line with the convex hull and the next intersection point, and sort the points of $\mathcal{S}$ around this point.

Note that, in contrary to Star Arrange, polygons may be encountered more than once during the execution of the algorithm. Thus, we have to keep track of all the polygons generated so far. Clearly, this can be done in more than one way (depending on the desired trade-off between time and space complexity). We opted for computing a unique point in the kernel and storing only this point of the kernel.

2.3.4 Complexity of the Algorithms

For algorithm Star Arrange, we need to generate the arrangement of all lines defined by points of $\mathcal{S}$. Since the number of lines on $\mathcal{S}$ is bound by $\mathcal{O}(n^2)$, the size of our arrangement is bound by $\mathcal{O}(n^4)$. With an incremental algorithm, the arrangement can be computed in time proportional to its size, and thus the time required is also bound by $\mathcal{O}(n^4)$. Having computed the arrangement, we need to compute all the kernels. With the method stated above this can be done in time linear to the size of the arrangement. Thus, algorithm Star Arrange has a space and time complexity of $\mathcal{O}(n^4)$.

For algorithm Star Universe we scan every line defined by two points of $\mathcal{S}$, and for each line we have to do the following: First, we need to compute all the intersection points with all the other lines and have to sort them, which requires $\mathcal{O}(n^2 \log n)$ time. Further, we have to compute the first "left" and "right" polygons. For storing and comparing the polygons, we compute a unique point in the kernel, which can be done in linear time[7] for each of the $\mathcal{O}(n^2)$ polygons, cf. Preparata and Shamos [PS85]. When storing those points in an AVL-tree, two polygons can be checked

---

[5] It consumes far too much main memory.

[6] We scan along the line in positive x direction.

[7] The kernel can be computed in linear time. Once the kernel has been determined, it is easy to compute a unique point within the kernel.

for identity in $\mathcal{O}(logn)$ time. Thus, we need $\mathcal{O}(n^3)$ time for each of the $\mathcal{O}(n^2)$ lines and therefore we have a total time complexity of $\mathcal{O}(n^5)$. However, the space requirement is only $\mathcal{O}(n^2+k)$, where k denotes the number of star-shaped polygons on $\mathcal{S}$, as opposed to $\mathcal{O}(n^4)$ for Star Arrange.

2.3.5 Quick Star

Clearly, both Star Arrange and Star Universe do not offer a good time or a good space complexity, and thus the need for a fast heuristic for the random generation of star-shaped polygons arises. For this purpose, we present the heuristic Fast Star.

As stated above, every point p which lies within $\mathcal{CH}(\mathcal{S})$ belongs also to a kernel and thus defines a star-shaped polygon. (If p lies on an edge or coincides with a vertex of the arrangement, up to four[8] star-shaped polygons are defined.)

This allows the following straightforward approach for the generation of a star-shaped polygon: Pick a random point within the convex hull $\mathcal{CH}(\mathcal{S})$ and sort the points of $\mathcal{S}$ around this random point. The generation of the random point can be done as follows. Triangulate the convex hull of $\mathcal{S}$, and choose a triangle at random with a probability proportional to the ratio between the area of the triangle and the area of the convex hull. This triangle can be directly mapped to the unit square. Thus, generating a random point in the unit square yields a random point in the triangle which is also a random point within the convex hull.

Complexity of Quick Star

The computation of the convex hull requires $\mathcal{O}(n \log n)$ time, and the triangulation can be done in linear time. Further, we need to sort the points in $\mathcal{S}$ around the randomly selected point p, which also requires $\mathcal{O}(n \log n)$ time. Thus, the total time complexity for generating one star-shaped polygon is $\mathcal{O}(n \log n)$. Clearly, Quick Star needs only a linear amount of space.

2.4 Steady Growth

Steady Growth is a heuristic that incrementally constructs a polygon: At phase k, we add a suitable point s_k to $\mathcal{P}_{k-1}$ by replacing some edge (v_i, v_{i+1}) with edges (v_i, s_k) and (s_k, v_{i+1}). Clearly, both v_i and v_{i+1} must be visible from s_k with respect to $\mathcal{P}_{k-1}$, otherwise the resulting polygon $\mathcal{P}_k$ would not be simple any more.

The addition of s_k to $\mathcal{P}_{k-1}$ has to be done in a way to ensure that all the remaining $n-k$ points can be added in phases $k+1$ to n. As can be seen in Fig. 2.23, for a point p lying within a polygon $\mathcal{Q}$, this point does not necessarily see any edge of $\mathcal{Q}$ completely. (I.e., for each edge p sees at most one of the two vertices defining the edge.) Unfortunately, the same observation holds for a point p lying outside of $\mathcal{Q}$, cf. Fig. 2.24.

---

[8] Recall that we assume $\mathcal{S}$ to be in general position.

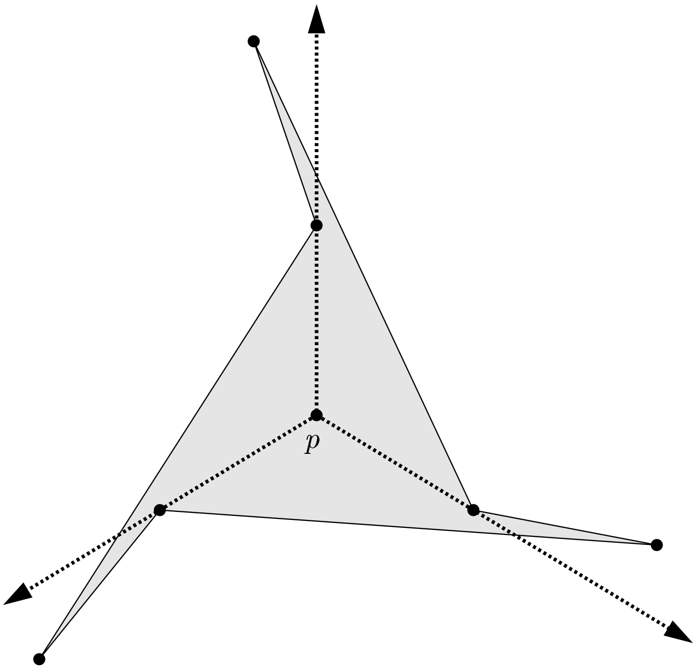


Figure 2.23: The point p does not see any edge of $\mathcal{Q}$ completely.

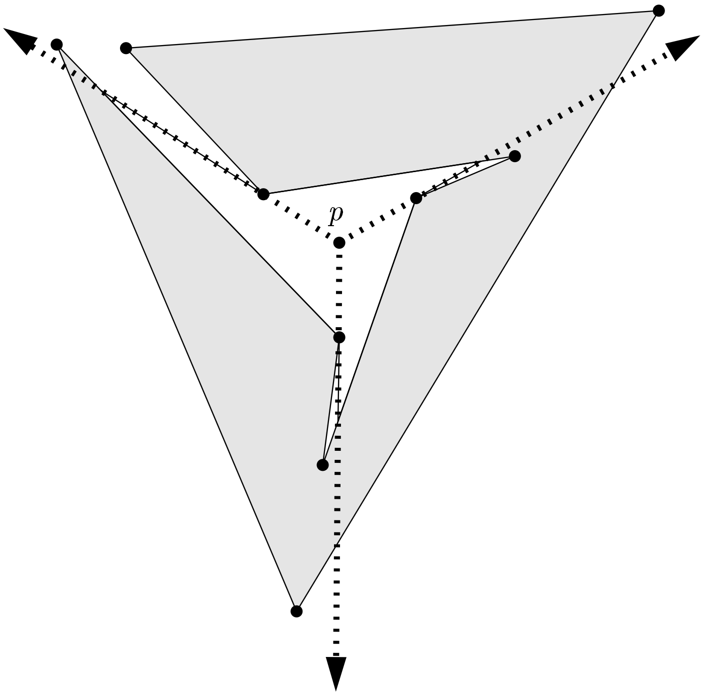


Figure 2.24: The point p does not see any edge of $\mathcal{Q}$ completely.

However, as we will show below, if p lies outside $\mathcal{CH}(\mathcal{Q})$ then there exists at least one edge that is completely visible from p. Let $\mathcal{S}_k := \mathcal{S} \setminus \{s_1, \ldots, s_{k-1}\}$, i.e., $\mathcal{S}_k$ contains all points except the vertices of the polygon $\mathcal{P}_{k-1}$. Thus, if we can guarantee that no point of $\mathcal{S}_k \setminus \{s_k\}$ lies within $\mathcal{CH}(\mathcal{P}_k)$, we will be able to successfully complete phases $k+1$ through n. For this purpose, we must incrementally construct the convex hull of $\mathcal{P}_k$, i.e., we must obtain $\mathcal{CH}(\mathcal{P}_{k-1} \cup s_k)$ from $\mathcal{CH}(\mathcal{P}_{k-1})$. All we have to do is to find the two lines of tangency from s_k to $\mathcal{CH}(\mathcal{P}_{k-1})$, cf. O'Rourke [O'R94]. Since we assumed general position, each of these lines intersects $\mathcal{CH}(\mathcal{P}_{k-1})$ at a vertex. We will call these two vertices the *left* supporting vertex v_l and the *right* supporting vertex v_r according to their appearance when looking from s_k in the direction of $\mathcal{CH}(\mathcal{P}_{k-1})$, cf. Fig. 2.25. Further, when considering a polygonal subchain $\mathcal{C}$ of the chain from the left to the right supporting vertex, we will denote its endpoints by v_l and v_r, too.

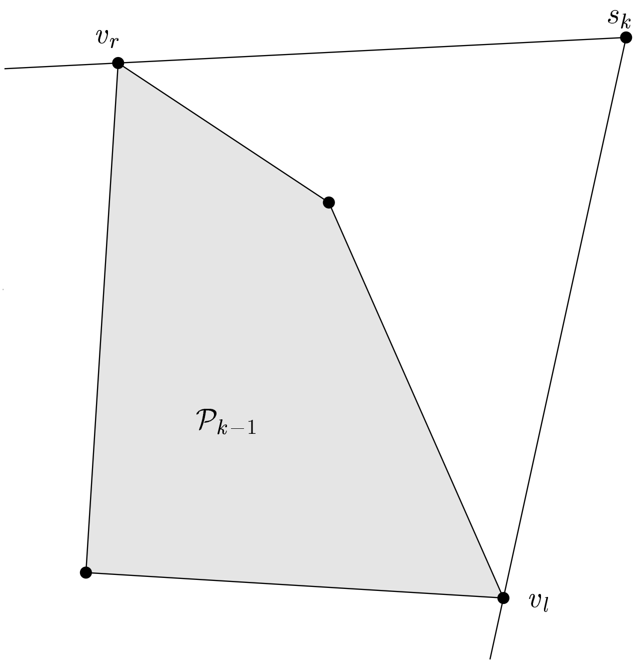


Figure 2.25: The supporting vertices of s_k.

First, we will show that we can always find a point s_k such that no point of $\mathcal{S}_k \setminus \{s_k\}$ lies within $\mathcal{CH}(\mathcal{P}_k)$:

Lemma 2.5 Let $\mathcal{P}_{k-1}$ be the polygon obtained after executing phase $k-1$ of the algorithm. There always exists a suitable point $s_k \in \mathcal{S}_k$, such that no other point $s \in \mathcal{S}_k \setminus \{s_k\}$ lies within $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$.

Proof: By definition, $\mathcal{P}_{k-1}$ contains $k-1$ points. Thus, we have $n-k+1$ points left to select s_k. Thus, for any choice of s_k, at most $n-k$ points of $\mathcal{S}$, i.e., the points of $\mathcal{S}_k \setminus \{s_k\}$, may lie within $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$.

Let us now consider a point s such that j points lie within $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s\})$. Further, let $s' \in \mathcal{S}_k \setminus \{s\}$ be one of these j points, i.e., $s' \in \mathcal{CH}(\mathcal{P}_{k-1} \cup \{s\})$.

$\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s\})$ is a convex set that contains both s' and $\mathcal{P}_{k-1}$, therefore the following relation holds: $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s'\}) \subseteq \mathcal{CH}(\mathcal{P}_{k-1} \cup \{s\})$. Thus, all points that lie outside $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s\})$ must also lie outside $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s'\})$. Consider now the supporting vertices v'_l and v'_r of s'. As depicted in Fig. 2.26, s lies right of both lines $\ell(v'_l, s')$ and $\ell(s', v'_r)$. Therefore, s does not lie within the convex hull $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s'\})$, and, as a direct consequence, at most $j-1$ points of $\mathcal{S}_k$ lie within $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s'\})$. We can now apply the same argument as before. Thus, in at most $n-k+1$ steps, we will find a suitable point s_k. □

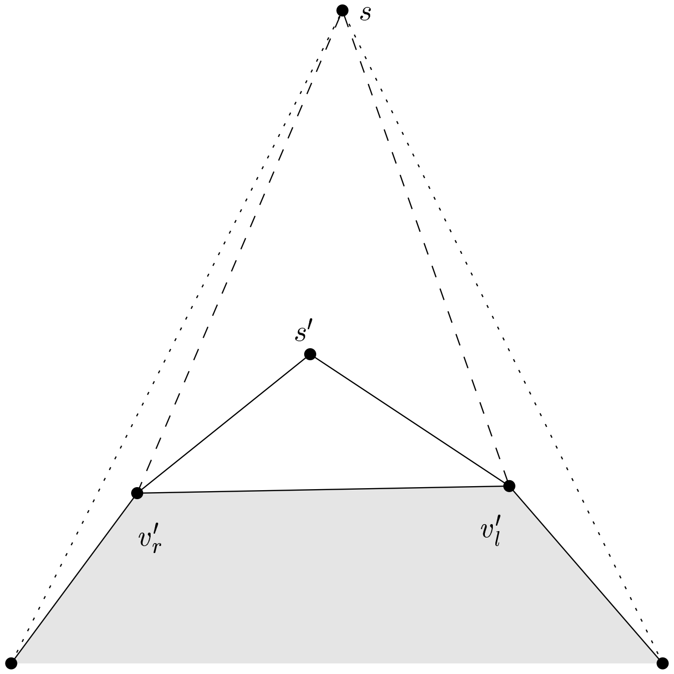


Figure 2.26: The supporting vertices of s'.

Next, after we have found s_k, we will show that it can be added to polygon $\mathcal{P}_{k-1}$, i.e., that at least one edge of $\mathcal{P}_{k-1}$ is completely visible from s_k.

Lemma 2.6 Let $\mathcal{Q}$ be a simple polygon, and p a point which lies outside of $\mathcal{CH}(\mathcal{Q})$. Further, let v_i and v_{i+1} be two consecutive vertices of $\mathcal{Q}$ which both are visible from p and which lie on the chain from the left supporting vertex v_l to the right supporting vertex v_r. Then the edge (v_i, v_{i+1}) must be completely visible from p.

Proof: We consider the triangle $\triangle(p, v_{i+1}, v_i)$. Obviously, there exists at least one edge of $\mathcal{Q}$ that lies outside of $\triangle(p, v_{i+1}, v_i)$: If the complete polygon $\mathcal{Q}$ were to lie within $\triangle(p, v_{i+1}, v_i)$ then $\mathcal{Q}$ is either degenerate[9] or v_i would be its right supporting vertex and v_{i+1} its left supporting vertex (cf. Fig. 2.27). This is a contradiction to the condition that v_i and v_{i+1} are two consecutive vertices on the chain from the left supporting vertex to the right supporting vertex.

If (v_i, v_{i+1}) were not visible from p then at least one edge e of $\mathcal{Q}$ which is in front of (v_i, v_{i+1}) necessarily exists. Especially, if e obstructs the visibility of (v_i, v_{i+1})

---

[9] I.e., it consists only of edge (v_{i+1}, v_i).

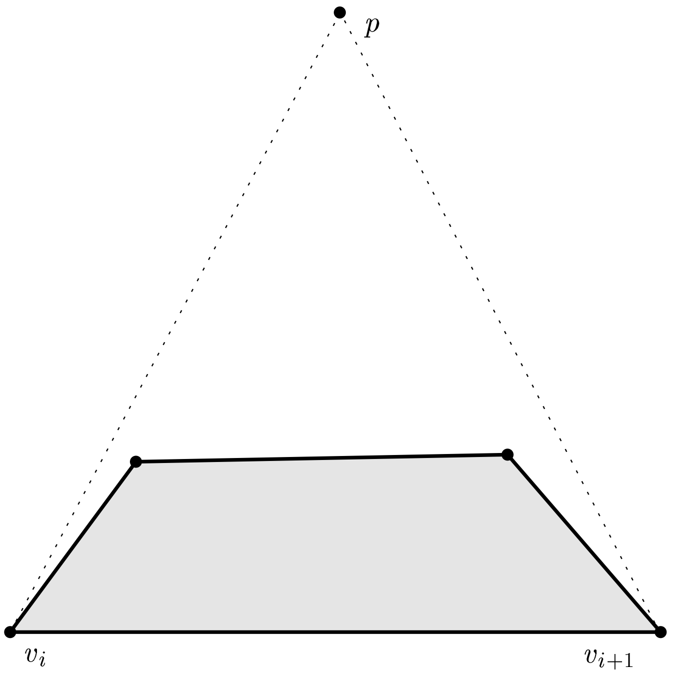


Figure 2.27: $\mathcal{Q}$ lies completely within $\triangle(p, v_{i+1}, v_i)$.

with respect to p then it must lie at least partially within $\triangle(p, v_{i+1}, v_i)$. Since e is an edge of $\mathcal{Q}$ and there exist no isolated edges in a polygon, $\mathcal{Q}$ must intersect $\partial\triangle(p, v_{i+1}, v_i)$. ($\mathcal{Q}$ does not lie completely within $\triangle(p, v_{i+1}, v_i)$, as stated above.) Based on the fact that $\mathcal{Q}$ is simple, it may not intersect (v_i, v_{i+1}). Since v_i and v_{i+1} are both visible from p, the polygon must not intersect $\overline{v_i p}$ or $\overline{v_{i+1} p}$ either. Thus, it is not possible that $\mathcal{P}$ intersects $\partial\triangle(p, v_{i+1}, v_i)$. From this it follows immediately that (v_i, v_{i+1}) must be completely visible from p. $\square$

Lemma 2.7 Let $\mathcal{Q}$ be a simple polygon, and p a point which lies outside of $\mathcal{CH}(\mathcal{Q})$. Then there exists at least one edge e on the polygonal chain from v_l to v_r that is completely visible from p.

Proof: For the proof, we consider a subchain $\mathcal{C}$ of the polygonal chain from v_l to v_r, whose endpoints are visible. We will show the property claimed in the lemma by induction on the length of $\mathcal{C}$: From Lemma 2.6, it follows immediately that the one and only edge of a chain $\mathcal{C}$ of length 1 is visible.

Assume that at least one edge is visible for any chain $\mathcal{C}$ with length at most $j-1$. Now we consider a chain $\mathcal{C}$ with length j. If the rightmost edge e of $\mathcal{C}$ is visible, we are done. If not, there must be at least one edge within $\mathcal{C}$ that is in front of e. We consider the rightmost edge e' (in angular order around p) of the edges obstructing the visibility of e, cf. Fig. 2.28. Let $\mathcal{C}'$ be the chain starting with v_l and ending with e'. Clearly, the "right" vertex v'_r of e' must be visible from p. Thus, we have a chain $\mathcal{C}'$ of length at most $j-1$ whose start point and whose end point are visible. By our assumption, at least one of the edges of $\mathcal{C}'$ must be visible from p. □

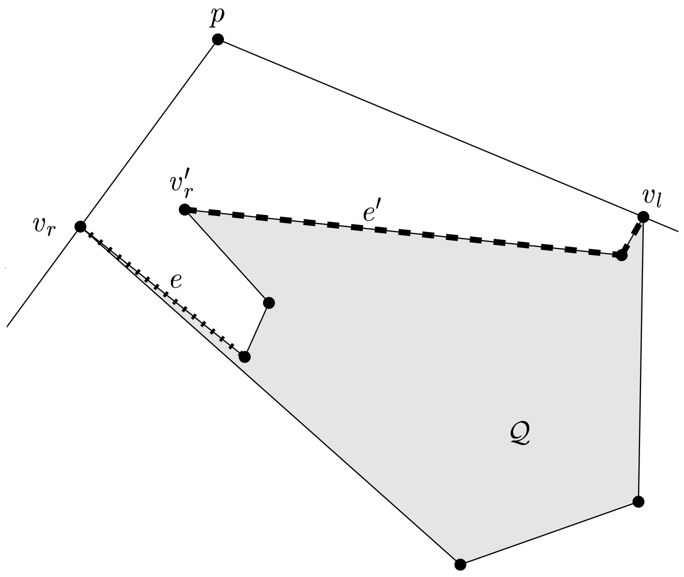


Figure 2.28: Finding a subchain with visible start and end point.

Now we can state the following algorithm which incrementally constructs a simple polygon on $\mathcal{S}$. Note that the polygon and the convex hull are degenerate when we init them with only two points. However, this has no negative impact on our algorithm.

Algorithm SteadyGrowth($\mathcal{S}$)

```
Select $s_1 \in \mathcal{S}$ at random.
Select $s_2 \in \mathcal{S} \setminus \{s_1\}$ at random.
Let $\mathcal{P}_2 := (s_1, s_2)$.
Let $\mathcal{S}_3 := \mathcal{S} \setminus \{s_1, s_2\}$.
Init the convex hull with $s_1$ and $s_2$.
for $k = 3$ to $n$ do
    Pick a point $s_k$ such that all points $s \in \mathcal{S}_k \setminus \{s_k\}$ lie outside $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$.
    Insert $s_k$ into the convex hull.
    Find all edges that are visible from $s_k$.
    Pick one visible edge $e = (v_i, v_{i+1})$ at random.
    $\mathcal{P}_k$ is obtained from $\mathcal{P}_{k-1}$ by replacing the edge $e$ with the edges $(v_i, s_k)$
    and $(s_k, v_{i+1})$.
    Let $\mathcal{S}_{k+1} := \mathcal{S}_k \setminus \{s_k\}$.
```

Theorem 2.2 Algorithm SteadyGrowth incrementally generates a simple polygon on $\mathcal{S}$.

Proof: From Lemma 2.5 it follows that a suitable point s_k can be found such that no point of $\mathcal{S}_k \setminus \{s_k\}$ lies within $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$, and from Lemma 2.7 it follows that s_k can be added to $\mathcal{P}_{k-1}$ such that the edges incident upon s_k do not intersect any other edges of the polygon. Thus, the resulting polygon is simple. □

Unfortunately, Steady Growth is not able to generate all polygons on a given set of vertices. In Fig. 2.29 we depict a polygon Steady Growth is not able to generate. We will consider all polygons with a fixed number of vertices and show that Steady Growth cannot construct a polygon with more than seven vertices on this set of points. We assume that the polygon is produced in *CCW* orientation, and therefore the vertex indices must be in increasing order. Due to the fact that the polygon in Fig. 2.29 is symmetric, we will only consider the polygons constructed when starting on the left. First, all valid triangles with three vertices are: (v_1, v_2, v_5), (v_1, v_5, v_6), (v_1, v_6, v_{11}), (v_2, v_3, v_4), (v_2, v_3, v_7), (v_2, v_3, v_8), (v_2, v_3, v_9), (v_2, v_3, v_{10}) and (v_2, v_4, v_5). Based upon these triangles, we can construct the following quadrangles: (v_1, v_2, v_4, v_5), (v_1, v_2, v_5, v_6), (v_1, v_3, v_5, v_6), (v_2, v_3, v_4, v_5), (v_2, v_3, v_7, v_8) and (v_2, v_3, v_9, v_{10}). Based upon these quadrangles, the polygon $(v_1, v_2, v_3, v_4, v_5)$ with five vertices can be constructed by Steady Growth. This polygon can be extended to polygon $(v_1, v_2, v_3, v_4, v_5, v_6)$. Finally, there exists only one possible extension: The last polygon Steady Growth will yield is $(v_1, v_2, v_3, v_4, v_5, v_6, v_9)$. For this polygon, no additional vertex can be added without violating the constraints of the algorithm.

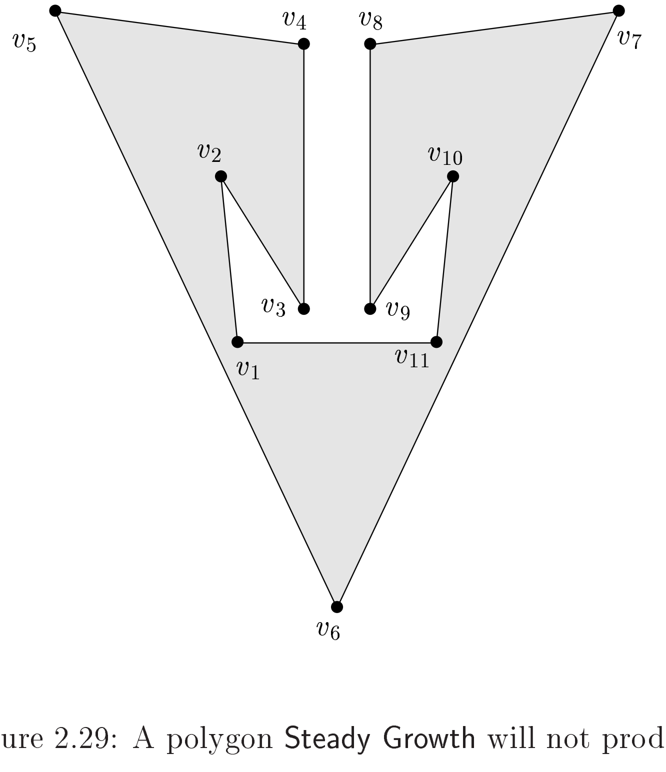


Figure 2.29: A polygon Steady Growth will not produce.

2.4.1 Complexity of the Algorithm

By using Steady Growth, one can compute a simple polygon in at most $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space: Clearly, the selection of s_1 and s_2 and the initialization of the convex hull can be done in constant time and with constant space. For each step, we have to do the following:

1. We have to find a suitable point s_k. This can be done in linear time: Pick one candidate s_k at random. Now check all the other points s of $\mathcal{S}_k$: If s lies outside $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$, proceed with the next point. Else, if s lies within $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$, discard s_k and take point s as new candidate. As shown in Lemma 2.5, no point already tested and lying outside $\mathcal{CH}(\mathcal{P}_{k-1} \cup \{s_k\})$ has to be considered again and we proceed with the remaining points. Thus, every point is considered only once and we can select a suitable point in linear time.

   Further we have to find the supporting vertices for each point. This can also be done in $\mathcal{O}(n)$ as follows: For the first candidate, find the supporting vertices by using binary search, cf. Preparata and Shamos [PS85]. For the following candidates, check the supporting vertices of the previous candidate. If they are also the supporting vertices of the new candidate, we are done. Otherwise, for the left supporting vertex, proceed along the chain $\mathcal{C}$ from v_l to v_r (in CCW direction), and for the right supporting vertex, proceed in CW direction along $\mathcal{C}$. Thus, all the updates of the supporting vertices can be done in linear time. Clearly, the selection of s_k only requires a constant amount of space.

2. Having selected s_k, we must update the convex hull. This can be easily done in constant time and with constant space if we store the convex hull as a linked list in an array. (By using an array-based linked list we do not need to de-allocate the vertices of the hull that have become obsolete.)

3. We have to find all edges that are visible from s_k. For this, we use the linear algorithm by Joe and Simpson [JS87]: Their algorithm computes the visibility polygon of a given polygon in linear time. It makes use of a stack, but due to the fact that during each step at most a constant number of items are added to the stack, the space requirement is bound by $\mathcal{O}(n)$. All we have to do is to check whether an edge of the visibility polygon is also an edge of the original polygon (this can be done in linear time when scanning along both polygons), and further we only consider edges lying on $\mathcal{C}$.

4. Among all the visible edges, we have to select one at random. Also, we have to add s_k to the polygon. Clearly, this can be done in constant time and space.

As we have outlined above, each phase can be completed in linear time and with linear space. Note that only the polygon under construction and its convex hull have to be saved from phase to phase. Thus, the total space requirement is bound by $\mathcal{O}(n)$. Further, the consumption of a linear amount of time for each phase leads to a total of $\mathcal{O}(n^2)$ time.

2.4.2 Trace of the Algorithm

For a better understanding of how Steady Growth works, we trace the algorithm on set $\mathcal{S}2$. This trace was actually obtained by running our implementation of Steady Growth. For each phase, we depict the point selected, the polygon obtained so far and its convex hull. Further, both supporting lines and all edges that are completely visible and that are thus candidates for adding s_k are shown. The selection of s_k has not been depicted, as this would have made the figures hard to digest. Note that in misuse of terminology, we will number the points in the order in which they are encountered and not according to their lexicographic ordering.

We did not depict the selection of the first two points, s_1 and s_2, as this is straightforward. In Fig. 2.30, the selection of s_3 is shown, as are s_1 and s_2. Since the "polygon" contains only one edge (to be exact, the two oriented edges corresponding to this edge), edge (s_2, s_1) is replaced by edges (s_2, s_3) and (s_3, s_1).

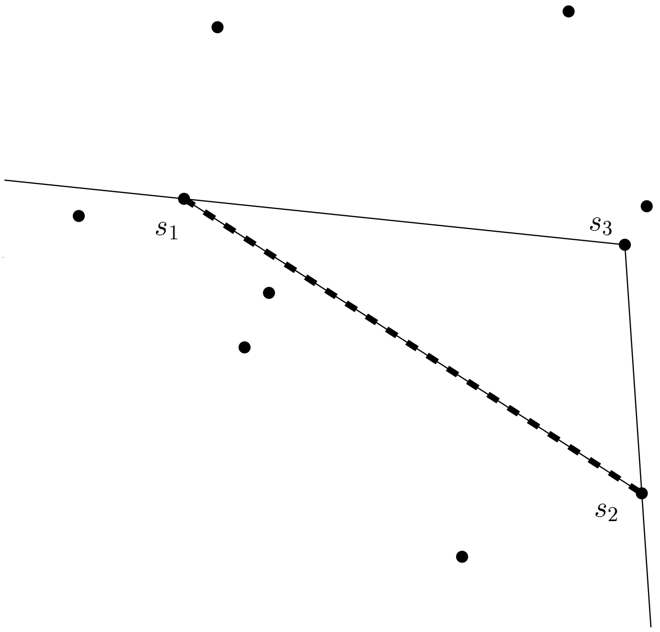


Figure 2.30: Phase 3 of Steady Growth

In phase 4, as depicted in Fig. 2.31, the supporting vertices are s_3 (left) and s_1 (right). Since edge (s_3, s_1) is the only visible edge, it is used for adding s_4.

The next point to be added, s_5, again sees only one edge, (s_1, s_2), and so this edge is replaced by edges (s_1, s_5) and (s_5, s_2). Phase 5 is depicted in Fig. 2.32.

In the next phase, as shown in Fig. 2.33, two edges of the chain from $v_l = s_2$ to $v_r = s_4$ are visible from s_6. Among these two edges, edge (s_3, s_4) is chosen at random and replaced by the edges (s_3, s_6) and (s_6, s_4).

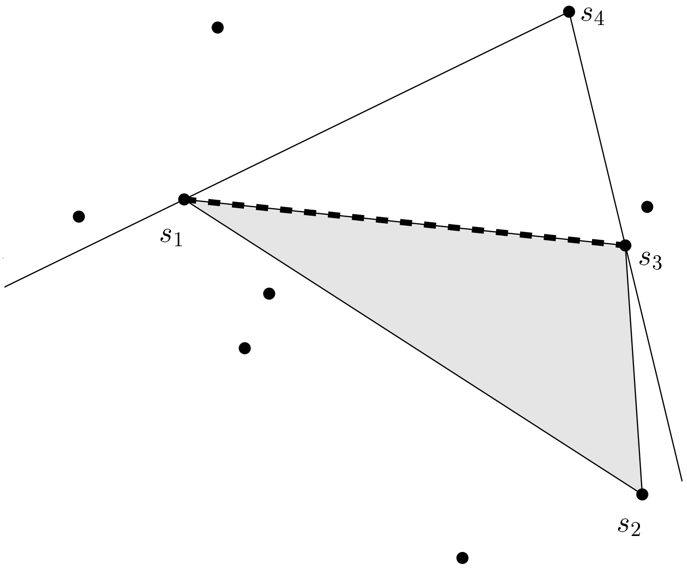


Figure 2.31: Phase 4 of Steady Growth

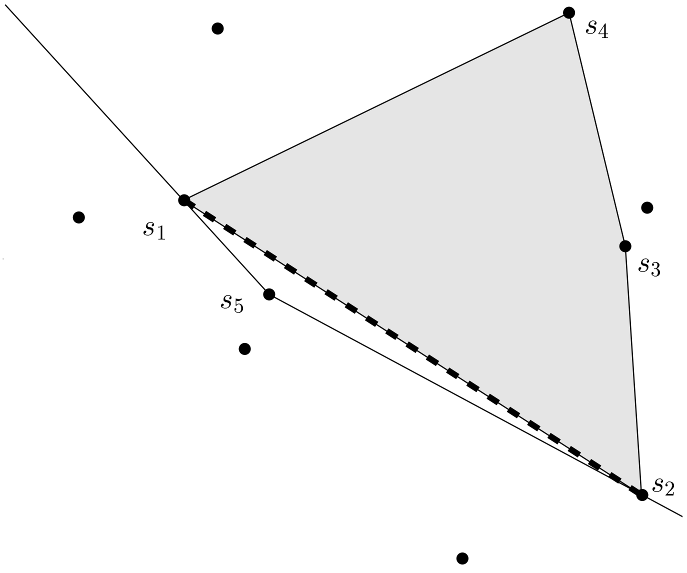


Figure 2.32: Phase 5 of Steady Growth

After the addition of s_6 to the polygon, the polygon is not identical to its convex hull any more, as it was during phases 1 to 6. The polygon, as depicted in Fig. 2.34, is now defined by $\mathcal{P}_6 = (s_1, s_5, s_2, s_3, s_6, s_4)$, whereas the convex hull $\mathcal{CH}(\mathcal{P}_6)$ is defined by $\mathcal{CH}(\mathcal{P}_6) = (s_1, s_5, s_2, s_6, s_4)$. For adding point s_7, only edge (s_4, s_1) is

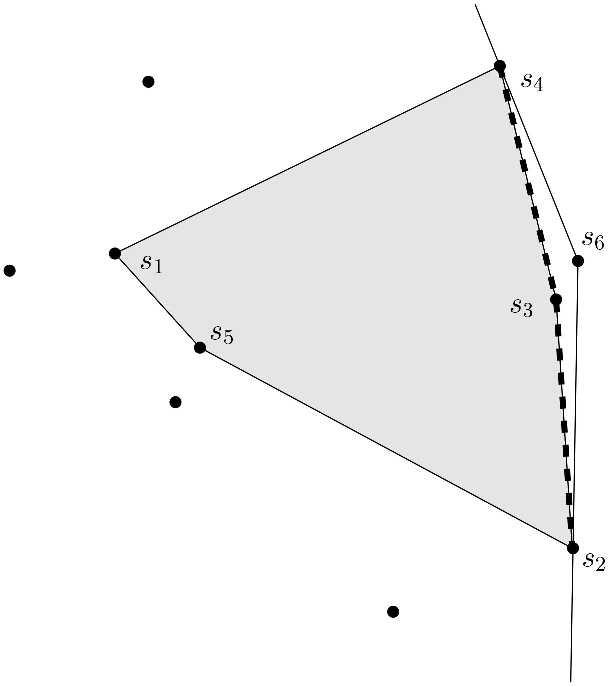


Figure 2.33: Phase 6 of Steady Growth

suitable.

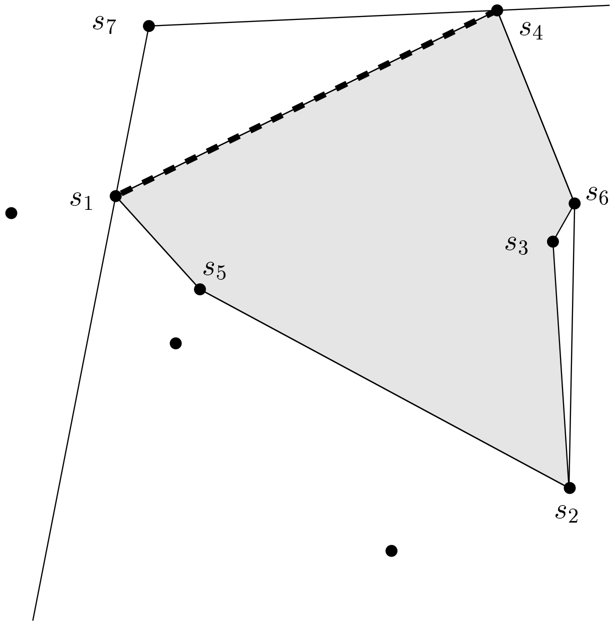


Figure 2.34: Phase 7 of Steady Growth

In Phase 8 (cf. Fig. 2.35), the left supporting vertex is s_7 and the right supporting vertex is s_2. Thus, the three edges (s_7, s_1), (s_1, s_5) and (s_5, s_2) are visible from s_8.

From these edges, (s_5, s_2) is selected at random and is used for adding s_8.

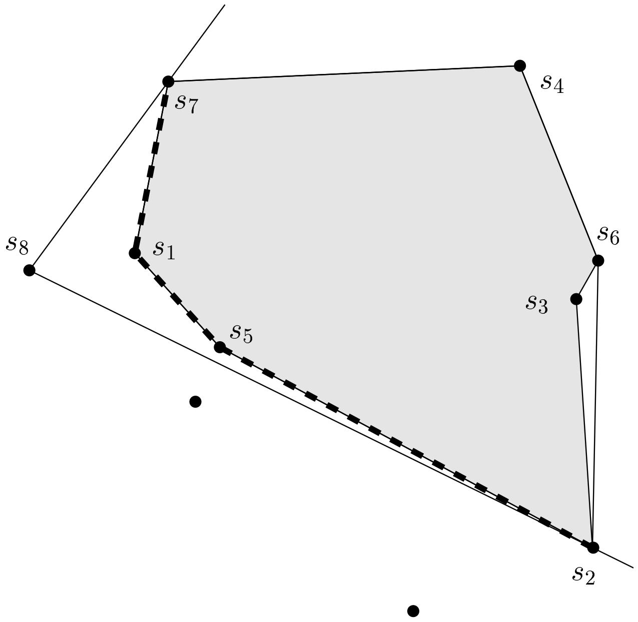


Figure 2.35: Phase 8 of Steady Growth

For adding s_9, the chain from the left supporting vertex s_8 to the right supporting vertex s_2 contains only one edge and therefore (s_8, s_2) is replaced by edges (s_8, s_9) and (s_9, s_2). This can be seen in Fig. 2.36.

In the last phase, Phase 10, there are two edges that might be used for adding s_{10}; those are (s_8, s_9) and (s_9, s_2). From these two edges that are completely visible from s_{10}, edge (s_8, s_9) is chosen at random and replaced by (s_8, s_{10}) and (s_{10}, s_9). Phase 10 is depicted in Fig. 2.37.

Finally, Fig. 2.38 shows the simple polygon generated by Steady Growth. It is defined by the vertices (in *CCW* order) $\mathcal{P} = (s_8, s_{10}, s_9, s_2, s_3, s_6, s_4, s_7, s_1, s_5)$, and its convex hull is given by $\mathcal{CH}(\mathcal{P}) = (s_8, s_{10}, s_2, s_6, s_4, s_7)$.

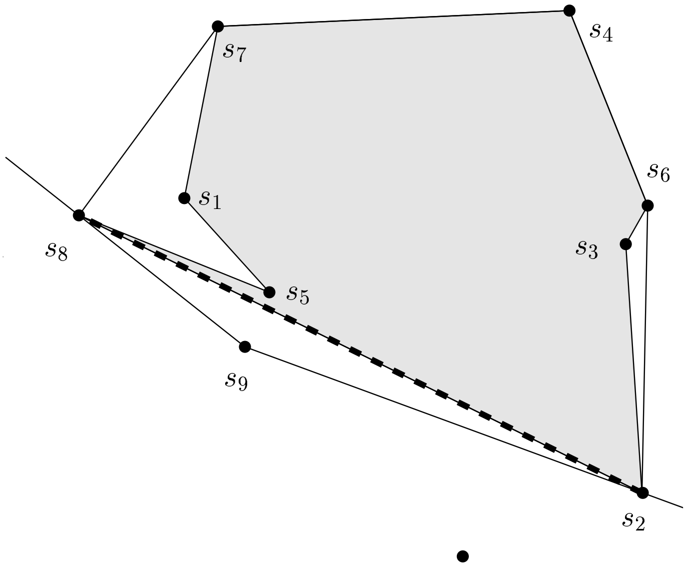


Figure 2.36: Phase 9 of Steady Growth

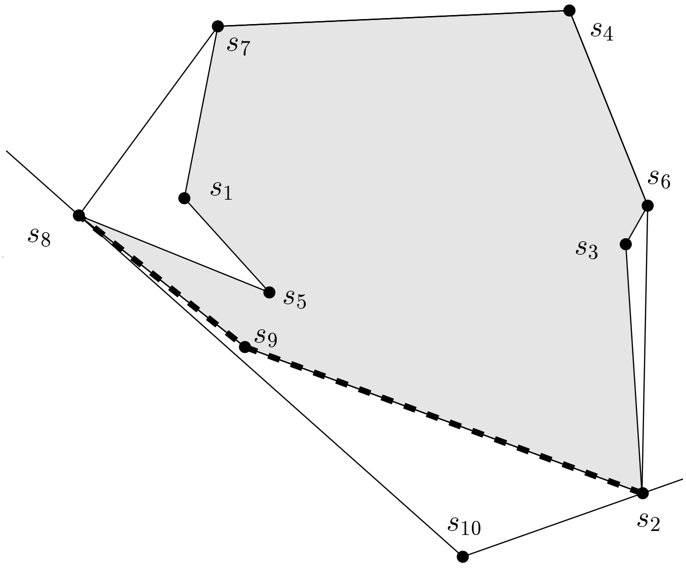


Figure 2.37: Phase 10 of Steady Growth

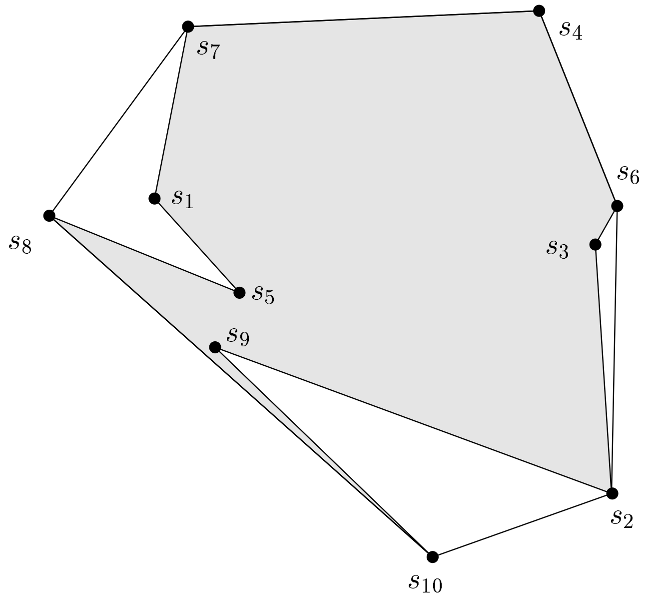


Figure 2.38: The simple polygon $\mathcal{P}$ generated by Steady Growth.

2.5 Space Partitioning

Space Partitioning is a divide-and-conquer approach for the generation of a random polygon: We divide the set $\mathcal{S}$ recursively into subsets with disjoint convex hulls. We will guarantee that the intersection of the resulting polygon $\mathcal{P}$ with the convex hull of any of the subsets consists of one single chain $\mathcal{C}$. The first point of $\mathcal{C}$ (in the same orientation as $\mathcal{P}$) is denoted by s_f, and the last point of $\mathcal{C}$ is denoted by s_l.

Let $\mathcal{S}'$ be a subset of $\mathcal{S}$ generated so far. When splitting $\mathcal{S}'$, we select the points s'_f and s'_l and then generate the polygon in a way that the intersection property stated above is satisfied. Because we want to connect different polygonal chains at their first and last points, we require that both s'_f and s'_l lie on the boundary of the convex hull of the subset. We choose a random point s' in the set $\mathcal{S}'$ to split and a random line ℓ through s' such that s'_f and s'_l lie on opposite sides of ℓ. The following lemma ensures that this method generates a valid chain in the set.

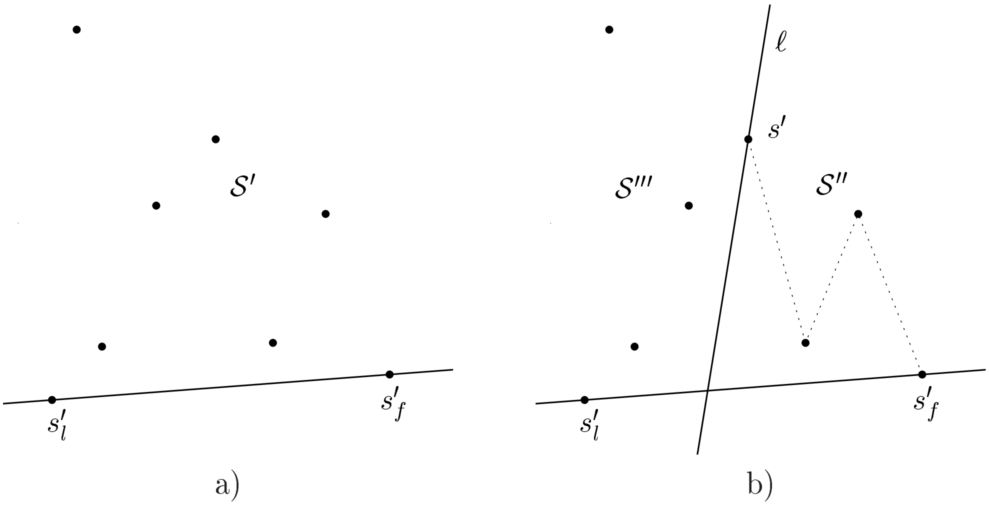


Figure 2.39: Division of set $\mathcal{S}'$ and a sample path through $\mathcal{S}''$.

Lemma 2.8 Let $\mathcal{S}'$ be a subset of $\mathcal{S}$, such that no point $s \in \mathcal{S} \setminus \mathcal{S}'$ lies within $\mathcal{CH}(\mathcal{S}')$. Further, let s'_f be its first point and s'_l its last point and assume that they lie on $\partial\mathcal{CH}(\mathcal{S}')$. Then we can recursively create a polygonal chain from s'_f to s'_l such that the chain is entirely contained in $\mathcal{CH}(\mathcal{S}')$, and such that all the points $s \in \mathcal{S}'$ are vertices of the chain:

1. We select a random point $s' \in \mathcal{S}'$, and a random line ℓ through s' separating s'_f and s'_l.
2. We split the points of $\mathcal{S}'$ according to whether they lie left of or right of ℓ.
3. We recursively generate the chain in the set of points lying to the left of ℓ and in the set of points lying to the right of ℓ.

Proof: We will show this by induction. First, we will consider a set $\mathcal{S}'$ of two points. In this case, the convex hull is the line segment from s'_f to s'_l. For the generation of the chain, there is only one solution, it consists of the edge from s'_f to s'_l. Clearly, this chain lies within the convex hull. Now let us assume that we can generate chains for all sets with $k-1$ points. We will now show that we can generate a chain for a set $\mathcal{S}'$ with k points. We choose a random point $s' \in \mathcal{S}' \setminus \{s'_f, s'_l\}$. Further, we select a line ℓ through s' such that s'_f and s'_l lie on opposite sides of ℓ and no other point lies on ℓ. This is illustrated in Fig. 2.39: Fig. 2.39a shows the set and Fig. 2.39b shows the resulting subsets. We orient ℓ in a way that s'_f lies to the right of ℓ. Let $\mathcal{S}''$ contain all the points of $\mathcal{S}'$ that lie right of ℓ, and let $\mathcal{S}'''$ contain all the points that lie left of ℓ. Note that "right of ℓ" does not include the points that lie on ℓ. Since $s_f \notin \mathcal{S}'''$ and $s_l \notin \mathcal{S}''$, both sets contain at most $k-1$ points. The first point of $\mathcal{S}''$ is s'_f and its last point is s', whereas the first point of $\mathcal{S}'''$ is s' and the last point is s'_l. Naturally, we add the first and the last points to the sets.

Now we have to show that both sets have their first and last points on the boundaries of their convex hulls and that the convex hulls contain no other points of $\mathcal{S}$. We will show this for $\mathcal{S}''$, but the argument is the same for $\mathcal{S}'''$. The convex hull $\mathcal{CH}(\mathcal{S}')$ is divided into two convex regions by ℓ. We consider the region $\mathcal{R}$ to the right of ℓ. Since s' lies on ℓ and ℓ forms an edge of $\partial\mathcal{R}$, and further, since s'_f lies on the boundary of $\mathcal{CH}(\mathcal{S}')$, both points lie on the boundary $\partial\mathcal{R}$ of $\mathcal{R}$. Thus, both points must also lie on $\partial\mathcal{CH}(\mathcal{S}'')$, because $\mathcal{CH}(\mathcal{S}'') \subseteq \mathcal{R}$.

Due to $\mathcal{CH}(\mathcal{S}'') \subseteq \mathcal{R} \subset \mathcal{CH}(\mathcal{S}')$, the convex hull $\mathcal{CH}(\mathcal{S}'')$ contains no points $s \in \mathcal{S} \setminus \mathcal{S}'$. Further, since $\mathcal{R}$ lies to the right of ℓ and the points of $\mathcal{S}'''$ lie to the left of ℓ, no point $s \in \mathcal{S}'''$ exists with $s \in \mathcal{R}$. Thus, $\mathcal{CH}(\mathcal{S}'')$ does not contain any point $s \in \mathcal{S} \setminus \mathcal{S}''$. Note that, the resulting polygonal chain on $\mathcal{S}'$ is simple due to the fact that the chains in the subsets $\mathcal{S}''$ and $\mathcal{S}'''$ lie completely within $\mathcal{CH}(\mathcal{S}'')$ and $\mathcal{CH}(\mathcal{S}''')$, respectively. □

Algorithm RecursiveDivide($\mathcal{S}', s'_f, s'_l$, *VAR* $\mathcal{C}$)

```
$n := |\mathcal{S}'|$.
if $n = 2$ then
    Add edge $(s'_f, s'_l)$ to $\mathcal{C}$.
else
    Select a point $s' \in \mathcal{S}' \setminus \{s'_f, s'_l\}$.
    Select a random line $\ell$ through $s'$ s.t. $s'_f$ lies right of $\ell$ and $s'_l$ left of $\ell$.
    $\mathcal{S}'' := \{s'\}$.
    $\mathcal{S}''' := \{s'\}$.
    for $i := 1$ to $n$ do
        if $s'_i$ lies right of $\ell$ then
            Add $s'_i$ to $\mathcal{S}'$.
        else
            Add $s'_i$ to $\mathcal{S}''$.
    RecursiveDivide($\mathcal{S}'', s'_f, s', \mathcal{C}$).
    RecursiveDivide($\mathcal{S}''', s', s'_l, \mathcal{C}$).
```

Algorithm RecursiveDivide uses the constructive method of Lemma 2.8 to recursively generate a chain in $\mathcal{S}$. Based on Lemma 2.8 we can now state the following theorem, which gives the correctness of Algorithm SpacePartitioning.

Algorithm SpacePartitioning($\mathcal{S}$)

```
$n := |\mathcal{S}|$.
Select a random point $s_f \in \mathcal{S}$.
Select a random point $s_l \in \mathcal{S} \setminus \{s_f\}$.
Let $\ell := \ell(s_f, s_l)$.
$\mathcal{S}'' := \{s_f, s_l\}$.
$\mathcal{S}''' := \{\}$[10].
for $i := 1$ to $n$ do
    if $s_i$ lies right of $\ell$ then
        Add $s_i$ to $\mathcal{S}''$.
    else
        Add $s_i$ to $\mathcal{S}'''$.
$\mathcal{C}_1 := ()$.
$\mathcal{C}_2 := ()$.
RecursiveDivide($\mathcal{S}'', s_f, s_l, \mathcal{C}_1$).
RecursiveDivide($\mathcal{S}''', s_l, s_f, \mathcal{C}_2$).
$\mathcal{P} := \mathcal{C}_1$ concatenated with $\mathcal{C}_2$.
```

Theorem 2.3 Using Space Partitioning a simple polygon $\mathcal{P}$ can be constructed on set $\mathcal{S}$.

Proof: First, we choose two random points $s_f, s_l \in \mathcal{S}$. Then, we split the points of $\mathcal{S}$ according to whether they lie left or right of $\ell(s_f, s_l)$. Let $\mathcal{S}''$ be the set of points lying to the right including s_f and s_l, and let $\mathcal{S}'''$ be the set of points lying to the left together with $\{s_f, s_l\}$. This is depicted in Fig. 2.40. Since $\mathcal{S}$ is split by $\ell(s_f, s_l)$, the points s_f and s_l must lie on the convex hulls of both $\mathcal{S}''$ and $\mathcal{S}'''$. Now we take s_f as first point of $\mathcal{S}''$ and s_l as last point of $\mathcal{S}''$, and for $\mathcal{S}'''$, we take s_l as first point and s_f as last point. On these two subsets, we can create two polygonal chains with the constructive method given in Lemma 2.8. Note that the two chains do not intersect except at s_f and s_l. Finally, we connect these two chains at s_f and s_l and thus obtain a simple polygon. □

Unfortunately, Space Partitioning will not generate all possible polygons on a given set $\mathcal{S}$ of points. In Fig. 2.41 we depict a polygon with six vertices which cannot be generated by Space Partitioning: We show that we cannot find two initial points for starting the recursive subdivision. Since the polygon is symmetric, we will only consider vertices v_1 and v_2. First, let us consider v_1. If the first two vertices were v_1 and v_2 then the polygon may not contain (v_4, v_5) since v_4 and v_5 lie on opposite sides of $\ell(v_1, v_2)$. The same observation applies to the initial points v_1 and v_6. For v_3 as second initial point the polygon must contain edge (v_1, v_3) since no

---

[10] Note that s_f and s_f will be added to $\mathcal{S}'''$ in the **else**-construct below.

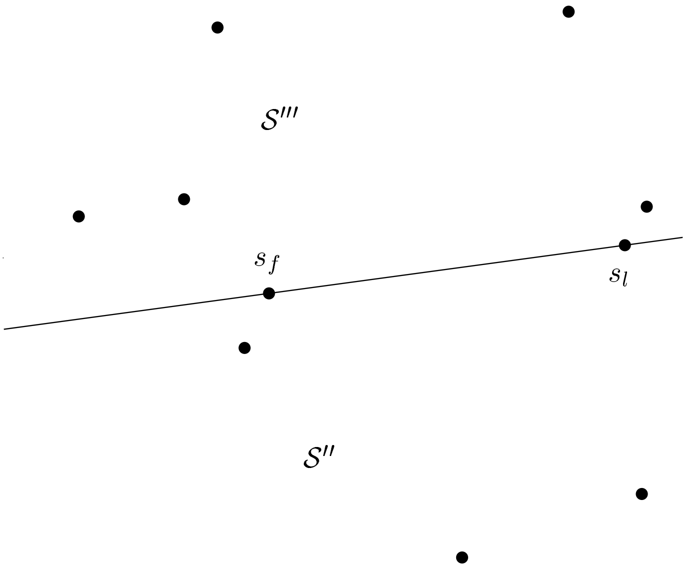


Figure 2.40: Dividing the initial set $\mathcal{S}$.

other point lies to the right of $\ell(v_1, v_3)$. The same is true for v_1 and v_5. Finally, for the initial points v_1 and v_2, the edge (v_2, v_3) would be no valid choice.

Now let us consider all possible second choices for v_2: Combined with v_3, v_4 or v_5, the edge (v_6, v_1) could not be generated. For v_2 and v_6, the edge (v_4, v_5) would not be possible.

2.5.1 Complexity of the Algorithm

For each phase, we have to find a random point and a random line ℓ through that point. Clearly, this can be done in constant time. Then we have to check for each point whether it lies left or right of ℓ. Again, this can be done in constant time for each point. Thus, if we have k points to handle in a phase then we need $\mathcal{O}(k)$ time. In the worst case, the input set is split in a way that one subset contains only two points, whereas the other subset contains $k-1$ of the k points. In this case, $\mathcal{O}(n)$ recursions are necessary. This leads to a total time complexity of $\mathcal{O}(n^2)$. The space complexity is $\mathcal{O}(n)$ as we can split each set by rearranging the points. However, if the set is split in two subsets of roughly the same size then the time complexity is about $\mathcal{O}(n \log n)$.

2.5.2 Trace of the Algorithm

For illustrating the algorithm, we traced the generation of a polygon on set $\mathcal{S}2$. This trace was generated with our implementation of Space Partitioning. Throughout the trace, lines of the polygonal chains are drawn solid and lines for splitting the sets

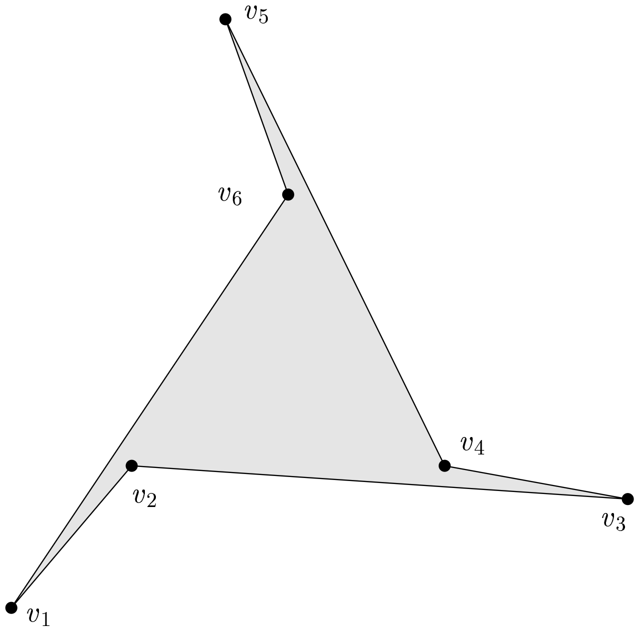


Figure 2.41: A polygon Space Partitioning will not generate.

are drawn dotted. (Of course, the lines used for splitting are restricted to the area which they split.)

First, we select two initial points at random: s_2 and s_9. Thus, we get the sets $\mathcal{S}'' = \{s_1, s_2, s_4, s_5, s_6, s_9\}$, which has s_2 as its first point and s_9 as its last point, and $\mathcal{S}''' = \{s_2, s_3, s_7, s_8, s_9, s_{10}\}$ with first point s_9 and last point s_2. This initial subdivision is depicted in Fig. 2.42. We will first trace the recursive subdivision of set $\mathcal{S}''$.

For splitting the set $\{s_1, s_2, s_4, s_5, s_6, s_9\}$, we choose on point at random: point s_4. A random line ℓ through s_4 splits the set $\mathcal{S}''$ into the set $\{s_1, s_2, s_4, s_5\}$, with first point s_2 and last point s_4, and into the set $\{s_4, s_6, s_9\}$, cf. Fig. 2.43.

Next, we have to divide the set $\{s_1, s_2, s_4, s_5\}$, where the first point and the last point are s_2 and s_4, respectively. We select point s_1 at random, and we also select a line through s_1 separating s_2 and s_4 at random, cf. Fig. 2.44. Thus, we get the two sets $\{s_1, s_2, s_5\}$ and $\{s_1, s_4\}$. For these sets, there exists only one way to generate the polygonal chains, and so we can add edges (s_2, s_5), (s_5, s_1) and (s_1, s_4) to chain $\mathcal{C}_1$.

Clearly, for the set $\{s_4, s_6, s_9\}$ the remaining subdivision and the resulting polygonal chain is straightforward. Thus, we can add edges (s_4, s_6) and (s_6, s_9) to the polygonal chain $\mathcal{C}_1$. This step is illustrated in Fig. 2.45.

After the generation of the chain for the points lying to the right of $\ell(s_2, s_9)$, we will now concentrate on the points lying to the left of $\ell(s_2, s_9)$. For dividing the set

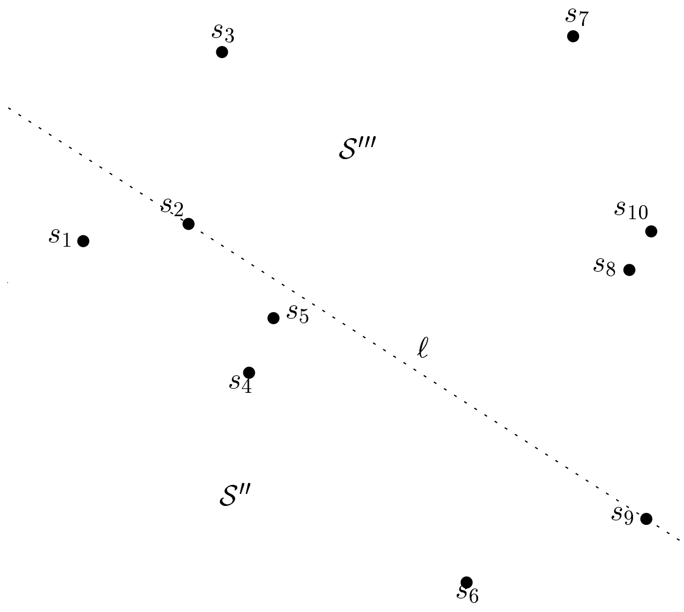


Figure 2.42: Initial division of the set $\mathcal{S}$.

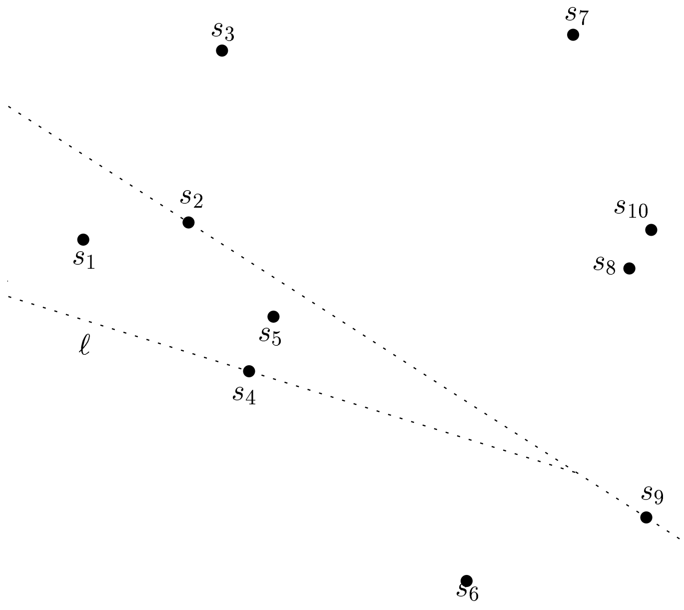


Figure 2.43: First step in splitting the points to the right.

$\mathcal{S}''' = \{s_2, s_3, s_7, s_8, s_9, s_{10}\}$ we select s_{10} and a line ℓ through s_{10} at random. Thus we get the sets $\{s_8, s_9, s_{10}\}$ and the set $\{s_2, s_3, s_7, s_{10}\}$. Again, the subdivision of set $\{s_8, s_9, s_{10}\}$ is straightforward, and thus, we can add the edges (s_9, s_8) and (s_9, s_{10}) to the chain $\mathcal{C}_2$. This phase is depicted in Fig. 2.46.

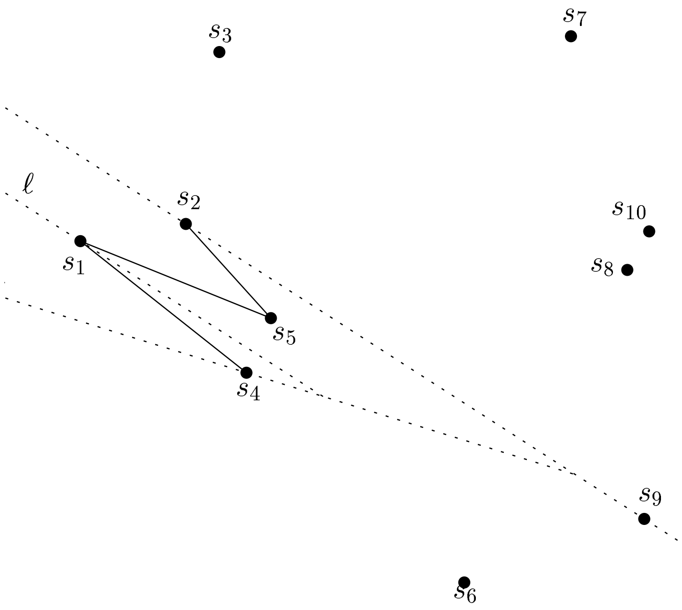


Figure 2.44: Second step in splitting the points to the right.

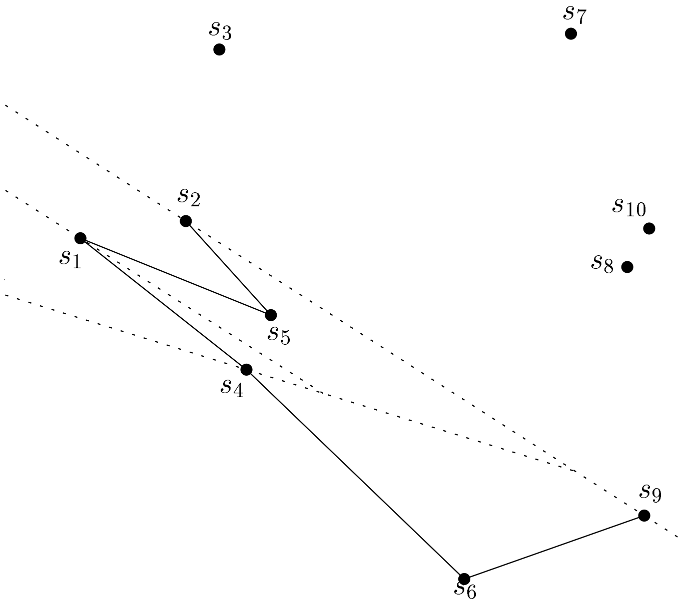


Figure 2.45: Completing the subdivision of the right side.

Next, we have to split the set $\{s_2, s_3, s_7, s_{10}\}$. By choosing point s_3 at random (and a random line through s_3), we get the sets $\{s_2, s_3\}$ and $\{s_3, s_7, s_{10}\}$, whose remaining subdivision is straightforward and thus we can complete the chain $\mathcal{C}_2$ with edges (s_{10}, s_7), (s_7, s_3), (s_3, s_2). This final phase is illustrated in Fig. 2.47. Finally,

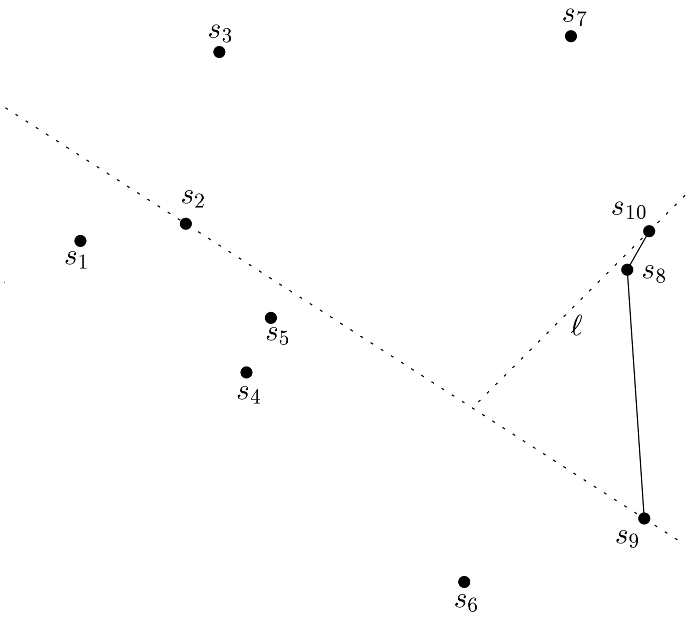


Figure 2.46: First step in splitting the points to the left.

we depicted the polygon resulting from algorithm Space Partitioning in Fig. 2.48.

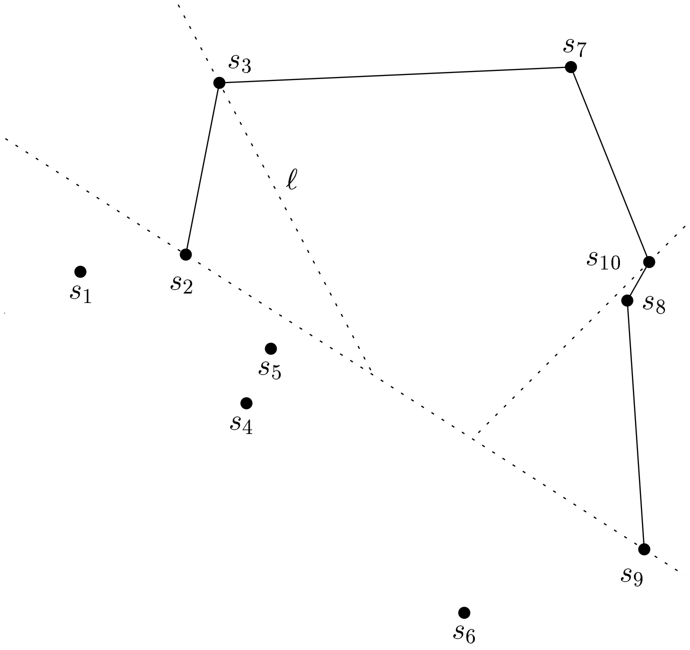


Figure 2.47: Completing the subdivision of the right side.

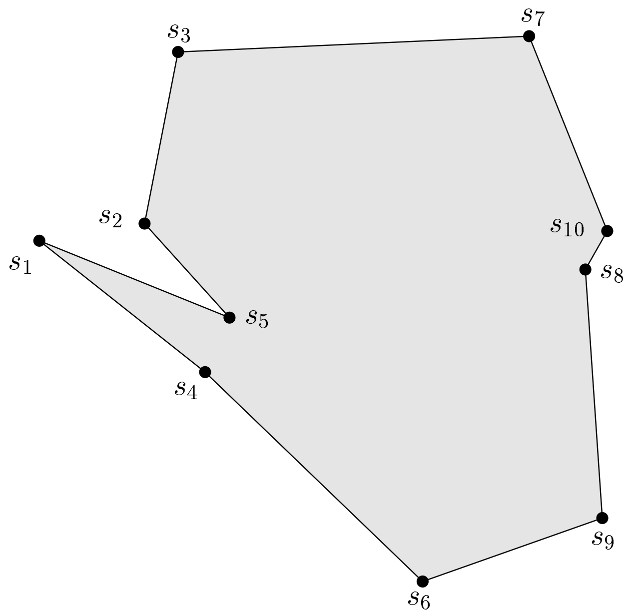


Figure 2.48: The resulting polygon.

2.6 Permute & Reject

For Permute & Reject, we start with creating a random permutation of the input points. This is done by first arranging the indices of the points in sorted order and then permuting them. This random permutation can be obtained by a shuffling algorithm as described by Knuth [Knu69]. (See the **for**-loop in the algorithm below.)

Having obtained a random permutation, we have to check whether the polygon corresponding to this permutation is simple. If it is simple then we output the polygon. Otherwise, a new random permutation is generated. Thus, the following algorithm will generate a simple polygon. (We denote the n-tuple of the indices by $\mathcal{H}$ and its elements by h.)

Algorithm PermuteReject($\mathcal{S}$)

```
1.  Init the $n$-tuple $\mathcal{H}$ of indices of points of $\mathcal{S}$.
2.  $n := |\mathcal{H}|$.
3.  repeat
4.     for $i := 1$ to $n$ do
5.        Generate a random integer number $x \in \{i, \dots, n\}$.
6.        Swap $h_i$ and $h_x$.
7.  until $(s_{h_1}, s_{h_2}, \dots, s_{h_n})$ describes a simple polygon.
```

The shuffling algorithm will generate a random permutation with a uniform distribution: For the first swap, there are n choices for the element to swap with;

in general, for the i-th swap there are $n-i+1$ possible choices. Thus, every one of the $n!$ permutations is generated with probability $\frac{1}{n!}$. As a consequence, Permute & Reject generates simple polygons with a uniform distribution.

For the theoretically best worst-case complexity, we could use the algorithm by Chazelle [Cha91] for testing whether the polygon corresponding to some random permutation $\mathcal{H}$ is simple. However, for a more practical approach one uses a less sophisticated algorithm as the one by Bentley-Ottmann, cf. Preparata and Shamos [PS85].

2.6.1 Complexity of the Algorithm

The initialization of the n-tuple $\mathcal{H}$ is done in linear time. Next, the shuffling algorithm generates a random permutation in linear time and with linear space. By using the algorithm of Chazelle, the test for simplicity can be done in linear time and space, whereas the more practical approach by Bentley-Ottmann requires $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

However, the performance of the algorithm depends mainly on the number of polygons that have to be generated in order to encounter a simple one. Experimental results for this relation are given in Chapter 4. Thus, the time complexity is $\mathcal{O}(n \cdot \vartheta(n))$ when using Chazelle's algorithm and $\mathcal{O}(n \log n \cdot \vartheta(n))$ otherwise, where $\vartheta(n)$ is a function that reflects the ratio of simple polygons to polygons. Further, the space complexity of the algorithm is $\mathcal{O}(n)$.

2.7 2-Opt Moves

The basic idea for 2-Opt Moves, which was devised by Zhu et al. [ZSSM96], is as follows: First, generate a random polygon and then remove all the self-intersections which occur in this polygon. The generation of the random polygon is done with the same method as in algorithm Permute & Reject. Then, we compute all the intersections that occur in the random polygon and store them. Obviously, an intersection is represented by a pair of edges. We do the following until the set of intersections is empty:

1. We choose one intersection at random. Let (v_i, v_{i+1}) and (v_j, v_{j+1}) be the two edges of the intersection.
2. We remove from the set of intersections all intersections that have either (v_i, v_{i+1}) or (v_j, v_{j+1}) as one of their edges.
3. We replace (v_i, v_{i+1}) and (v_j, v_{j+1}) with the new edges (v_i, v_j) and (v_{i+1}, v_{j+1}) (this is depicted in Fig. 2.49).
4. We compute the intersections of (v_i, v_j) and (v_{i+1}, v_{j+1}) with the remaining $n-2$ edges and add them to the set of intersections.

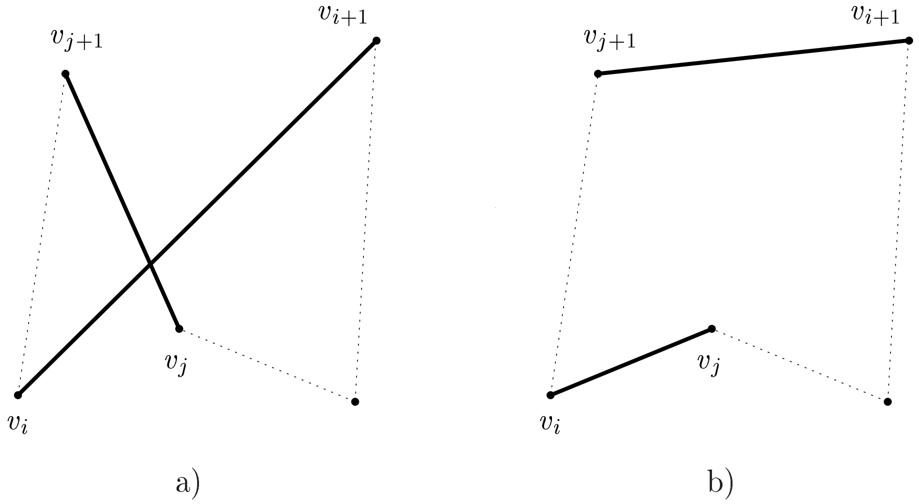


Figure 2.49: An example for a 2-opt move.

The replacement operation in Step 3 is called a *2-opt move*, and a phase of the algorithm denotes the execution of Step 1 through Step 4.

In order to generate a simple polygon with the scheme outlined above, one has to ensure that the total number of 2-opt moves to be applied is always bound. Van Leeuwen and Schoone [vLS82] showed that at most $\mathcal{O}(n^3)$ many 2-opt moves are necessary in order to obtain a simple polygon: They consider all lines defined by two points $s_i, s_j \in \mathcal{S}$. For each edge e of the polygon, they count the lines that intersect e. Further, they count all[11] lines supporting e twice. Let c_e denote this number. Clearly, the sum of c_e over all edges e in the current polygon indicates the number of all (possible) intersections. The number of lines on $\mathcal{S}$ is bound by $\mathcal{O}(n^2)$, and therefore the sum of c_e over all n edges e of the polygon is bound by $\mathcal{O}(n^3)$. Now they consider two intersecting edges $f_1 := (v_i, v_{i+1})$ and $f_2 := (v_j, v_{j+1})$, which are replaced by edges $e_1 := (v_i, v_j)$ and $e_2 := (v_{i+1}, v_{j+1})$. The edges are depicted in Fig. 2.50. First, they argue that any line ℓ not supporting any of the four edges which intersects either e_1 or e_2 will also intersect either f_1 or f_2. Similarly, if ℓ (where ℓ does not support any of the four edges) intersects both e_1 and e_2, it must also intersect both f_1 and f_2. Further they show that the number of hits decreases at least by two when considering all the lines that support any of the four edges. Thus, with each 2-opt move, the sum of c_e over all edges e in the polygon is decremented and after at most $\mathcal{O}(n^3)$ 2-opt moves, the resulting polygon will be simple. Thus, we can now state Algorithm 2-OptMoves. In the algorithm, $\mathcal{J}$ denotes the set of intersections, and $j := (f_1, f_2)$ is the intersection of edges f_1 and f_2.

2-Opt Moves will produce all possible polygons, but not with a uniform distribution: As shown in Zhu et al. [ZSSM96], there exist polygons which are obtained by a unique series of 2-opt moves, whereas other polygons may be obtained by several different series of 2-opt moves. This is illustrated in Fig. 2.51 and in Fig. 2.52,

---

[11] In the work by Van Leeuwen and Schoone it is not assumed that the points are in general position. Thus, for collinear points, multiple supporting lines exist.

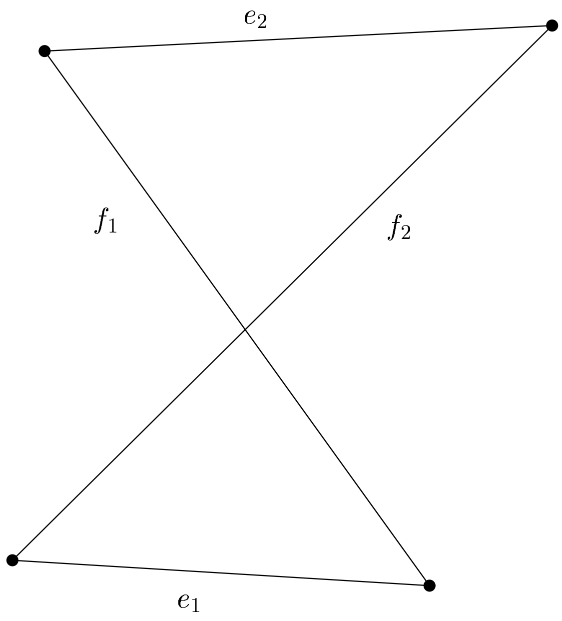


Figure 2.50: A pair of intersecting edges and their replacement.

Algorithm 2-OptMoves($\mathcal{S}$)

```
Compute a random polygon $\mathcal{P}$ on $\mathcal{S}$.
Compute the intersections in $\mathcal{P}$.
Insert the intersections of $\mathcal{P}$ into $\mathcal{J}$.
while $\mathcal{J} \neq \{\}$ do
    Select an intersection $j = (f_1, f_2)$ at random in $\mathcal{J}$.
    Remove all intersections of $f_1$ and any other edge of $\mathcal{P}$ from $\mathcal{J}$.
    Remove all intersections of $f_2$ and any other edge of $\mathcal{P}$ from $\mathcal{J}$.
    Remove the edges $f_1$ and $f_2$ from $\mathcal{P}$.
    Insert the new edges $e_1$ and $e_2$ into $\mathcal{P}$.
    Compute all intersections between $e_1$ and any other edge of $\mathcal{P}$.
    Compute all intersections between $e_2$ and any other edge of $\mathcal{P}$.
    Insert the computed intersections into $\mathcal{J}$.
```

which were both taken from Zhu et al. [ZSSM96]: Whereas the polygon depicted in Fig. 2.51 cannot be obtained from any other polygon by means of 2-opt moves, the polygon depicted in Fig. 2.52a can be obtained from the one in Fig. 2.52b by a 2-opt move.

2.7.1 Complexity of the Algorithm

We opted for storing the intersections in an array. In addition, we store for each edge e the indices of its intersections[12] in the intersection array. For each phase, we

---

[12] I.e., intersections between edge e and any other edge in the polygon.

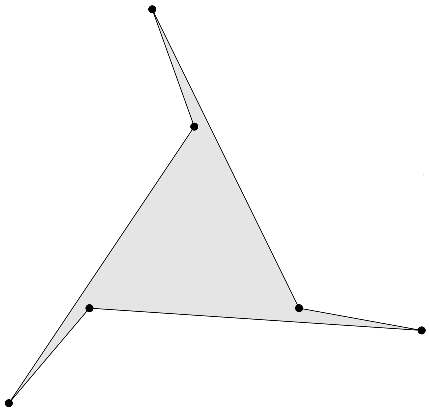
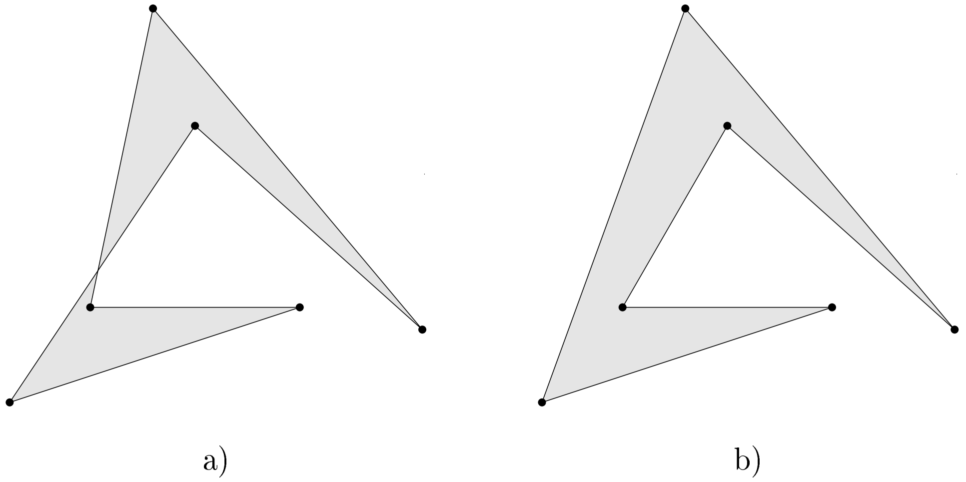


Figure 2.51: A polygon that cannot be obtained from any non-simple polygon by 2-opt moves.



a) b)

Figure 2.52: The right polygon is obtained from the left one by a 2-opt move.

need to randomly select a pair of edges f_1, f_2 which intersect. This can be done in constant time. Further, we have to remove all the other intersections of either f_1 or f_2 with any other edge of the polygon from the array. This can also be done in constant time for each intersection, if we keep track of the intersections for each edge. Note that the number of intersections per edge is bound by $\mathcal{O}(n)$. Next, we have to compute the intersections of the new edges e_1 and e_2 with all the other edges of the polygon, which also can be done in linear time. According to Van Leeuwen and Schoone [vLS82], we need at most $\mathcal{O}(n^3)$ many 2-opt moves (and thus $\mathcal{O}(n^3)$ phases) to obtain a simple polygon. Thus, the total time complexity is bound by $\mathcal{O}(n^4)$, because we need linear time for each phase. Clearly, the space complexity is bound by $\mathcal{O}(n^2)$, because we have to store the polygon and all self-intersections that occur in the polygon.

2.7.2 Trace of the Algorithm

We illustrate algorithm 2-Opt Moves on set $\mathcal{S}2$. The output was generated by our implementation of 2-Opt Moves. In Fig. 2.53 we depict the initial random polygon. It contains the following intersections: $((s_1, s_7), (s_3, s_9))$, $((s_2, s_6), (s_4, s_5))$, $((s_3, s_9), (s_5, s_8))$, $((s_3, s_9), (s_6, s_7))$ and $((s_5, s_8), (s_6, s_7))$.

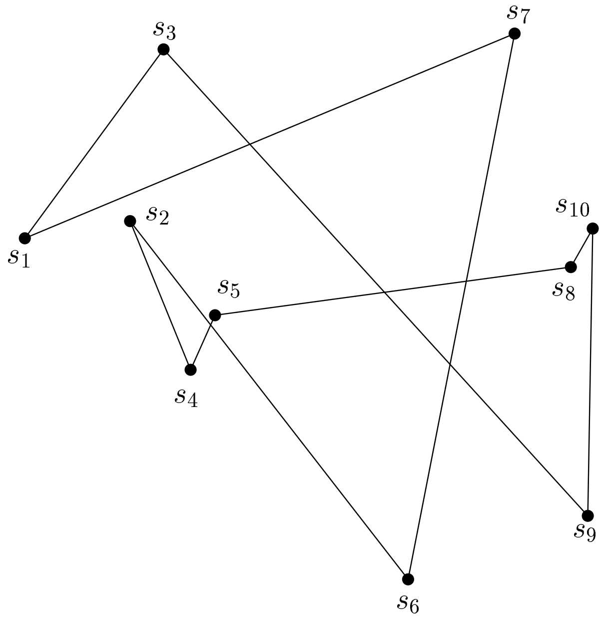


Figure 2.53: The initial random (non-simple) polygon

In the first phase, we select intersection $((s_3, s_9), (s_6, s_7))$ at random. Thus we have to remove intersections $((s_1, s_7), (s_3, s_9))$, $((s_3, s_9), (s_5, s_8))$ and $((s_5, s_8), (s_6, s_7))$ from the set of intersection, and we replace edges (s_3, s_9) and (s_6, s_7) with edges (s_3, s_6) and (s_7, s_9). For these two edges, we have to add intersections $((s_1, s_7), (s_3, s_6))$, $((s_3, s_6), (s_5, s_8))$ and $((s_5, s_8), (s_7, s_9))$ to the set of intersections. The 2-opt move of Phase 1 is depicted in Fig. 2.54.

The next intersection we selected for removal is intersection $((s_3, s_6), (s_5, s_8))$. Thus, the following intersections have to be removed from the set of intersections: $((s_1, s_7), (s_3, s_6))$ and $((s_5, s_8), (s_7, s_9))$. Further, we remove edges (s_3, s_6) and (s_5, s_8). We add the new edges (s_3, s_5) and (s_6, s_8) to $\mathcal{P}$, and we add the intersections $((s_1, s_7), (s_3, s_5))$ and $((s_6, s_8), (s_7, s_9))$ to the set of intersections. The 2-opt move of this phase is depicted in Fig. 2.55.

For the third phase, intersection $((s_6, s_8), (s_7, s_9))$ is selected at random. We replace the edges (s_6, s_8) and (s_7, s_9) by the new edges (s_6, s_9) and (s_7, s_8). Since these two edges do not intersect any other edges of $\mathcal{P}$, nothing is added to the set of intersections, cf. Fig. 2.56.

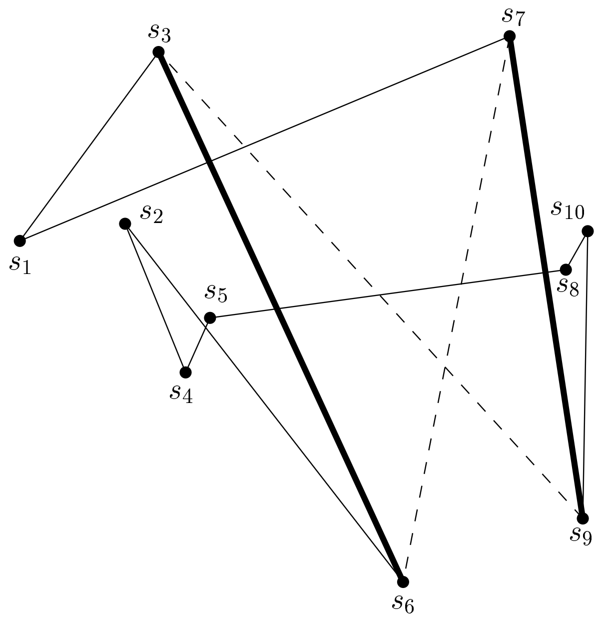


Figure 2.54: Phase 1 of 2-Opt Moves

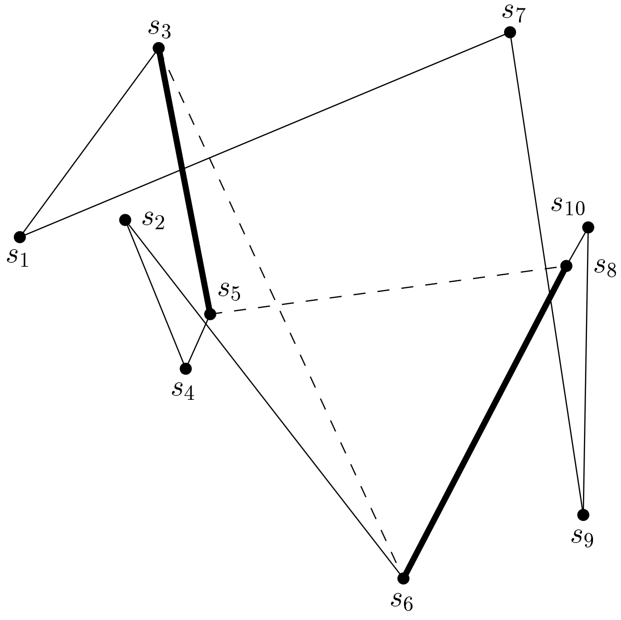


Figure 2.55: Phase 2 of 2-Opt Moves

Next, we select intersection $((s_2, s_6), (s_4, s_5))$. When inserting the new edges (s_2, s_5) and (s_4, s_6) no new intersections are created. This is depicted in Fig. 2.57. The next intersection to be removed is intersection $((s_1, s_7), (s_3, s_5))$. Since the edges (s_1, s_7) and (s_3, s_5) do not intersect any other edges of $\mathcal{P}$, we do not need to remove

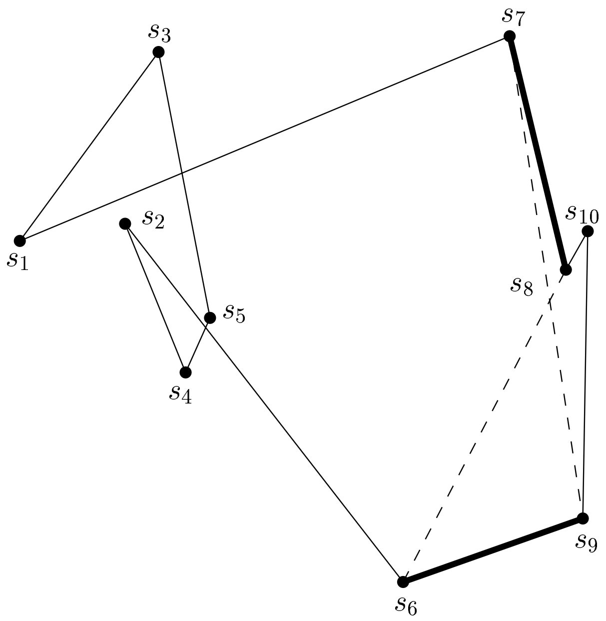


Figure 2.56: Phase 3 of 2-Opt Moves

any other intersections from the set of intersections. However, when inserting the new edges (s_1, s_5) and (s_3, s_7) in $\mathcal{P}$, we have to add the intersection $((s_1, s_5), (s_2, s_4))$ to the set of intersections, cf. Fig. 2.58.

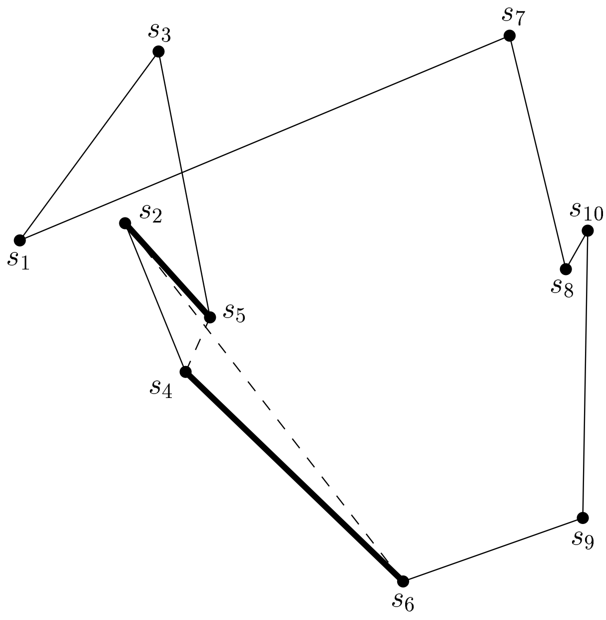


Figure 2.57: Phase 4 of 2-Opt Moves

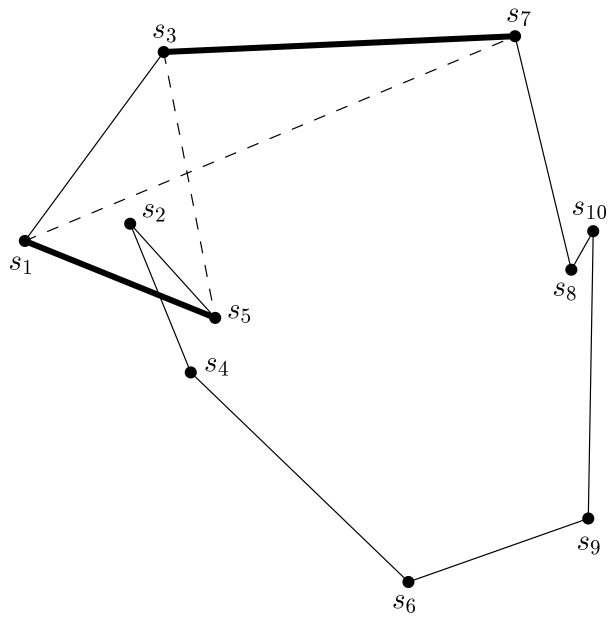


Figure 2.58: Phase 5 of 2-Opt Moves

Thus, we have only the intersection between the edges (s_1, s_5) and (s_2, s_4) left for removal. We replace these edges by the new edges (s_1, s_2) and (s_5, s_5), cf. Fig. 2.59. This yields the simple polygon depicted in Fig. 2.60.

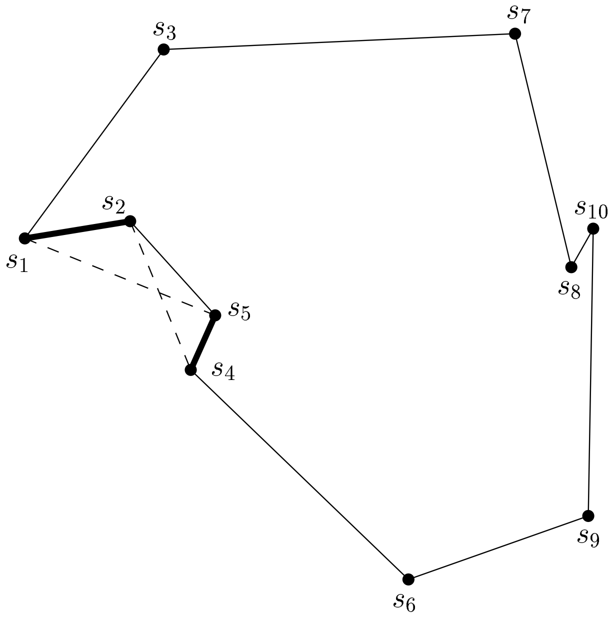


Figure 2.59: Phase 6 of 2-Opt Moves

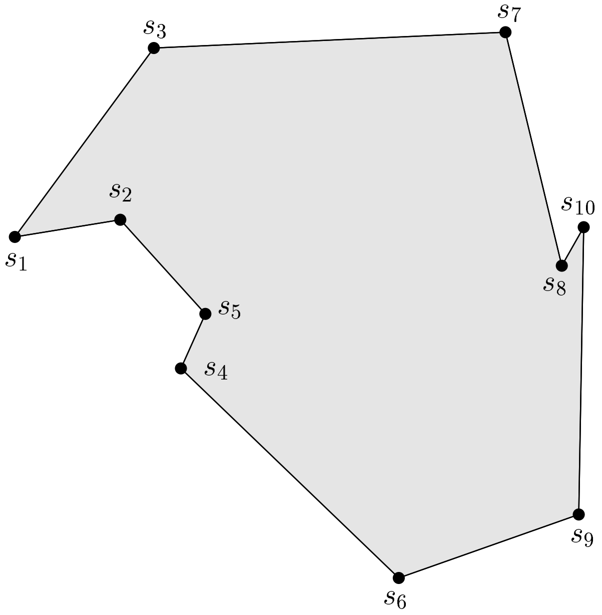


Figure 2.60: The polygon resulting from 2-Opt Moves

2.8 Incremental Construction & Backtracking

Finally, we studied an approach based on exhaustive search and backtracking which is akin to the work by Shuffelt and Berliner [SB94]. We incrementally construct a polygon by adding randomly selected points. If the resulting polygonal chain is not simple any more or if it is clear that we will not be able to complete the chain to a simple polygon then we have to apply backtracking. Shuffelt and Berliner studied the Knight's Tour Construction (KTC) problem, which is known to be $\mathcal{NP}$-hard. However, they tried to find conditions that reduce the backtracking necessary to find a solution. We adopted some of the conditions they used to solve the KTC problem in order to find a simple polygon. We did not use all of their conditions, as some of them cover only few cases but require a large amount of time (sometimes even more than $\mathcal{O}(n^5)$).

Before we present the conditions we used for reducing the amount of backtracking, we give the basics needed to check a violation of the conditions. For checking the conditions, we have to keep track of all edges that are still usable for completing the polygon. We start with the complete graph on $\mathcal{S}$. Basically, all edges of the graph that do not intersect with any edge already in the polygonal chain and that have endpoints which are not vertices (except for the first and for the last vertex) of the chain, are usable. Let s be the last vertex in the chain. Clearly, when we add another point[13] to the chain, all the edges that have s as vertex are no longer usable. We will mark all edges that are not usable. In the following, two points are called "adjacent" if an unmarked (i.e., usable) edge connecting these two points exists.

---

[13] Note that adding a point to the chain also implies adding an edge to the chain, and vice versa.

Obviously, if only two unmarked edges are incident upon a point then both edges need to be part of the resulting polygon $\mathcal{P}$. Thus, we call these edges "mandatory" edges. Further, if a point has two mandatory edges[14], all other edges that have this point as vertex are no longer usable. In addition, we can mark all edges as unusable that intersect one of the mandatory edges. Note that the use of an edge we have marked as unusable would lead to self-intersections in the resulting polygon as this edge intersects an edge already in the polygon.

Lemma 2.9 Each point that does not yet belong to the polygonal chain under construction needs to have at least two incident usable edges. (Otherwise, we will not be able to generate a polygon without self-intersections.)

Proof: Assume that there exists a point s that has at most one usable edge incident. Clearly, each point in the resulting polygon $\mathcal{P}$ has two edges incident. Thus, at least one of the two edges of $\mathcal{P}$ which are incident on s must have been marked as unusable. As stated above, the use of marked edges leads to self-intersections in the resulting polygon. $\square$

Lemma 2.10 At most one point adjacent to the point s last added may have only two incident unmarked edges. (Otherwise, we will not be able to generate a polygon without self-intersections.)

Proof: Without loss of generality, assume that we add s_i to $\mathcal{S}$ before dealing with s_j. (If s_i is to become part of the polygon then it needs to be linked to s.) However, this would render the edge (s, s_j) unusable, which contradicts Lemma 2.9. $\square$

Lemma 2.11 Points that lie on the boundary of $\mathcal{CH}(\mathcal{S})$ must appear in the polygonal chain in the same relative order as on the hull. (Otherwise, we will not be able to generate a polygon without self-intersections.)

Proof: Let us assume that we generate the polygon in CCW direction. Let v_i and v_{i+1} be two consecutive vertices of the convex hull. Further, let v_k (with $k \neq i$ and $k \neq i+1$) be a vertex of the convex hull such that the order of these three vertices in the polygon is $v_i < v_k < v_{i+1}$. Consider the vertex v' following v_i in the polygon $\mathcal{P}$ and the preceding vertex v''. Due to the CCW orientation, v'' lies to the left of the oriented line $\ell(v_i, v')$. Further, since v_i and v_{i+1} form an edge of the convex hull, v' lies left of the oriented line $\ell(v_i, v_{i+1})$, which implies that v_{i+1} lies right of the line $\ell(v_i, v')$. Further, we note that v' lies on the polygonal chain $\mathcal{C}$ from v_i to v_k, whereas v'' and v_{i+1} lie on the other chain from v_k to v_i. The chain $\mathcal{C}$ partitions $\mathcal{CH}(\mathcal{S})$ into a "left" and a "right" set. Thus, any polygonal chain connecting a point lying to the left of $\mathcal{C}$ and a point lying to the right of $\mathcal{C}$ necessarily intersects $\mathcal{C}$. Hence, the chain from v_k to v_i must intersect $\mathcal{C}$, since it contains both v'' and v_{i+1}. $\square$

---

[14] Consider a point that is adjacent to two vertices that each have only two unmarked vertices.

Algorithm Incremental Construction & Backtracking checks the conditions that can be derived from the lemmas stated above in the same order as the lemmas are given. Naturally, it generates every possible simple polygon on $\mathcal{S}$ with positive probability. Note that the algorithm will generate polygons in both CCW and CW orientation.

2.8.1 Complexity of the Algorithm

The complexity of the algorithm depends mainly on the amount of backtracking necessary. The condition stated in Lemma 2.9 can be checked in linear time; the condition of Lemma 2.10 also requires $\mathcal{O}(n)$ time and the last condition (cf. Lemma 2.11) can be checked in constant time if we keep track of the last vertex of the hull we added to the chain. Further, we have to check whether any point has only two incident edges, which requires $\mathcal{O}(n)$, and we have to mark all edges intersecting a mandatory edge. This requires $\mathcal{O}(n^2)$ time per mandatory edge (Note that there are at most n mandatory edges). Finally, for each point with two mandatory edges we have to mark all other edges incident on this point. For all points, this can be done in $\mathcal{O}(n^2)$ time. Thus, all conditions can be checked in $\mathcal{O}(n^3)$ per phase.

We have to keep track of all edges that are unusable, and thus the space requirement of the algorithm is bound by $\mathcal{O}(n^2)$.

2.8.2 Trace of the Algorithm

We will trace this algorithm on set $\mathcal{S}2$. Since we want our polygon to start with vertex index 1, s_1 is the first vertex of our polygon. Next we select point s_5 at random and add edge (s_1, s_5) to the chain. This is depicted in Fig. 2.61. The next point added at random is s_8, cf. Fig. 2.62.

Then we select s_9 at random as next point to be added. Unfortunately, both s_1 and s_9 are vertices of the convex hull, but they are not consecutive vertices. Thus, we have to select another point. The next random choice is point s_7, for which the same comment applies. Again, we have to select another random point, namely s_2. This phase is depicted in Fig. 2.63.

In the fourth phase, the first point chosen at random is s_7 again. However, as in Phase 3, s_7 and s_1 are not consecutive vertices of $\mathcal{CH}(\mathcal{S})$. The next random choice, s_3, is a vertex of the hull, and s_1, s_3 are consecutive vertices of $\mathcal{CH}(\mathcal{S})$. Thus, point s_3 is a valid choice, cf. Fig. 2.64.

In the next phase, only points s_7 and s_{10} are adjacent to s_3, and s_{10} is chosen at random. However, adding s_{10} violates Lemma 2.11, and thus s_7 is added to the chain. This is depicted in Fig. 2.65.

In the next phase, there is no real choice: The only point that is connected to s_7 by a usable edge is s_{10}, cf. Fig. 2.66. From s_{10}, there are two choices for extending the chain, namely s_9 and s_6, from which s_9 is chosen at random. This is depicted in Fig. 2.67.

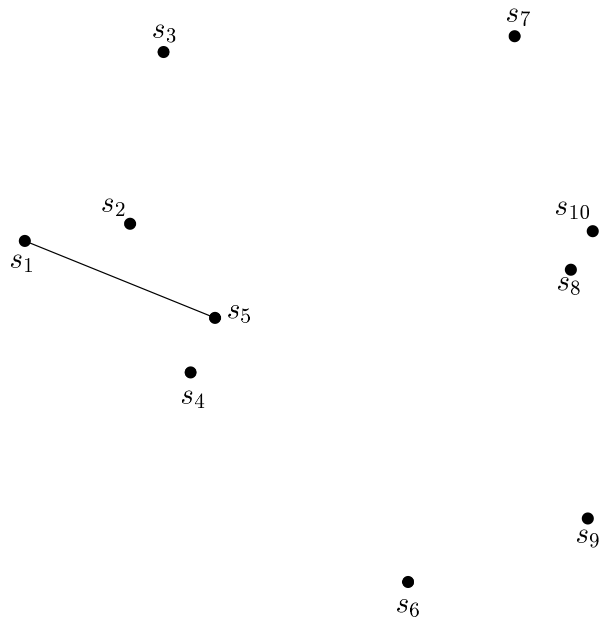


Figure 2.61: Phase 1 of Incremental Construction & Backtracking.

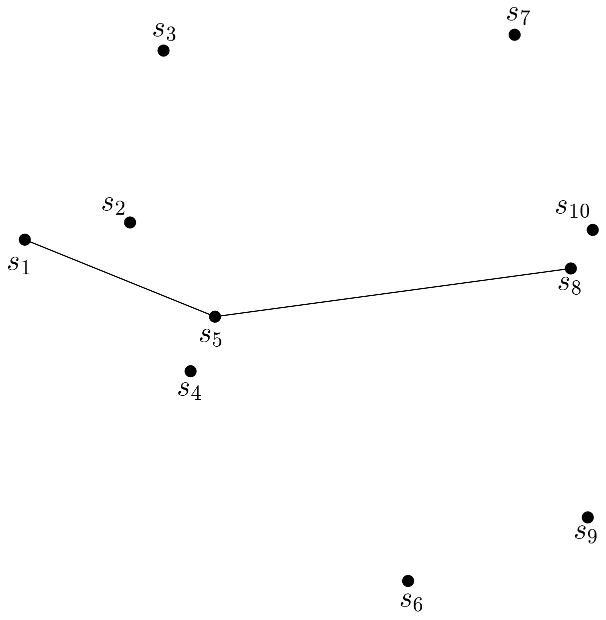


Figure 2.62: Phase 2 of Incremental Construction & Backtracking.

For the next phase, there are again two choices: both s_4 and s_6 are adjacent to s_9. However, s_6 is chosen at random, cf. Fig. 2.68. Thus, only s_4 is left for completing the polygon, cf. Fig. 2.69. The resulting polygon is depicted in Fig. 2.70. Note that the polygon was generated in CW order!

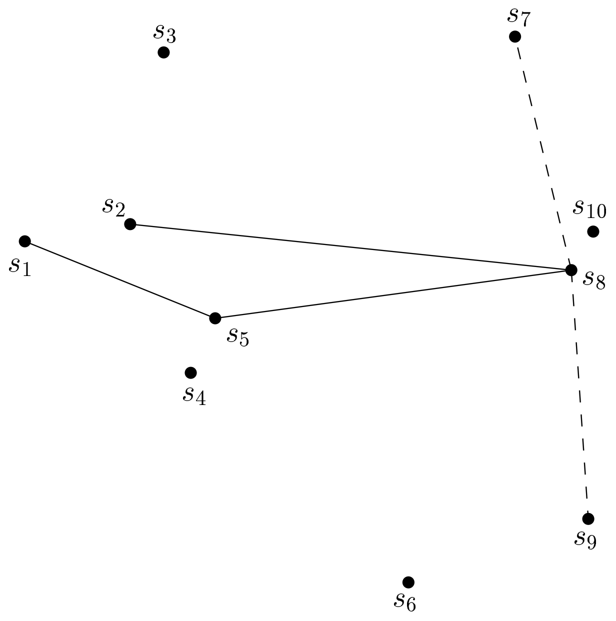


Figure 2.63: Phase 3 of Incremental Construction & Backtracking.

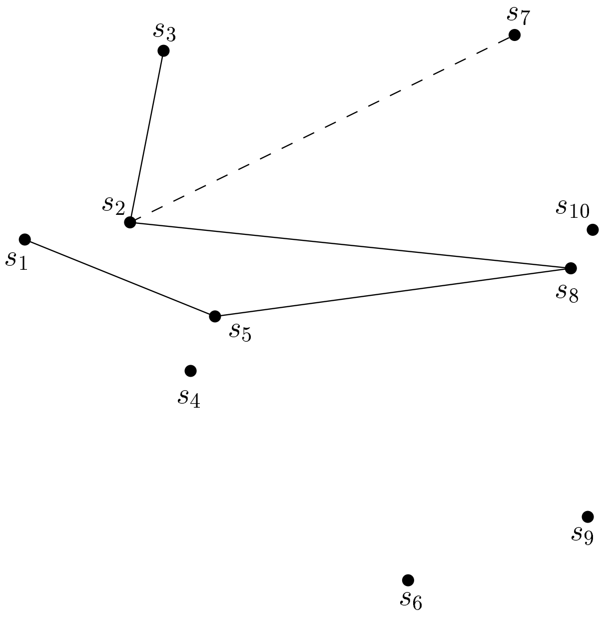


Figure 2.64: Phase 4 of Incremental Construction & Backtracking.

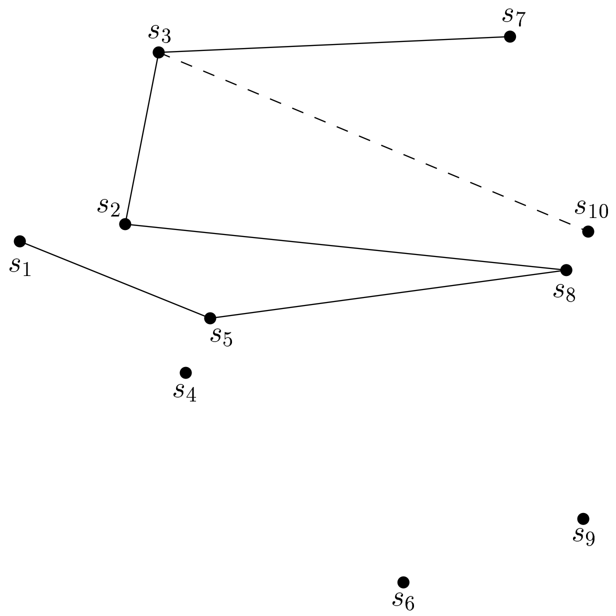


Figure 2.65: Phase 5 of Incremental Construction & Backtracking.

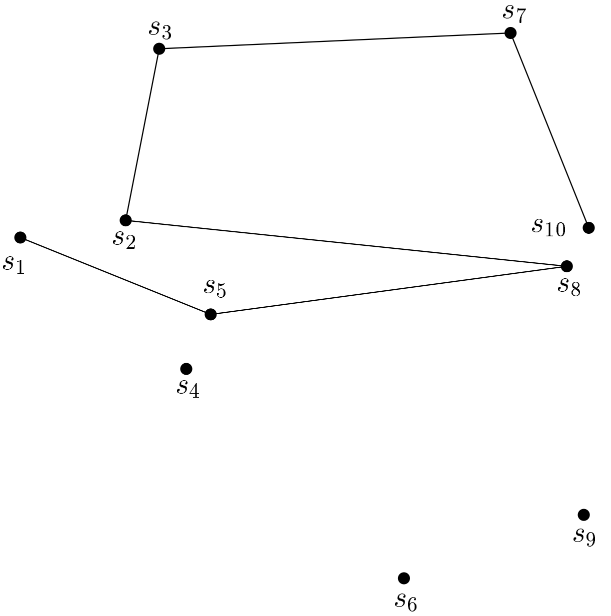


Figure 2.66: Phase 6 of Incremental Construction & Backtracking.

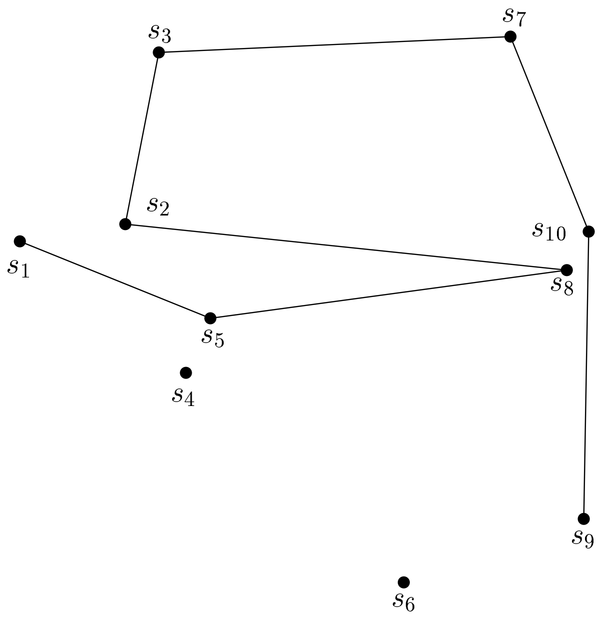


Figure 2.67: Phase 7 of Incremental Construction & Backtracking.

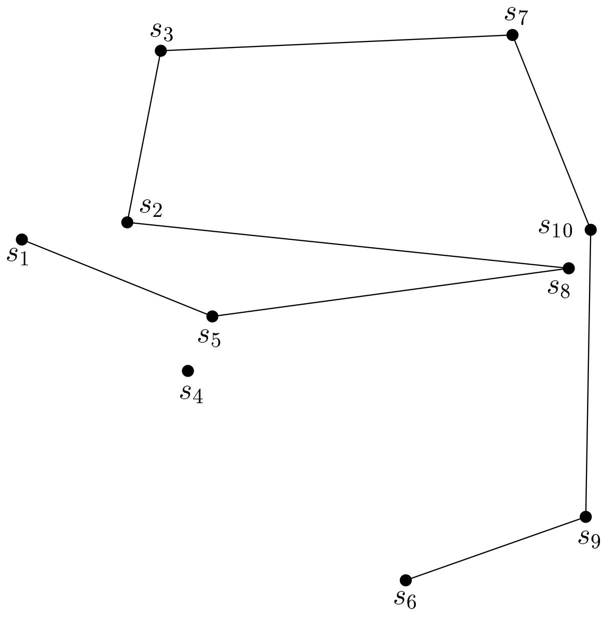


Figure 2.68: Phase 8 of Incremental Construction & Backtracking.

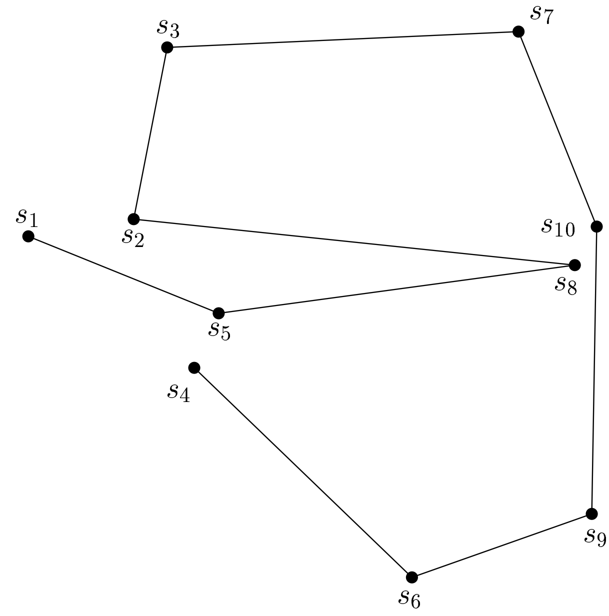


Figure 2.69: Phase 9 of Incremental Construction & Backtracking.

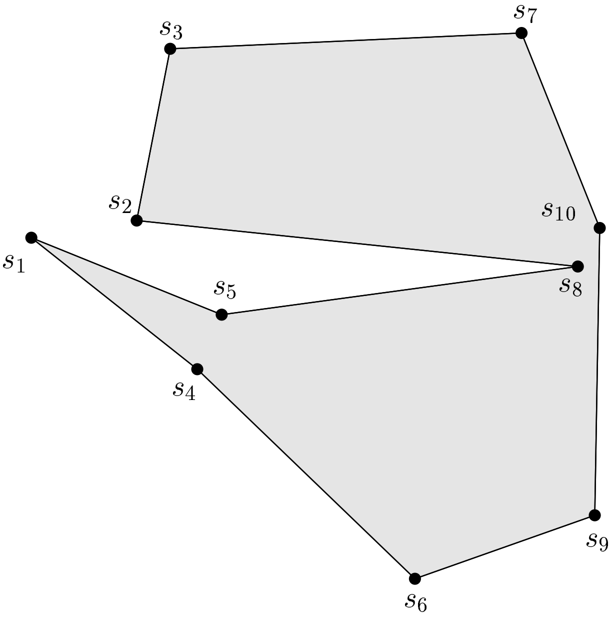


Figure 2.70: The polygon $\mathcal{P}$ generated by Incremental Construction & Backtracking.

Chapter 3

Implementation

The major goal of this chapter is to warn about particular pitfalls and to provide sufficient information about implementational details such that an experienced graduate-level programmer could re-implement our algorithms. (This chapter does not provide a crash course on software engineering, though.)

First, we describe the general principles of the implementation, then the basic algorithms (such as computing a convex hull) are discussed, and finally, we state implementational details of the algorithms.

3.1 General Principles

We implemented the algorithms using the programming language "C" as specified by ANSI. The choice of the programming language was limited to "C", "C++" or "Objective C" since we implemented the algorithms under UNIX and we wanted our implementation to be portable and compatible with existing code and software libraries. Of course, there are a lot of other programming languages available for UNIX platforms, such as Ada or Oberon, which may both be better suited for software engineering than C. However, C is the only programming language which can be expected to be available on almost any UNIX platform.

For generating the results presented in Chapter 4, we implemented a testbed. This testbed consists of several executables, most of which were apted for producing very specific output, e.g., computing 10.000 polygons on 10 sets of 10 points each. In addition, we implemented a graphical user interface which allows the user to specify the points of $\mathcal{S}$ either by interactively adding them with the mouse or by calling a random number generator. Then, all the algorithms described in Chapter 2 can be run on $\mathcal{S}$.

For the generation of the random numbers, we use the *rand48* random number generator. This is a linear congruential generator[1], which uses 48-bit integer arithmetic. The maximum number generated is $2^{31}-1$, i.e., 2,147,483,647. We chose this generator because it generates a sufficiently large range of pseudo random numbers

---

[1] For information on random number generators, cf. [L'E94].

and it is included in the standard UNIX C library. By default, we initialize the generator with seed 0.

3.1.1 User Interface

In addition to the test bed for the algorithms, we also implemented a graphical user interface using the *Forms* library by Zhao and Overmars [ZO95]. A sample screen shot of the user interface is depicted in Fig. 3.1. Its basic design was created with *fdesign*[2]. The user interface consists of three parts: Pull down menus, drawing canvas and status display

Pull-down Menus On the top of the window, the pull-down menus ("General", "Points", "Zoom" and "Help") reside. We give a short description of the options offered by each menu. The **"General"** menu supports the following operations:

Reset Generator resets the random number generator, i.e., calls *srand48* with a fixed seed[3]. This is useful for reproducing either point sets or polygons.

Set Seed is used for manually setting the seed of the random number generator.

Toggle Polygon Fill selects the drawing style: The polygon is either displayed filled in dark gray or it is displayed in outline mode (where only its boundary is drawn).

Write Ipe File writes a file that can be read by the Ipe drawing editor of Schwarzkopf, cf. Schwarzkopf [Sch94].

Write Voronoi File writes the current polygon in a format that is understood by Held's Voronoi program, cf. Held [Hel93].

Quit exists the program and returns control to the calling process.

Next, the **"Points"** menu offers the following options:

Load Points allows the user to load points from a file. (The data format of such a file is described in Section 3.3.)

Save Points saves the current points to a file.

Random Points generates the desired number of random points.

Add Points allows the user to manually add the points with the mouse. Click on the left button to add a point, and on the right to finish adding points.

Delete Points is for deleting points. The left mouse button is for actually deleting the points, whereas a click on the right button ends the delete mode.

Clear Points deletes all current points.

---

[2] fdesign is a program for interactively designing dialogue forms for use with the Forms library.

[3] In the current version, the default seed is 0.

Clearly, when the points are modified (i.e., new points are added, or points are deleted), any polygon which was computed on these points is no longer valid and is thus deleted.

The third menu is **"Polygons"**. It offers the following options:

Load Polygon loads a polygon from a file, where the file has to be formatted as described in Section 3.3.

Save Polygon writes the current point set $\mathcal{S}$ and the current polygon to a file.

Bouncing Vertices invokes algorithm Bouncing Vertices on the current point set $\mathcal{S}$.

x-Monotone generates an x-monotone polygon on $\mathcal{S}$.

Arrange Star generates a star-shaped polygon on $\mathcal{S}$ by means of Arrange Star.

Star Universe generates a star-shaped polygon uniformly at random on $\mathcal{S}$.

Quick Star invokes the fast heuristic Quick Star for generating a random star-shaped polygon.

Steady Growth invokes the algorithm Steady Growth for the random generation of a simple polygon.

2-Opt Moves generates a simple polygon on $\mathcal{S}$ by means of 2-Opt Moves.

Space Partitioning generates a simple polygon on $\mathcal{S}$ by means of Space Partitioning.

Permute & Reject invokes algorithm Permute & Reject on $\mathcal{S}$.

Incremental Construction uses Incremental Construction & Backtracking for generating a simple polygon.

The next menu in the top of the window is **"Zoom"** which offers the following options:

Original Scale sets the zoom region to the original settings. In our implementation, the original region displayed is $[0, 1]^2$.

Best Fit selects a zoom region such that the bounding box of the points covers at most 90% of the plane in each direction. The points are displayed centered.

Zoom In zooms in on the center and reduces the area displayed by 10% in each direction.

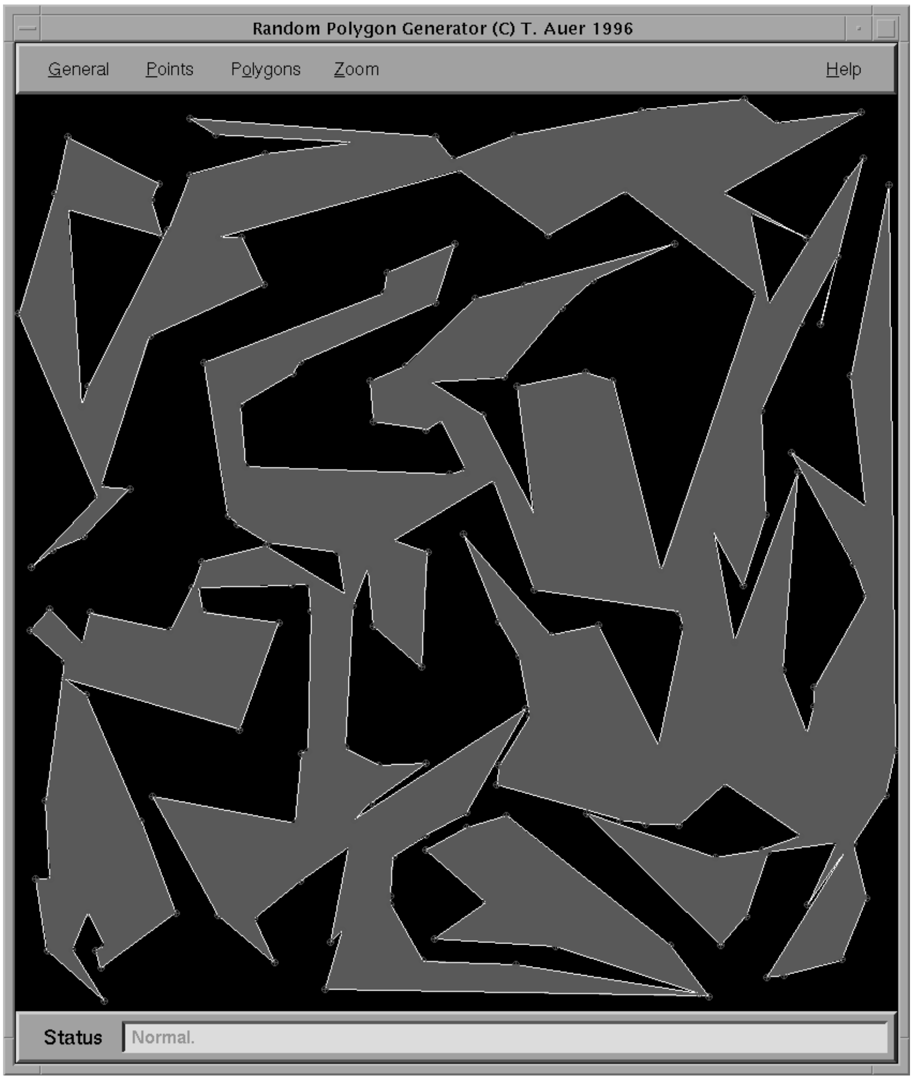
Zoom Out does the opposite as "Zoom In", i.e., the displayed area is enlarged by 10% in each direction.

Select Zoom Region allows the user to specify two points (i.e., a rectangle) that indicate the area to be displayed. The longer side of the rectangle specifies the size of the zoom area.

Finally **"Help"** presents information on the author of the software, and copyright issues.

Drawing Canvas The large black canvas filling the center of the window is used to display both the points and the polygon. The points are displayed as red circles with an inscribed red cross. The polygon is displayed either only with a white outline, or the interior is filled in a dark gray in addition to the outline. The area displayed is normally the unit square $[0, 1]^2$. The points are bound to this region, i.e., no point outside $[0, 1]^2$ is accepted by our implementation.

Status Display Finally, the bottom of the window contains a status display. When a polygon is computed, the status display shows with which method the computation is done. When a user input with the mouse is required, the possible actions are described. Otherwise, the line states "Normal".



Normal.

Figure 3.1: A screen shot of the graphical user interface of our implementation.

3.1.2 Command Line Options

Our implementation accepts the following command-line options:

--noX specifies that the graphical user interface is not to be started. Instead, all input data has to be provided by command line options in order to generate a polygon. By default, the graphical user interface is started.

--random <*number of points*> allows the user to specify a number of random points to be generated. Thus, <*number of points*> has to be a positive integer. By default, no points are generated.

--seed <*seed*> sets the seed of the random number generator. This option is useful for generating different polygons on one set $\mathcal{S}$ of input data with the same algorithm. By default, no seed is set. (I.e., the polygon is generated with the default seed.)

--algo <*name of the algorithm*> is for specifying the algorithm with which to compute the polygons. Valid algorithm names are *bounce* (Bouncing Vertices), *xmono* (x-Monotone Polygon), *arrange* (Star Arrange), *quick* (Quick Star), *star* (Star Universe), *growth* (Steady Growth), *growthII* (Steady Growth II), *2opt* (2-Opt Moves), *space* (Space Partitioning), *reject* (Permute & Reject) and *search* (Incremental Construction & Backtracking). For this option there exists no default.

--format <*format*> specifies the format of the output data. Valid format specifiers are *standard* for the file format used by our implementation, *ipe* for writing an ipe file and *Voronoi* for the Voronoi file format. The default value of this option differs: If the user interface is started, the default is to print no output, whereas *standard* is the default for the non-interactive version ("--noX").

--output <*filename*> is used for specifying the name of an output file. By default, the output is written to "stdout"[4].

--input <*filename*> allows the specification of a file containing a set of input points.

Our implementation checks for the correctness of the command line options, but it is up to the user to specify meaningful parameters. I.e., if no algorithm is specified, our implementation will generate no polygon at all.

3.2 Data Types

The data type on which polygons, convex hulls and points are based is a simple array. We have two data types for arrays: The first data type allows to specify the exact size when creating the array, then the size is fixed[5]. We will call this a "static" array. The second data type allows the resizing of the array after its allocation. This will be called a "dynamic" array. Further basic data types used are heaps, stacks and AVL-trees, cf. Sedgewick [Sed83].

---

[4] "stdout" is the file handle for the standard output in C.

[5] Note that any array in C can be resized. However, the data types may not be accessed directly, but only by predefined access routines.

3.2.1 Points

We keep the points always in a lexicographically[6] sorted order. Note that this order also implies a comparison operation on points: We say that $p < q$ if p precedes q in our order. Thus, we face the problem of finding a data structure that allows both direct access to each point's coordinates and easy maintenance (i.e., insertion or deletion) of the points. To provide both, we use the following approach: For the manipulation of points, we keep the points in a sorted list. Obviously, a sorted list allows for easy deletion or insertion of points. Once the manipulation of the points has been finished, we copy the points into an array, which permits the direct access needed for our algorithms.

3.2.2 Polygons and Polylines

A polygon is defined by its vertex set and its edge set. Since we assume that two consecutive vertices form an edge, we only need to keep track of the order of the vertices for storing a polygon. We do not need to additionally store the vertices, since all algorithms use all points of the given point set $\mathcal{S}$ as vertices and we have already stored the points in a sorted array.

There are two ways in which we store a polygon. We will illustrate this with the polygon shown in Fig. 3.2. In the simple version, the indices of the vertices are stored in the array in the same order as they appear in the polygon, cf. Table 3.1. The first index is always 1, and the polygon is always represented in CCW orientation[7]. We can test whether the polygon is in CCW direction in constant time: Consider the first vertex v_1, and test whether the last vertex v_n lies left of the oriented line $\ell(v_1, v_2)$. The second method is to have a "linked list": For each vertex index, we store the index of the next vertex encountered when walking along the boundary of the polygon in CCW direction.

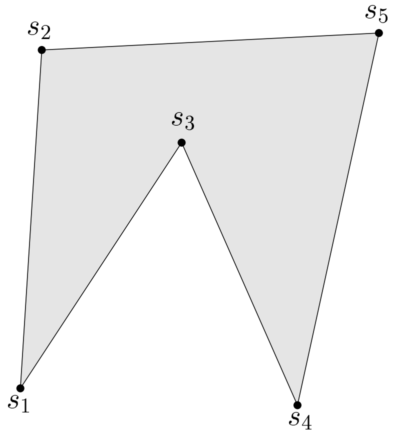


Figure 3.2: A simple polygon for illustrating the two storage methods used.

---

[6] I.e., sorted by x-coordinate, or sorted by y-coordinate if the x-coordinates are identical.

[7] Note that polygons may be computed in CW orientation nevertheless.

Array Index	1	2	3	4	5
Vertex Index	1	3	4	5	2

Table 3.1: A polygon represented by a list of vertex indices.

Array Index	1	2	3	4	5
Vertex Index	1	2	3	4	5
Index of Next Vertex	3	1	4	5	2

Table 3.2: A polygon represented by a linked list.

In our algorithms, both data structures are used. The simple list is the standard method which is also used as global data structure for polygons, i.e., the current polygon we save for drawing on the canvas is stored in a simple list, and all input/output operations require simple lists as parameters. On the other hand, we use the linked list concept, when the vertices are not encountered in the same order as they appear in the polygon (i.e., we do not compute the k-th vertex of the resulting polygon during phase k). E.g., Steady Growth needs the linked-list structure.

The same representations are available for polylines (polylines are our implementation of polygonal chains). Note that a polygon always contains n vertex indices, where n is the number of vertices, whereas a polyline may contain an arbitrary number (at most n) of vertex indices. Further, a polygon has a circular structure, i.e., after the vertex index at position n follows the one at position 1, whereas no such relation exists on polylines. However, we can use the same data type for storing linked polygons and for storing linked polylines.

3.2.3 Convex Hulls

Basically, convex hulls are polygons. However, whereas a polygon on a set of n points always contains all n vertices, this will not be the case for the convex hull except for degenerate cases. In general, the convex hull will contain k vertices, where $3 \leq k \leq n$. Therefore, we have to store the number of actual vertices the convex hull contains. Our data structure used for representing polygons does not allow this, whereas polylines do. For this reason, convex hulls are stored in polylines in our implementation. Note, however, that a convex hull adds circular structure to the polyline.

Except for Steady Growth, where we need to dynamically update the convex hull, the convex hulls used in the algorithms are static[8] hulls of the point sets and thus a representation by static arrays is used.

---

[8] I.e., they are not modified after their computation.

3.2.4 Lines

A line ℓ is defined by two (different) points p_1 and p_2. We store p_1, p_2 in sorted order, i.e., $p_1 < p_2$. In addition to the two points, we store its Hessian normal form: We compute the vector that is normal to $\ell(p_1, p_2)$ and that lies to the left of the oriented line through p_1 and p_2. Next, we normalize[9] this vector to unit length. Let a denote the x-coordinate of this vector, and let b denote the y-coordinate. Further, let c be the dot product of p_1 and $-\binom{a}{b}$. Now, every point p with coordinates x and y lying on ℓ satisfies the following equation:

$$a \cdot x + b \cdot y + c = 0. \tag{3.1}$$

Thus, if we evaluate the left side of Equation (3.1) for some point p then we get 0 if p lies on ℓ. Otherwise, we get the signed distance from p to ℓ.

3.3 File Formats

Our implementation allows us to import points and polygons from files and to output them to files. Due to the fact that the polygons are given as a list of point indices, saving a polygon naturally requires saving the points upon which the polygon is defined. Our file formats support commentary lines. (Comments must be started with the character "#".)

We start with describing the format in which points are stored. The first line contains the region in which the points lie. It is specified by minimum x, maximum x, minimum y and maximum y, where the values are separated by blanks. The next line contains the number of points that are contained in the file. Then each point is given on a separate line with x-coordinate and y-coordinate separated by a blank. It does not do any harm if a file contains more entries than the number of points specified; any excess entries are ignored. The contrary (less entries than specified) is not permitted, though. When writing an output file, the points are written in sorted order. However, input points need not be specified in sorted order. A valid file containing point data is given in Table 3.3.

In addition to files containing point data, our implementation also allows to store and load polygons. Since polygons are a list of point indices, the first part of a file containing a polygon is identical to a file containing points. Note, however, that the number of points and the actual number of point entries may not differ for a file containing a polygon. After the last point has been specified, the number of polygons follows in the next line. In the current implementation, this number is always 1 and ignored when reading the file. Nevertheless, the file must contain this number. In the next line the word "POLYGON" follows. (This is for the sake of an easier separation of the polygons, in case that a file contains more than one polygon.) Then, the point indices are given one per line. Naturally, a total of n indices is necessary. When writing the output file, index 1 is the first index to be written. (An input file may start with any index.) A polygon file is depicted in Table 3.4.

---

[9] We use a Euclidean norm, i.e., *length* $= \sqrt{x^2 + y^2}$.

```
################################################
#                                              #
#    Data file for random polygon generator    #
#                                              #
#    written by T. Auer - tom@cosy.sbg.ac.at   #
#                                              #
#          as part of my master thesis         #
#                                              #
################################################
0.000000 1.000000 0.000000 1.000000
5
0.120807 0.129406
0.226119 0.843143
0.443743 0.426983
0.735386 0.092030
0.916021 0.916660
```

Table 3.3: A sample file containing point data.

```
################################################
#                                              #
#    Data file for random polygon generator    #
#                                              #
#    written by T. Auer - tom@cosy.sbg.ac.at   #
#                                              #
#          as part of my master thesis         #
#                                              #
################################################
0.000000 1.000000 0.000000 1.000000
5
0.120807 0.129406
0.226119 0.843143
0.443743 0.426983
0.735386 0.092030
0.916021 0.916660
# number of polygons
1
POLYGON
1
4
5
2
3
```

Table 3.4: A sample polygon file.

Further, our implementation provides two additional output formats. It is able to write the current points and polygons in a format which is understood by the drawing editor Ipe[10] implemented by Schwarzkopf, cf. Schwarzkopf [Sch94]. It is possible to save points even when no polygon has been generated. The other format supported is the file format required by Held's Voronoi program, cf. Held [Hel93]. For the latter an output file is written only in case that a polygon has been computed. It is not possible to save the points without a corresponding polygon[11].

3.4 Basic Algorithms

3.4.1 Convex Hull

For calculating the convex hull $\mathcal{CH}(\mathcal{S})$ of a set $\mathcal{S}$ of points, we use Graham's algorithm, cf. O'Rourke [O'R94]. The algorithm works as follows: First, $s_2, \dots, s_n$ are arranged in polar coordinates (α, r) with respect to s_1. Second, the points $s_2, \dots, s_n$ are sorted lexicographically. Of course, no "real" polar coordinates are computed, but comparisons are carried out by means of determinants. Also, note that this ordering guarantees that the last point in our ordering is on the convex hull[12].

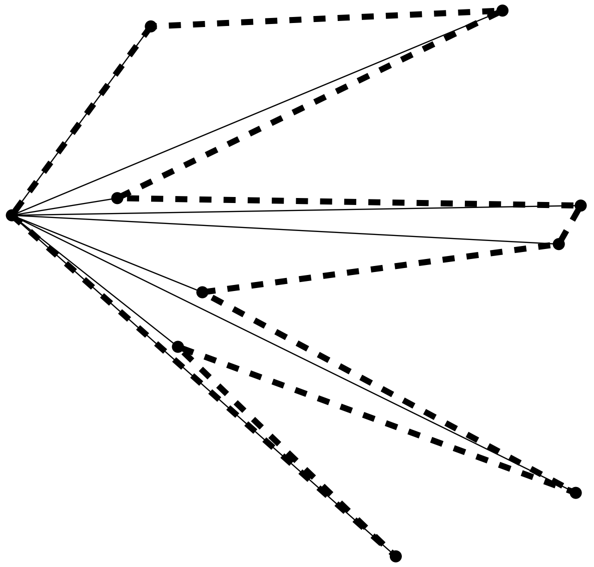


Figure 3.3: The points are sorted in angular order around the first point.

---

[10] We used this feature for the generation of the figures for the traces of our algorithms: The output of our program was converted into input suitable for LaTeX 2_ε by means of Ipe.

[11] This is motivated by the fact that the Voronoi program requires curvi-linear data as input.

[12] For collinear points, only the two points with the maximum distance may be vertices of the hull.

Next, we initialize our hull with the last point of the lexicographical sorting and the first point, which necessarily form an edge of the convex hull. We proceed with the points in the order in which they are sorted and do the following: Consider the last edge (v_{k-1}, v_k) of the convex hull (i.e., the last two points in the array), and test on which side of the oriented edge (v_{k-1}, v_k) the current point s lies. If the point lies to the left, the angle formed by v_{k-1}, v_k and s is convex, and therefore s is added as v_{k+1} to the hull. This is illustrated in Fig. 3.4a. Otherwise, if the point lies on the line or to the right of the line, the angle is not convex. In this case, we have to remove v_k, and do the same check with vertices v_{k-1} and v_{k-2}, as illustrated in Fig. 3.4b. Since we initialized the hull with the last point (in our order) and with the first point of the point set, we will always have at least one edge of the convex hull. After processing all the points, we have to delete the last point, because it has been added to the hull twice.

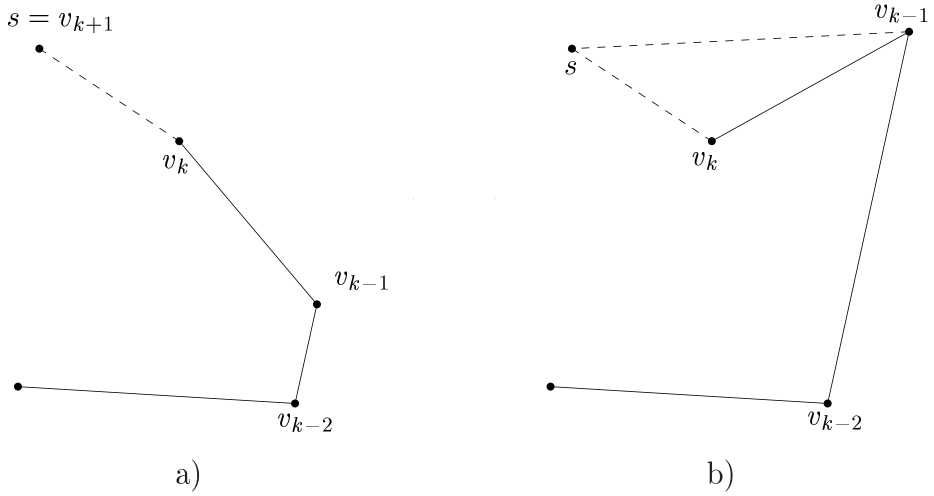


Figure 3.4: Processing of a point by Graham's scan algorithm.

3.4.2 Computing Intersections

Testing Line Segments for Intersection

We will only be interested in detecting intersections of segments defined by points of $\mathcal{S}$. Also, two line segments sharing a vertex are not regarded as intersecting. For testing whether two line segments $\overline{p_1p_2}$ and $\overline{q_1q_2}$ intersect, we use the following property: The line segments do only intersect if and only if the supporting line of one line segment splits the endpoints of the other segment, and vice versa, cf. O'Rourke [O'R94]. Thus, the endpoints lie on opposite sides of the supporting line. This is illustrated in Fig. 3.5, where the segments in Fig. 3.5a do not intersect, whereas the segments in Fig. 3.5b intersect.

Based on this property, we compute whether two line segments intersect as follows. First, we want to guarantee that the result for two line segments is independent of the order in which the segments or their endpoints are given. To ensure this, we swap the endpoints and the segments in a way that the following properties hold:

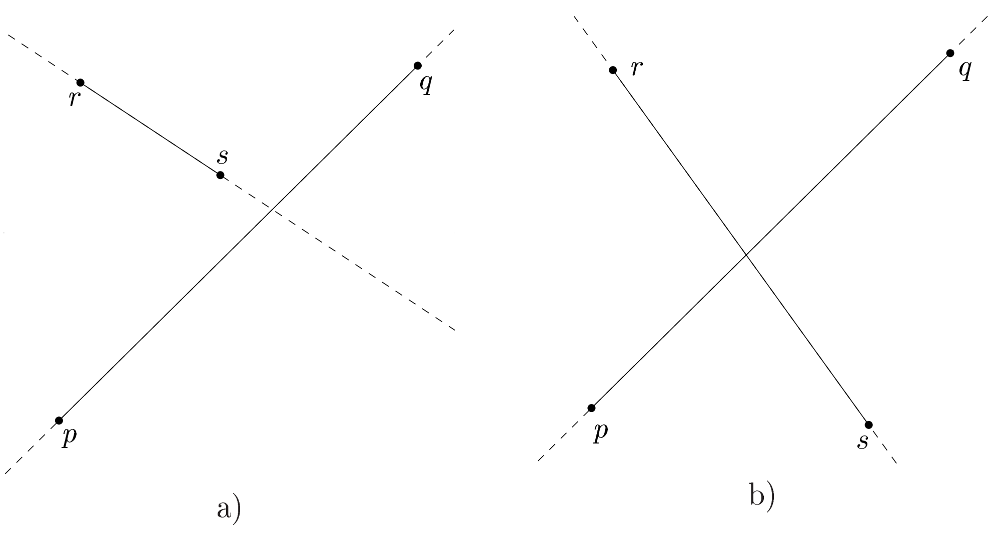


Figure 3.5: Line segments and their supporting lines.

- $p_1 < p_2$,
- $q_1 < q_2$ and
- $p_1 < q_1$ or $(p_1 = q_1) \wedge (p_2 < q_2)$[13].

Note that this order can be inferred from the indices of the points. Before actually computing whether the segments intersect, we test whether their bounding boxes overlap. If they do not overlap then the segments cannot intersect. Determining the bounding box is computationally inexpensive due to the ordering stated above.

Now that we have this particular ordering, we determine where each point lies with respect to the supporting line ℓ of the other segment; the point may lie left, right or on ℓ. In the following, we denote the supporting line of $\overline{p_1p_2}$ by ℓ_1 and the supporting line of $\overline{q_1q_2}$ by ℓ_2. First, we test whether either p_1 and p_2 lie on the same side of ℓ_2 or whether q_1 and q_2 lie on the same side of ℓ_1. In this case, the line segments do not intersect. Next we test whether p_1 and p_2 lie on opposite sides of ℓ_2 and whether the same is true for q_1 and q_2 with respect to ℓ_1. In this case, the line segments intersect. A special case we have to deal with is that all four points are collinear: All we have to check is whether $q_1 < p_2$, in which case the segments intersect. This test is sufficient, as we already have ensured that $p_1 < p_2$, $q_1 < q_2$ and $p_1 \leq q_1$, cf. Fig. 3.6. Having handled the collinear case, the only remaining position of the line segments is that at least one endpoint lies on the supporting line of the other segment. In this case, we test whether the segments share an endpoint, otherwise we have intersection, cf. Fig. 3.7.

Computing the Intersection Point of two Lines

When intersecting two lines, we want to know

---

[13] Note that in the case that the second points are also identical, the ordering of the segments is of no importance at all, since both segments are identical.

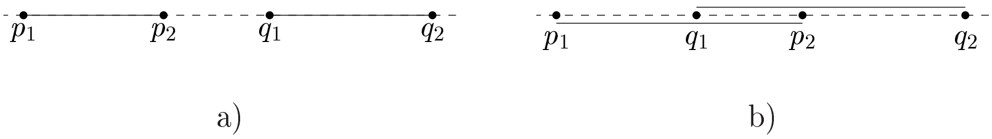


Figure 3.6: Segment intersection for four collinear points.

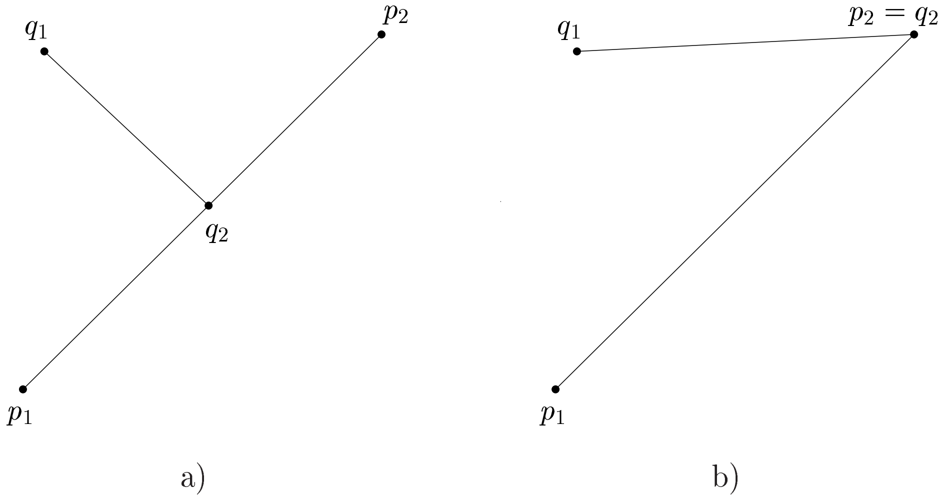


Figure 3.7: Line segments with three collinear points.

a) whether the two lines intersect at all (i.e., a logic value), and

b) if the lines intersect, we want to know the coordinates of their intersection point[14].

For calculating the intersection point of two lines $\ell_1 = \ell(p_1, p_2)$ and $\ell_2 = \ell(q_1, q_2)$, we proceed as follows. First, we want to ensure that testing two lines always leads to the same result, no matter in which order the lines are specified and therefore we enforce the same particular ordering as when testing line segments for intersection. Next, we test whether the lines coincide. In this case, we return FALSE as result, because no single intersection point can be found. Otherwise, we test whether any of the points defining the two lines are identical. If this is the case, we have found a point that lies on both lines, and we can return this point as intersection point. The last of the simple cases is that a point defining ℓ_1 lies on ℓ_2, or vice versa. If we find such a point, we return this point as intersection point. This additional test is done since it yields a numerically robust solution.

If none of the tests given above yields a point of intersection, we need to compute the intersection point. First, we test whether the lines are almost parallel: we compute the vector $p_2 - p_1$, normalize its length and compute the product of this vector with the normalized normal vector of ℓ_2. If this product is less than a predefined constant, we consider the lines to be parallel and return FALSE as result.

---

[14] Note that our goal is to compute an intersection point. Thus, for all cases, where no single intersection point exists, we will return FALSE.

Otherwise, we compute a point of ℓ_1 that lies on ℓ_2, i.e., we insert a general point $p = p_1 + \lambda \cdot (p_2 - p_1)$ on ℓ_1 in Equation (3.1) of ℓ_2. Thus, we get a point that certainly lies on ℓ_1. (Actually, we get a value for λ.) In the absence of numerical errors, p would also lie on ℓ_2. However, if p does not lie on ℓ_2 (numerically), then we first set $\lambda_1 = \lambda_2 = \lambda$ and then shrink or grow λ_2 with a factor of 1.1 until the point determined by λ_2 lies on the opposite side of ℓ_2 as the point determined by λ_1 does. Now that we have two points which lie on opposite sites of ℓ_2, we bisect the parameter interval $[\lambda_1, \lambda_2]$ until we eventually get a point that lies on ℓ_2. Note that this procedure handles intersection between lines that are nearly parallel in a robust way.

3.5 Details of the Algorithms

3.5.1 Bouncing Vertices

Our implementation of Bouncing Vertices is straightforward and differs only in minor details from the description of the algorithm in Section 2.1. Our implementation offers two methods for moving the vertices:

a) the vertices are moved by assigning them new random values, and

b) the vertices are moved by generating a random vector which is added to the current vector[15] associated with the vertex. Note that the random vectors are not uniformly distributed in the plane.

The method of choice is selected by the preprocessor constant *BOUNCE_POINTS*, by default the implementation uses method a).

Note that the points of $\mathcal{S}$ are always stored in sorted order. Naturally, the movement of the vertices destroys this order. Thus, we have to sort our points for intersection tests, because the relative order of points can no longer be derived from the vertex indices. Further, after the generation of the polygon, we have to resort the array containing the points and we need to update the vertex indices of the polygon computed, because obviously the sorting of the points modifies the indices. Having done this, we finally have to shift our polygon in order to ensure that the first index in the polygon is 1. Further, since we store the points in both a list and an array, we have to delete the old points and to generate a list containing the new points. The sorting process requires $\mathcal{O}(n \log n)$ time, whereas the re-indexing and the shifting of the polygon can be done in linear time. Finally, the generation of the list is also done in linear time. Thus, these steps do not influence the time complexity of the algorithm. Obviously, these steps can be done within $\mathcal{O}(n)$ space, and thus the space requirement of the algorithm remains at $\mathcal{O}(n)$.

3.5.2 x-Monotone Polygons

Our implementation of x-Monotone Polygons is straightforward and differs only in minor details from the description in Section 2.2. We store both the top and the

---

[15] Remember that each vertex corresponds to the vector from the origin to its location.

bottom chains in separate polylines. When we have computed both chains, we combine (copy) them into a polygon. In order to have the resulting polygon in CCW order, we first copy the bottom chain and then the top chain. Since we have both chains with the first point being the one with the minimum x-coordinate, we have to revert the order of the top chain.

For the computation of the sets of points that are above-visible and that are below-visible (Algorithm MakeTop in Section 2.2), we have implemented only one algorithm that either calculates the above-visible or the below-visible set, since the only difference is whether we check if the point lies above some reference line or whether we check if a point lies below the line. Further, in Algorithm generateTop and in Algorithm generateBottom, the variable *lastSib* is used to keep track of the last node encountered. Due to the fact that this variable is modified in each call and we need this modification to be visible in the calling process, we extended the tree structure such that each tree provides a variable in which we store the last node we processed.

In order to be able to compute x-monotone point sets, we store the values for $\mathcal{V}^B$ and $\mathcal{V}^T$ in arrays of *unsigned long* variables. Nevertheless we encountered an overflow[16] for a set with 1,000 points.

One problem occurring is that the random number generator may not produce sufficiently large random numbers. In this case, we spread the random numbers evenly over the entire range of numbers required. E.g., if we have 10 polygons but are only able to produce random numbers from 1 to 5, then we generate the numbers 2, 4, 6, 8 and 10.

3.5.3 Enumeration of Star-Shaped Polygons

For both Star Universe and Star Arrange we resort the vertices of the polygon each time we enter a new kernel in order to simplify the cumbersome handling of collinear points. This adds a factor of $\log n$ to the time complexities stated in Section 2.3.

Star Arrange

For actually storing the arrangement, we use the quad-edge data structure by Guibas and Stolfi [GS85]. Our implementation of the quad-edge data structure is based upon an implementation by Truong[17]. Further, for the sake of an "easier" implementation, we use a sweep-line algorithm which requires $\mathcal{O}(n^4 \log n)$ for the computation of the arrangement instead of the incremental version (which requires $\mathcal{O}(n^4)$ time): We compute all the intersection points within the convex hull and sort them lexicographically. With each line, we store the last quad-edge structure we generated. When processing an intersection point, we first complete the data structures of the segments that lie to the left of the point (i.e., we complete the data associated with each incoming line), and then we generate the data for the segments that lie to the

---

[16] Since C does not provide any exception handling, the code needs to check every addition for overflow.

[17] M. Held obtained this implementation from J. Snoeyink some years ago.

left of the current intersection point. Finally, we have to check whether there are multiple intersections at one intersection point (i.e., we have to check whether there are collinear points).

Star Universe

We store the polygons in an AVL-tree instead of computing a unique point in the kernel of each polygon. Thus, we have a time complexity of $\mathcal{O}(n^5 \log n)$ instead of $\mathcal{O}(n^5)$, and the space requirement is $\mathcal{O}(n^2 + k \cdot n)$. Next, we consider each intersection point only once during the algorithm, because when considering an intersection point, all the (up to four) polygons associated with it are considered. Further, we have to check whether we have points that are collinear with the current line. If there exist several such points, we have to consider all possible relative orders among these points. A heap is used for storing the intersection points.

3.5.4 Quick Star

Instead of triangulating the convex hull, we use a rejection method in order to generate a point within $\mathcal{CH}(\mathcal{S})$ with uniform distribution: Select a point at random within the bounding box of $\mathcal{CH}(\mathcal{S})$, until a point which actually lies within $\mathcal{CH}(\mathcal{S})$ is found. Clearly, the efficiency of this method depends on the ratio of the area of the convex hull to the area of the bounding box: If the convex hull covers a significant area of the unit square (which is to be expected of practical point sets), it takes only a few steps to compute a suitable point.

Further, we always sort points that have the same polar angle with respect to the random point p (which belongs to the polygon's kernel) by their (increasing) distance from p. Thus, our implementation may miss some star-shaped polygons on sets with multiple collinear points.

3.5.5 Steady Growth

For Steady Growth, the following changes from the description in Chapter 2 were made: First, we do not use the linear algorithm by Joe and Simpson in order to determine all line segments that are visible from point s_k. Instead, the following approach was implemented: We sort the start- and endpoints of all edges that are between the left supporting vertex v_l and the right supporting vertex v_r in angular order around s_k. Clearly, we can apply backface culling (cf. Laszlo [Las96] p. 146): We only consider edges that have s_k to their right. We keep our edges on a heap; they are ordered according to their distances from s_k. Thus, an edge that is on top of the heap is visible from s_k. We process each point s of the sorted points as follows:

- First, we check whether we have an edge in the heap that has s as its endpoint. If such an edge exists, it is visible if it always has been the top element of the heap, in which case we save the edge. Independent of the visibility of the edge, we remove it from the heap.

- Second, we check whether the edge starting at s has s_k on its right side. If yes, we add the edge to the heap.

When maintaining the heap (this implies comparing two edges), we mark all edges that lie "behind"[18] some other edge as invisible. Clearly, an edge that is always on top of the heap lies in front of all other edges, whereas any edge that is not always on top of the heap must have been marked during the updates.

The second major difference from the description in Chapter 2 is the way in which we determine a suitable point s_k. Actually, we have two different versions:

Steady Growth, our original implementation selects one point s_k at random and marks that point as used in the current phase. If we find any point s lying inside the convex hull, we choose another (unmarked) point and test again. For each point we select in phase k, we have to check all the other points that are not already vertices of the polygon. We also have to compute the supporting vertices for each such point. Thus, in the worst case, we will try each point of $\mathcal{S}_k$, and test with all the remaining points for containment. Clearly, this leads to a time complexity of $\mathcal{O}(n^2)$ per phase, and a total time complexity of $\mathcal{O}(n^3)$.

Steady Growth II, the faster version, uses the selection method described in Section 2.4: When we encounter a point s lying inside the convex hull, we use this point as a new candidate for s_k. However, for the first computation of the supporting vertices, we need $\mathcal{O}(n)$ time instead of $\mathcal{O}(\log n)$ time. $\mathcal{O}(n)$ efforts are required because we store the convex hull as a linked list, since this allows an update operation in constant time. The update of the bridges for a new candidate is done in linear time for all candidates, as described in Section 2.4.

Finally, we describe how we incrementally update the convex hull. We initialize the convex hull with the first two vertices[19]. In Steady Growth, we have to compute the supporting vertices in each phase: We scan along $\partial\mathcal{CH}(\mathcal{P}_k)$, and set the left (respectively right) bridge to the current point until the current point does not lie left of the line $\ell(s_k, v_l)$ (respectively not right of $\ell(s_k, v_r)$), cf. O'Rourke [O'R94]. Clearly, this takes as many steps as there are vertices on the boundary of the convex hull. As mentioned above, the corresponding update step for Steady Growth II is carried out as described in Chapter 2.

3.5.6 Space Partitioning

Our implementation adheres mainly to the description given in Section 2.5 with only minor differences. First, our recursion terminates when we encounter sets with either two or three points, whereas the algorithm described in Chapter 2 terminates

---

[18] I.e., they lie further from s_k than some oder edge.

[19] Note that the convex hull consists only of two line segments which coincide. (However, this has no negative impact on our algorithm.)

only on sets with two points. This can be done because only one chain exists on sets of three points where the first and last points of the chain are fixed.

The partitioning of a set $\mathcal{S}'$ requires that we find a line ℓ through a randomly selected point s' such that ℓ separates s'_f and s'_l. We do this by calculating a random point p on the line $\ell(s'_f, s'_l)$ and then we use $\ell(p, s')$. Further, we require that the distance from the random point and s'_f, s'_l is at least a predefined constant of the length of $\ell(s'_f, s'_l)$. Note that the additional condition that neither s'_f nor s'_l lies on ℓ simplifies the handling of several collinear points. Thus, we need to find a point s that is not collinear with $\ell(s'_f, s'_l)$. If we cannot find such a point, all points are collinear. In that case, all that is left is to sort the points. Further, if we select ℓ and find a point that lies on ℓ, we test whether the current selection of s' or the new point have the greater distance from the base line $\ell(s'_f, s'_l)$. If the new point is the one with the greater distance, we add s' to the set of points lying left of ℓ and take the new point as candidate for s'.

The same has to be done when selecting the two random points for separating $\mathcal{S}$: If there are several collinear points, we will always choose as s_f and s_l the two points with the greatest distance. Further, points that lie on a line ℓ used for partitioning are always added to the set of points on the left of ℓ.

Since we will not encounter the vertices in order, we use a linked list for storing the polygon. Thus, a point can be added to the polygon by "linking" it with is predecessor and its successor.

3.5.7 Permute & Reject

The implementation of Permute & Reject differs in only one aspect from the description in Section 2.6. Instead of a linear test for simplicity, we use a straightforward quadratic approach, i.e., we test every edge e with every other edge for intersection.

Whereas we normally generate our polygons in CCW order, this is not necessary for Permute & Reject. We generate a random polygon and test it for simplicity. If it is simple, we revert the order of its vertices when necessary (i.e., when the polygon is in CW order). Clearly, this reversal can be done in linear time.

3.5.8 2-Opt Moves

For the implementation of algorithm 2-Opt Moves, we keep each intersection in three lists: We have a global list of all intersections from which we randomly select an intersection. Further, for each actual edge of the polygon we keep track of its intersections, thus we can remove all the intersections associated with an edge without searching for them.

A special case we have to treat is when we encounter the situation that one edge completely lies within another edge. In this case, we have to consider the edges that occur before the actually overlapping edges in order to successfully remove the intersection. We have to distinguish whether the two edges share an endpoint or whether they do not. Both cases are depicted in Fig. 3.8. In the polygon depicted in Fig. 3.8a which is defined by $(s_1, s_6, s_7, s_4, s_5, s_3, s_2)$, edge (s_3, s_5) lies within edge (s_1, s_6). Thus, we have to replace the edges (s_1, s_6), (s_3, s_5), (s_2, s_3) and (s_4, s_5) with the edges (s_1, s_3), (s_3, s_5), (s_5, s_6) and (s_2, s_4). In the polygon in Fig. 3.8b which is given as $(s_1, s_5, s_4, s_6, s_3, s_2)$, the edges (s_1, s_5) and (s_4, s_5) overlap and additionally share an endpoint. Thus, we have to replace the edges (s_1, s_5), (s_5, s_4) and (s_4, s_6) with the new edges (s_1, s_4), (s_4, s_5) and (s_5, s_6). Further, when removing an intersection, the direction of one of two the chains connecting the two intersecting edges has to be reverted.

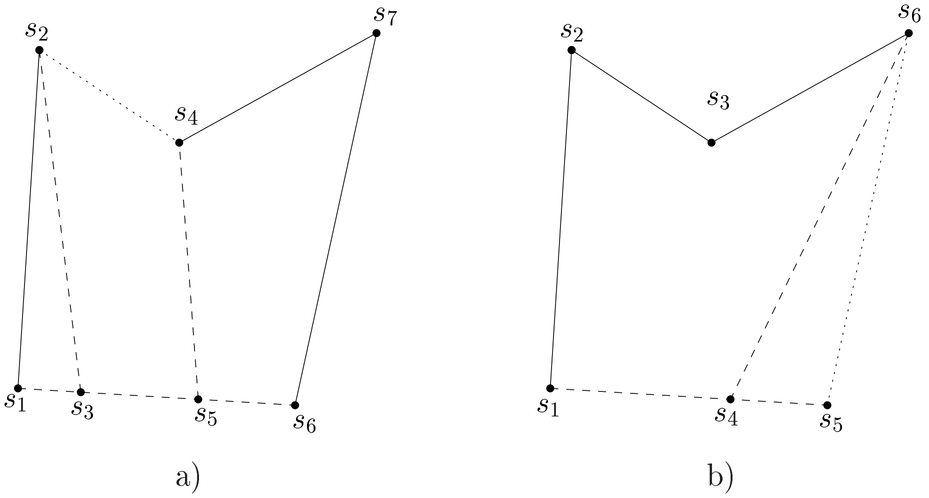


Figure 3.8: Two special cases for 2-Opt Moves.

3.5.9 Incremental Construction & Backtracking

Our implementation of Incremental Construction & Backtracking is straightforward: We recursively add points to the chain, until one of the conditions is violated. In this case, backtracking has to be applied. Obviously, we keep track of the points already tried unsuccessfully. Thus, every point is tried exactly once in each phase until backtracking takes place. Further, we keep track of all edges marked during a phase, because all marks have to be deleted before backtracking is started. For each point, we keep track of its usable edges and its mandatory edges in order to make checking of the conditions faster. In addition, we precompute all the edge intersections in order to speed-up the backtracking. (If a lot of backtracking is done then the same edges will be tested for intersection again and again.)

Chapter 4

Experimental Results

We ran three different series of experiments, which are reported in the following sections.

1. We recorded the CPU-consumption of our algorithms.
2. We obtained experimental bounds on the numbers of star-shaped and simple polygons in terms of the cardinality of the point set.
3. We evaluated the number of polygons generated by our algorithms in order to assess the quality and practical applicability of these heuristics.

In addition, we determined experimental bounds for parameters (such as the number of random points one has to generate for algorithm Bouncing Vertices, cf. Section 2.1) which have a great influence on the actual running time of the algorithms.

4.1 CPU-Time Consumption

We measured the CPU-time consumption of each algorithm when applied to random point sets (within the unit square) of the following cardinalities: 10, 25, 50, 100, 200, 300, 400 and 500. For each of these cardinalities we generated three independent sets. Our algorithms had to compute 50 polygons on each of these sets. The mean elapsed CPU-times (in milliseconds) were plotted using a logarithmic scale ($\log_2 t$).

As expected, Star Universe and Star Arrange are only feasible for input sets with a small cardinality: Our attempts to run Star Universe on 100 points had to be aborted due to lack of main memory[1]. (Star Arrange did not even yield any results on sets with 50 points.) The time difference between these two algorithms is only marginal. Quick Star, however, seems well suited for larger point sets: computing a star-shaped polygon on 500 points takes roughly 72 milliseconds. The cpu-time consumptions for the generation of star-shaped polygons are depicted in Fig. 4.1.

Two of the algorithms for the generation of simple polygons are not applicable to anything but extremely small point sets, cf. Fig. 4.2: In more than three weeks of

---

[1] 192MB of main memory and adequate swap space did not suffice.

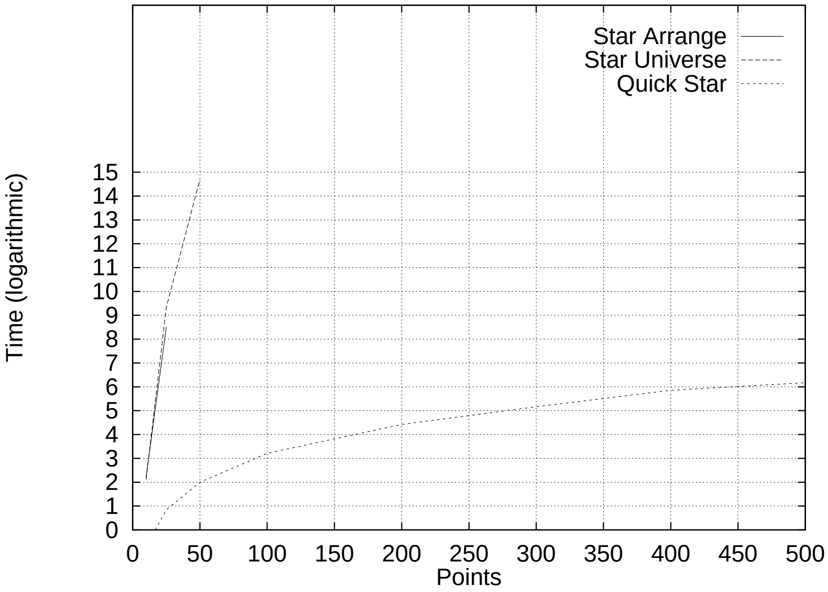


Figure 4.1: CPU-consumption of the algorithms for the generation of star-shaped polygons.

running time we were not able to generate results for 25 points when using Permute & Reject or Incremental Construction & Backtracking. Clearly, those two algorithms are not suited for practical purposes. Note that using an $\mathcal{O}(n \log n)$ test for polygonal simplicity instead of our brute-force $\mathcal{O}(n^2)$ test would not improve the applicability of Permute & Reject at all.

Among the remaining four methods, Space Partitioning is significantly faster than the two other algorithms. Roughly, Space Partitioning takes about 62 milliseconds to compute a simple polygon on 500 points, whereas Steady Growth consumes about 23 seconds and 2-Opt Moves takes about 15 seconds. Fairly interesting is the time behavior of Steady Growth II: For 500 points, it is approximately eight times faster than Steady Growth. It takes about 2.7 seconds for generating a simple polygon on 500 points. Thus, from a performance point of view, Space Partitioning is the candidate of choice. Note that Quick Star and Space Partitioning consume about the same amount of CPU time. These results are depicted in Fig. 4.2.

4.2 Number of Polygons

By using a modified version of Incremental Construction & Backtracking, we determined the number of simple polygons on groups of ten sets with 10 respectively 15 random points. For star-shaped polygons, we enumerated all polygons for groups of ten sets with 10, 15, 20, 25 and 50 points each by means of Star Universe. All these numbers are listed in Table 4.1. In addition, we listed the total number[2] of polygons existing on a set of size n. For 10 points, the number of simple polygons

---

[2] Recall that the total number of different polygons on a set with n vertices is $\frac{(n-1)!}{2}$.

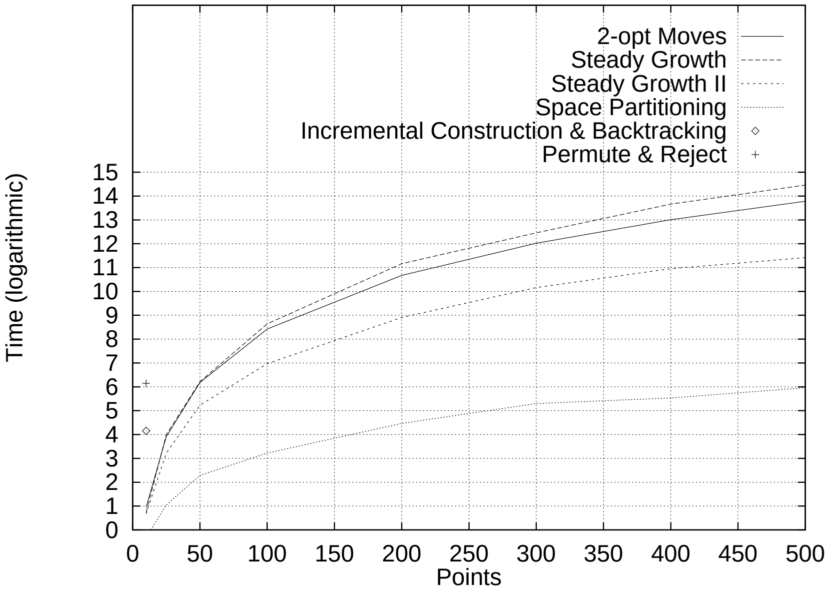


Figure 4.2: CPU-consumption of the algorithms for the generation of simple polygons.

was in the range from 146 to 852, whereas for 15 points we found an experimental minimum of 58,768 simple polygons and a maximum of 291,232 simple polygons. For star-shaped polygons, the variance is smaller than for simple polygons. On 10 points there exist 44 to 103 polygons, for 15 points the numbers lie between 266 and 516, for 20 points between 995 and 1,816 and for 25 points between 2,340 and 4,120. For 50 points, the number of star-shaped polygons is still smaller than the number of simple polygons (when taking the mean over all sets): The number of star-shaped polygons ranges from 59,017 to 82,478. In our tests, all polygons which describe the same geometric figure were counted exactly once. (All polygons were generated in CCW order, with the point with minimum x-coordinate as the first point.) Note that test runs for counting all simple polygons on 20 points crashed due to lack of disk space after generating more than three million different polygons.

The gigantic number of simple polygons on comparatively small sets of points also constitutes a practical problem when implementing any algorithm which is based on an enumeration of all polygons: Integer overflows are very likely to occur[3], unless one switches to variable-length integer arithmetic. Also, the conventional random number generators (as contained in the standard libraries) can no longer be applied without modifications because they do not generate sufficiently large random numbers. (Section Section 3.5 reports how we produced random numbers that were large enough.)

---

[3] We experienced integer overflows when running our implementation of the algorithm by Zhu et al. [ZSSM96] for 1,000 points.

$\lvert\mathcal{S}_i\rvert$	Simple		Star-Shaped				
	10	15	10	15	20	25	50
1	351	195,554	67	320	1,061	2,666	59,017
2	329	58,768	51	266	1,015	2,340	77,685
3	164	65,338	44	287	995	3,318	63,741
4	776	291,232	103	516	1,816	4,120	82,478
5	146	149,701	44	358	1,170	2,827	66,708
6	321	269,022	57	435	1,450	3,696	71,943
7	852	199,266	76	418	1,447	3,906	70,147
8	346	150,423	50	357	1,136	3,203	67,213
9	380	281,324	56	382	1,293	3,680	64,466
10	599	205,536	87	392	1,353	2,916	65,004
$\frac{(n-1)!}{2}$	181,440	$4 \cdot 10^{10}$	181,440	$4 \cdot 10^{10}$	$6 \cdot 10^{16}$	$3 \cdot 10^{23}$	$3 \cdot 10^{62}$

Table 4.1: Results for the numbers of simple and star-shaped polygons.

4.3 Experimental Bounds

The running times of the algorithms presented in Chapter 2 depend on several parameters. When describing the complexity of an algorithm, we either assumed a worst-case scenario or modeled the dependence on such parameters by introducing specific complexity terms. Whereas this yields a theoretical bound for the algorithms in question, their practical running time depends mainly on these parameters. Thus, we ran tests to investigate these parameters. We used the same series of tests we used for timing the algorithms.

The running time of the first algorithm in our thesis, Bouncing Vertices, is heavily influenced by the number of points we need to consider until a suitable one is encountered. We generated 10 polygons on each set and for the generation of each polygon, we ran five phases of Bouncing Vertices. In Table 4.2 we present the results (i.e., the average number of random vectors generated per point moved) first for the method of moving vertices, where we obtained numbers for different maximum lengths ("radii") of the vectors. Clearly, the numbers increase when the number of points goes up or when the radius is larger. The last row of the table gives the numbers when we replace each vertex with a randomly created new one. A question yet to be answered is whether more phases of the algorithm are required if only small perturbations per point are allowed. (In order to answer this question we would need a measure for the quality of a random polygon, though.)

For the generating star-shaped polygons by means of Quick Star, we determined how many points have to be generated until a point within the convex hull is encountered. In the table, we list the average (mean) for the generation of one point within the convex hull. Note that the number of points generated decreases when the number of points increases. This is due to the fact that the convex hull will cover a larger area when the number of points increases.[4]

---

[4] Recall that we use a uniform distribution within the unit square.

	Number of Points							
Radius	10	25	50	100	200	300	400	500
0.1	1.25	1.44	1.73	2.65	4.67	5.76	7.26	8.37
0.2	1.60	1.95	2.69	4.56	7.87	9.90	12.75	14.67
0.3	1.95	2.57	3.74	6.21	10.56	13.92	17.61	20.65
0.4	2.31	3.16	4.65	7.61	13.69	17.57	22.01	26.28
0.5	2.63	3.76	5.77	9.24	16.55	21.48	26.79	31.56
0.6	3.04	4.49	7.02	11.18	19.55	24.83	31.65	37.09
0.7	3.82	5.54	7.89	12.92	22.18	28.98	36.4	43.47
0.8	4.16	6.09	8.97	14.36	25.19	33.10	40.93	49.13
0.9	4.42	6.76	10.13	16.14	28.28	37.08	46.47	54.92
1.0	5.48	7.55	11.49	17.79	31.18	41.39	51.83	61.55
Vertices	7.35	16.56	26.95	55.23	126.36	175.31	209.96	265.99

Table 4.2: Average number of points generated by **Bouncing Vertices**.

Number of Points							
10	25	50	100	200	300	400	500
2.27	1.63	1.27	1.18	1.08	1.03	1.03	1.03

Table 4.3: Average number of points generated by **Quick Star**.

For **Steady Growth** we actually implemented two versions which differ in the way a suitable point s_k to be added to the polygon is selected. Thus, we ran tests to determine how many points have to be considered for both versions. In Table 4.4, we list the number of points that needed to be considered until a suitable one (that could be added to the polygon) was found. Clearly, the number of points to be considered increases when the total number of points increases. As expected, this number is drastically smaller and grows substantially slower for **Steady Growth II** than for **Steady Growth**.

	Number of Points							
Algorithm	10	25	50	100	200	300	400	500
Steady Growth	1.17	2.60	4.75	9.10	17.78	25.51	34.67	43.19
Steady Growth II	1.00	1.39	1.67	1.96	2.29	2.39	2.50	2.57

Table 4.4: Average number of points considered by **Steady Growth**.

For **2-Opt Moves**, we experimentally determined the number of 2-opt moves that have to be applied in order to obtain a simple polygon. These numbers are given in Table 4.5. In addition, we listed the numbers for n^3, the theoretical bound on the number of 2-opt moves. As can be seen in the table, the average actual number of 2-opt moves necessary is very small compared to the theoretical bound; roughly, it seems to grow about as fast as $\frac{n \log^2 n}{2}$.

	Number of Points							
	10	25	50	100	200	300	400	500
2-opt moves	4.83	25.18	74.58	195.47	439.13	828.37	1,183.18	1,565.86
n^3	1,000	15,625	125,000	10^6	$8 \cdot 10^6$	$2.7 \cdot 10^8$	$6.4 \cdot 10^8$	$1.25 \cdot 10^9$

Table 4.5: Average number of 2-opt moves.

Next, we ran tests to determine the number of polygons that have to be generated by Permute & Reject until a simple one is encountered. For 10 points, approximately 400 polygons have to be generated. However, this number increases so rapidly that we were not able to generate any results for sets with 25 points[5] The same problem occurred when trying to quantify the amount of backtracking Incremental Construction & Backtracking does. For the generation of a polygon on 10 points, backtracking has to be applied approximately 1.5 times per polygon generated. For 25 points, our tests did not yield any results.

4.4 Quality Assessment

For each algorithm, we started with experimentally determining the ratio of the number of polygons generated and the total number of possible simple polygons that could be generated. Let m be the number of different polygons we generated. Further, let t denote the number of polygons generated and let k denote the number of simple polygons that exist on the set upon which we computed the polygons. Clearly, if the number of tests is smaller than the number of simple polygons, the optimal result would be the generation of t different polygons. Otherwise (i.e., the number of simple polygons is smaller than the test size), the optimum would be the generation of k simple polygons. Thus, in the figures we depicted the value of $\frac{m}{\min(t,k)}$ which describes the ratio of different polygons generated and the number of different polygons that could be generated by an optimal algorithm.

For star-shaped polygons we have results for sets with 20 and 25 points. When generating 100,000 star-shaped polygons on 20 points with Quick Star, the mean percentage of polygons hit at least once was 91.439 with a minimum of 89.250 and a maximum of 94.679. When generating 10,000 polygons on 20 points we got 65.361 as mean, 59.141 as minimum and 69.241 as maximum, cf. Fig. 4.3. For 25 points and 100,000 polygons generated, we got a mean of 84.045, a minimum of 80.461 and a maximum of 87.634, whereas the corresponding numbers for 10,000 polygons are 51.769, 44.830, and 55.085. The results for 25 points are depicted in Fig. 4.4. Since Quick Star is capable of producing all possible star-shaped polygons it does not come as a big surprise that the hit rate goes up as the number of polygons generated is increased.

In the case of simple polygons, we tested 10 groups of sets with 10, 15 and 100 random points each. For sets with 10 or 15 points, we have results for 10,000 and

---

[5] Our implementation was not able to compute 50 simple polygons on 25 points despite more than three weeks of running time.

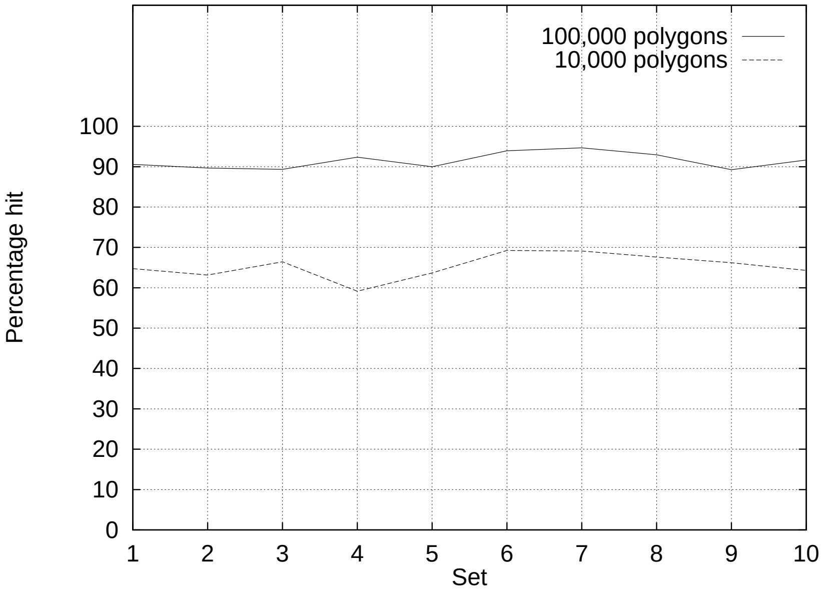


Figure 4.3: Results for star-shaped polygons on 20 points.

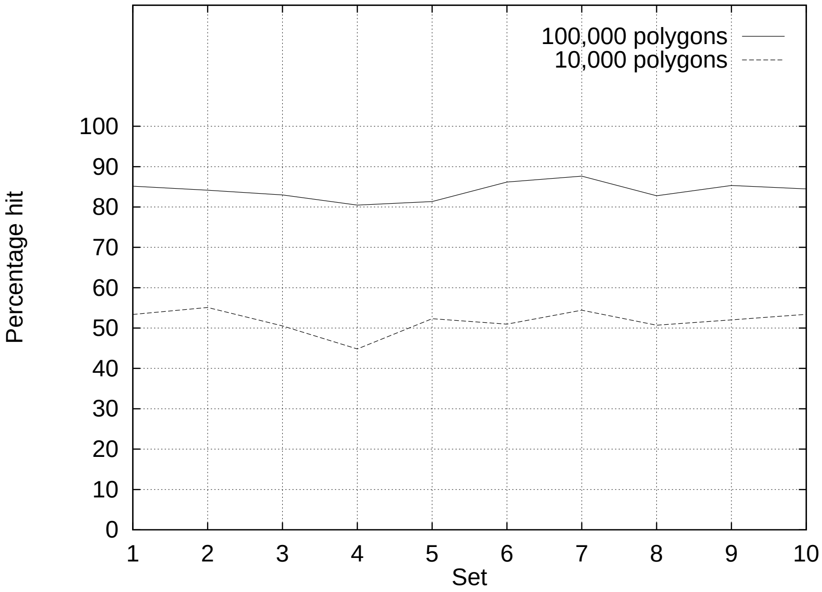


Figure 4.4: Results for star-shaped polygons on 25 points.

100,000 polygons generated on each set. For sets with 100 points we have results for 100,000 polygons generated on each set.

Among the methods for simple polygons, one method is significantly worse than the others: Incremental Construction & Backtracking. As can be seen in Fig. 4.5, less than half of all possible simple polygons are hit at least once when generating 10,000 polygons on 10 points. As could be expected, Permute & Reject exhibits an optimal hit rate of 100%. For the three algorithms with a modest CPU-consumption, the results are good for 2-Opt Moves, acceptable for Steady Growth and Steady Growth II, but rather poor for **Space Partitioning**. Note that the results improved when generating 100,000 polygons instead of 10,000 polygons, cf. Fig. 4.6: **2-Opt Moves** generates almost all polygons, **Steady Growth** lies around or above 90 percent, as does **Steady Growth II**, and **Space Partitioning** generates about between 80 and 90 percent of all possible polygons. To our surprise, the results for **Steady Growth II** are slightly better than for **Steady Growth**. In both tests the distribution of the polygons turned out to be highly non-uniform, though.

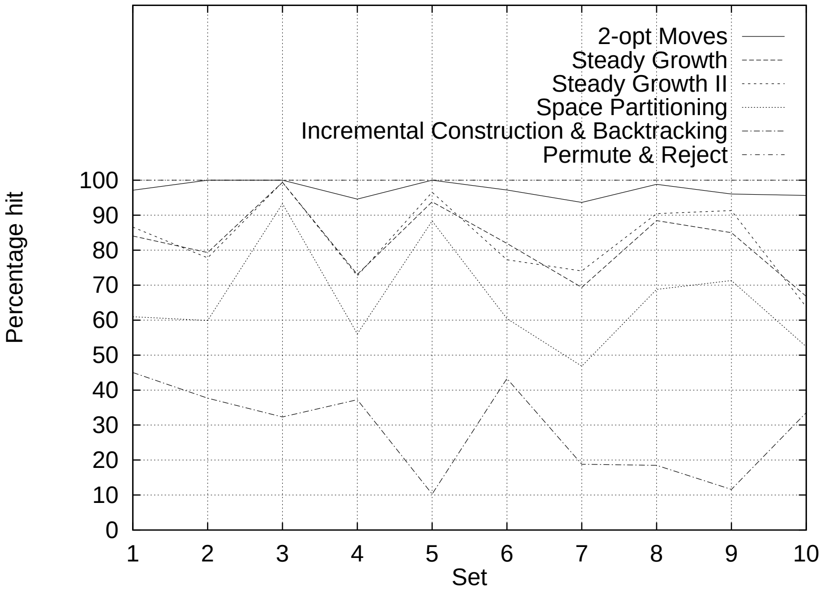


Figure 4.5: Results for 10,000 simple polygons on 10 points.

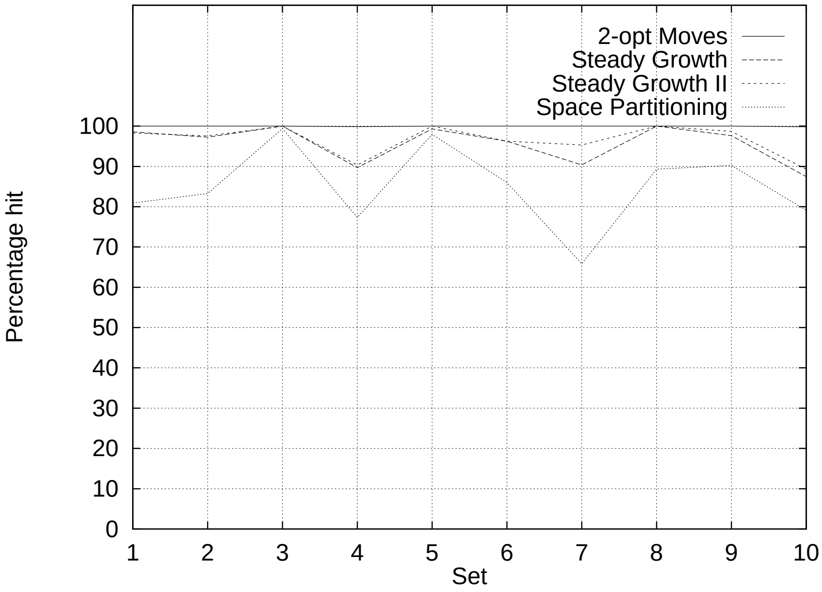


Figure 4.6: Results for 100,000 simple polygons on 10 points.

The results for 15 points give the same relative order of the algorithms as the tests on 10 points. **Permute & Reject** generates about 99% of all possible polygons. Note however, that we were not able to generate more than 2,500 polygons on 15 points despite more than four weeks of running time! Despite being very slow, **Incremental Construction & Backtracking** generates output with a low quality: It hits about 21% to 37 % of all polygons it could generate. The algorithms that are suited for practical purpose, are ranked in the same order as for 10 points: **2-Opt Moves** generates 62% - 85% of all possible polygons, **Space Partitioning** 30% to 48% of all possible polygons. Finally **Steady Growth** generates 50% to 80% of all possible polygons, whereas **Steady Growth II** is slightly better with 54% to 82%. These results are presented in Fig. 4.7.

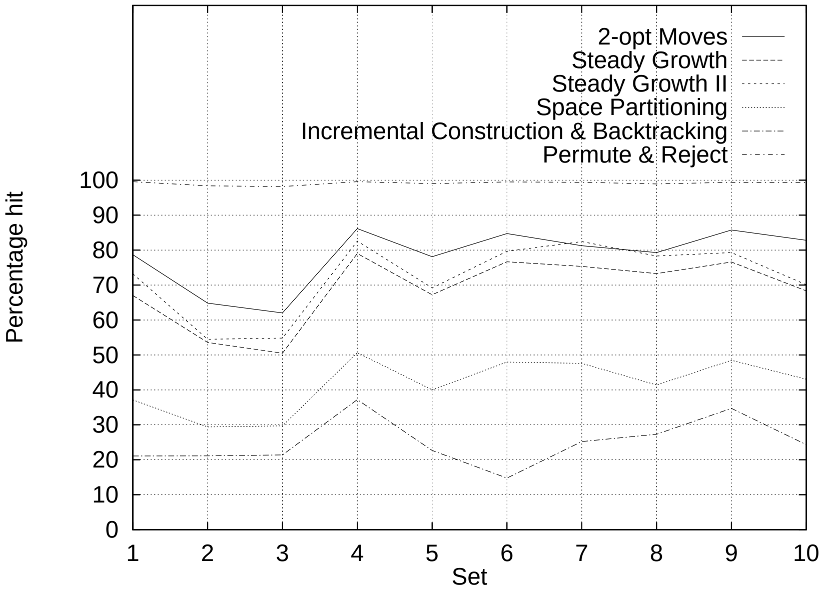


Figure 4.7: Results for 10,000 simple polygons on 15 points.

The results when generating 100,000 polygons on 15 points are worse than when generating 10,000 polygons: This is due to the reason that for 10,000 polygons, one would expect 10,000 different polygons, because for all point sets there exist more than 50,000 different simple polygons. However, when generating 100,000 polygons, for eight of the ten sets we expect 100,000 different polygons to be generated. Clearly, it is more likely to hit 10,000 different polygons out of about 200,000 than hitting 100,000 out of 200,000 polygons. However, again we have the same order of the algorithms: On the top, **2-Opt Moves** generates between 40% and 50% of the possible polygons, **Steady Growth** and **Steady Growth II** are between 25% to 40%, where **Steady Growth II** again has marginally better results than **Steady Growth**. Finally, **Space Partitioning** produces between 12% and 20% of the polygons expected. The results for 100,000 polygons on 15 points are depicted in Fig. 4.8.

However, when generating 100,000 polygons on 100 points all four algorithms **2-Opt Moves**, **Steady Growth**, **Steady Growth II** and **Space Partitioning** behaved optimally: They all generated exactly 100,000 different polygons! It is likely that this

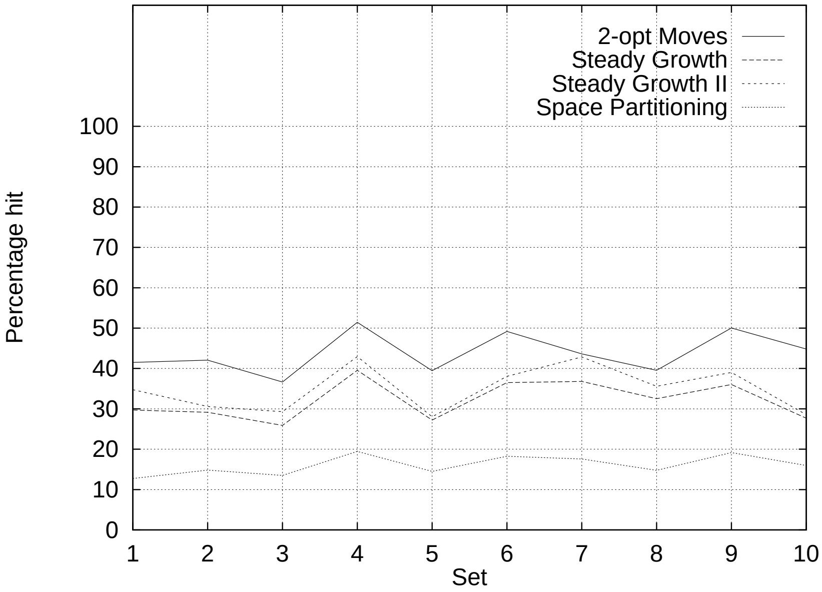


Figure 4.8: Results for 100,000 simple polygons on 15 points.

result is due to the fact that there exists an enormous number of simple polygons on 100 points. (We encountered sets with 20 points which already allowed more than three million simple polygons.) On one hand, our tests suggest that a user of any of these three algorithms need not worry about repeatedly generating the same "random" polygons when dealing with 100 or more points. On the other hand, the distribution of the polygons generated should not be expected to be (close to) uniform. (Any further statistical analysis of the distributions of the polygons generated had to be abandoned due to hardware constraints imposed on the CPU-time consumption and the available main memory and disk space.)

We note, though, that the polygons generated by **2-Opt Moves**, **Steady Growth**, **Steady Growth II** and **Space Partitioning** seem to have different characteristics for larger sets of points. For sets of 5,000 points, the polygons generated by **Steady Growth**, **Steady Growth II** and **Space Partitioning** exhibit a lot of "zigzagging" (for one of the "worst" examples encountered in our tests, cf. Fig. 4.9) whereas the polygons generated by **2-Opt Moves** are much more pleasing from a visual point of view, cf. Fig. 4.10. Presumably, the polygons generated by **2-Opt Moves** come closest to complex polygons actually occurring in practice. However, all polygons generated are quite distinct from a "smooth" polygon typically obtained by discretizing some free-form curves.

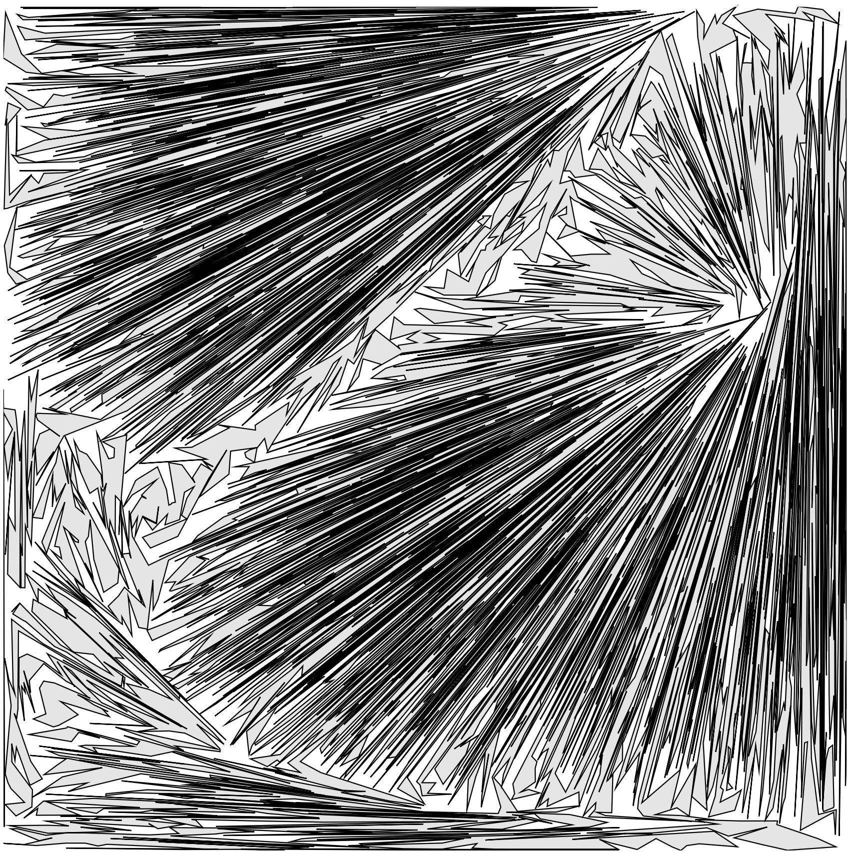


Figure 4.9: A polygon on 5,000 points exhibiting “zigzagging”.

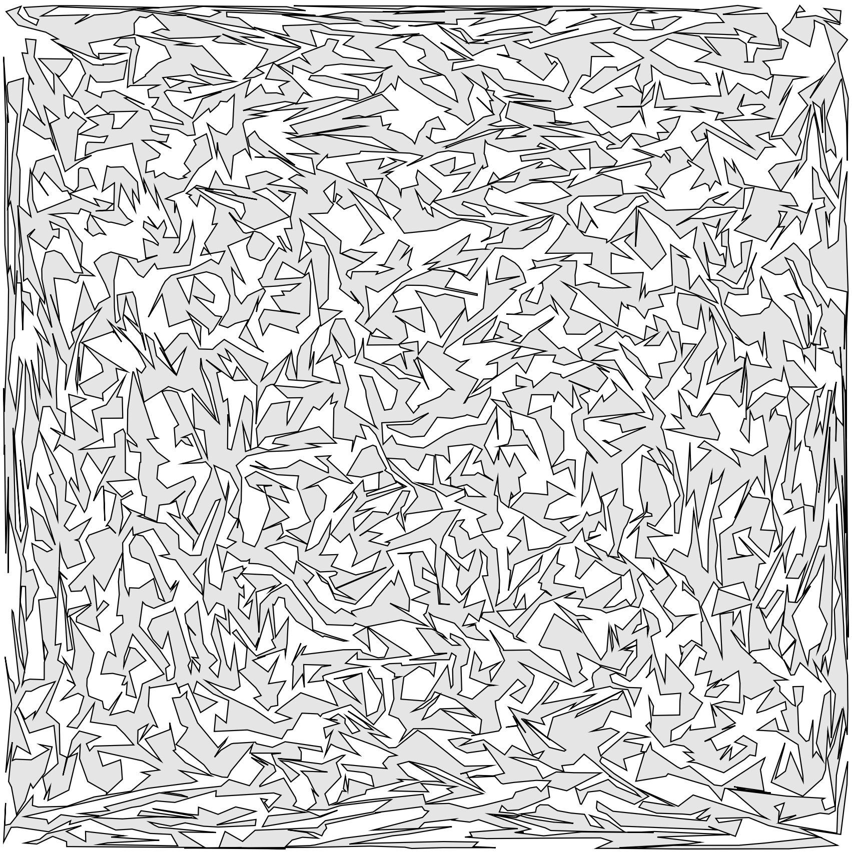


Figure 4.10: A polygon on 5,000 points without "zigzagging".

Chapter 5

Conclusion

5.1 Summary

We presented five heuristics for the random generation of simple polygons. Three of these heuristics, namely 2-Opt Moves, Steady Growth (II) and Space Partitioning, are suited for practical purposes. However, we experienced a clear trade-off between the quality of the heuristic and the running time. Thus, when the CPU-consumption is not at a premium, one can afford to generate a large variety of polygons by 2-Opt Moves. In order to achieve maximum speed Space Partitioning would be the method of choice. Steady Growth is slightly faster than 2-opt Moves but generates a less rich set of polygons. Steady Growth II, however, offers a good ratio of running time versus the number of different polygons generated.

It is very likely that the same trade-offs also hold true for the generation of random polygons on sets of 100 or more points. However, the class of simple polygons on such point sets is rich enough and the power of 2-opt Moves, Steady Growth (II) and Space Partitioning is large enough that any of them can be expected to yield fairly good results. In particular, it is quite unlikely that a simple polygon will be generated repeatedly by any of these three heuristics. In our experimental analysis, constraints imposed by our computing equipment on the available CPU time, memory, and on the disk space clearly turned out to be the limiting factor.

For the random generation of star-shaped polygons we presented a fast heuristic, Quick Star. It has about the same characteristics as Space Partitioning. Unfortunately, the enumeration algorithms Star Arrange and Star Universe do not work for sets with more than, say, 50 points since they require far too much space.

5.2 Open Problems

In order to enhance our statistical analysis, we would need to circumvent time and space constraints imposed by the hardware on the enumeration of all simple (respectively, of all star-shaped) polygons on $\mathcal{S}$. Also, it would be desirable to classify the classes of polygons which are likely to be generated by our heuristics in some intuitive manner.

From a theoretical point of view, it remains an open problem to generate polygons on a given set of vertices uniformly at random. From a practical point of view, it is not even clear what constitutes a good "random" polygon. A typical user may want to generate "random" polygons within some fuzzy subclass of polygons in order to best simulate the expected class of inputs for his application: e.g., generate random polygons which consist of two dominant "roughly convex" regions linked by a "roughly x-monotone" tunnel. Also, none of our heuristics generates polygons which resemble discretized free-form curves. (A polygon with 3,200 vertices which resembles a discretized free-form curve is depicted in Fig. 5.1.)

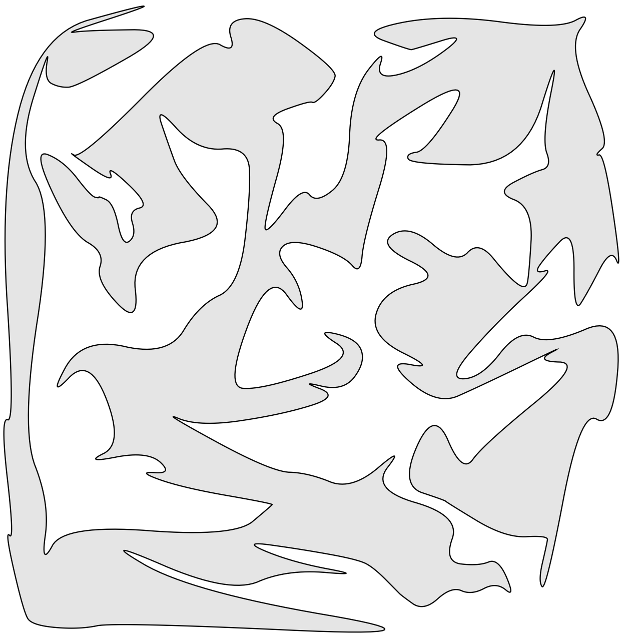


Figure 5.1: A polygon resembling a free-form curve.

Another topic for future work is the random generation of multiply-connected planar areas: Given a set $\mathcal{S}$ of n points and an integer $k <= \frac{n}{3}$, generate k random polygons on $\mathcal{S}$ which bound a multiply-connected planar area, cf. Fig. 5.2.

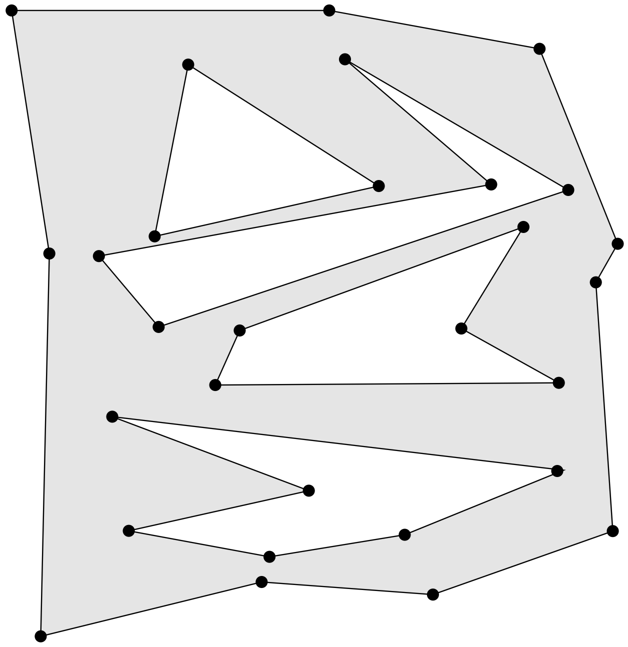


Figure 5.2: A multiply-connected planar area.

Bibliography

[AF91] D. Avis and K. Fukuda. A basis enumeration algorithm for linear systems with geometric applications. *Appl. Math. Lett.*, 4(5):39–42, 1991.

[AF92] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8:295–313, 1992.

[AKS84] A. Asano, S. Kikuchi, and N. Saito. A linear algorithm for finding Hamiltonian cycles in 4-connected maximal planar graphs. *Discrete Appl. Math.*, 7:1–15, 1984.

[ARS93] N. Alon, S. Rajagopalan, and S. Suri. Long non-crossing configurations in the plane. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 257–263, 1993.

[AS94a] M.D. Atkinson and J.-R. Sack. Uniform generation of binary trees in parallel. *J. Parallel Distrib. Comput.*, 23:101–103, 1994.

[AS94b] M.D. Atkinson and J.-R. Sack. Uniform generation of forests of restricted height. *Inform. Process. Lett.*, 50:323–327, 1994.

[BJH93] J. Bang-Jensen and P. Hell. Fast algorithms for finding Hamiltonian paths and cycles in in-tournament digraphs. *Discrete Appl. Math.*, 41:75–79, 1993.

[CD91] S. Chattopadhyay and P.P. Das. Counting thin and bushy triangulations of convex polygons. *Pattern Recogn. Lett.*, 12:139–144, 1991.

[Cha91] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[Cra78] I.K. Crain. The Monte-Carlo generation of random polygons. *Comput. Geosci.*, 4:131–141, 1978.

[DES93] L. Devroye, P. Epstein, and J.-R. Sack. On generating random intervals and hyperrectangles. *J. Comput. Graph. Stat.*, 2(3):291–307, 1993.

[DFJ94] M. Dyer, A.M. Frieze, and M. Jerrum. Approximately counting Hamilton cycles in dense graphs. In *ACM-SIAM Sympos. Discrete Algorithms*, 1994.

[Dil92] M.B. Dillencourt. Finding Hamiltonian cycles in Delaunay triangulations is NP-complete. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 223–228, 1992.

[Eps92] P. Epstein. Generating Geometric Objects at Random. Master's thesis, CS Dept., Carleton University, Ottawa K1S 5B6, Canada, 1992.

[ES92] P. Epstein and J.-R. Sack. Generating triangulations at random. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 305–310, 1992.

[FF83] T.I. Fenner and A.M. Frieze. On the existence of Hamiltonian cycles in a class of random graphs. *Discrete Math.*, 45:301–305, 1983.

[FJM$^+$94] A.M. Frieze, M. Jerrum, M. Molloy, R. Robinson, and N. Wormald. Generating and counting Hamilton cycles in random regular graphs. To appear, 1994.

[FM91] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, 1991.

[Fri88] A.M. Frieze. Finding Hamiltonian cycles in sparse random graphs. *J. Combin. Theory*, 44:230–250, 1988.

[Fri90] A.M. Frieze. Probabilistic analysis of graph problems. In G. Tinhofer, E. Mayr, H. Noltemeier, and M. Syslo in cooperation with R. Albrecht, editors, *Comput. Graph Theory*, pages 209–233. Springer, 1990.

[FS92] A.M. Frieze and S. Suen. Counting the number of Hamilton cycles in random digraphs. *Random Struct. Algorithms*, 3(3), 1992.

[GB82] D. Gouyou-Beauchamps. The Hamiltonian circuit problem is polynomial for 4-connected planar graphs. *SIAM J. Comput.*, 11:529–539, 1982.

[GMBS94] S. Sen Gupta, K. Mukhopadhyaya, B.B. Bhattacharya, and B.P. Sinha. Geometric classification of triangulations and their enumeration in a convex polygon. *Comput. Math. Applic.*, 27(7):99–115, 1994.

[Gol] Michael Goldwasser. An implementation for maintaining arrangements of polygons. Technical report, Dept. of CS, Stanford University, CA, USA.

[GS85] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[Hel93] M. Held. A Fast Incremental Algorithm for Computing the Voronoi Diagram of a Planar Shape. In *Communicating with Virtual Worlds (CGI'93)*, pages 318–329, Lausanne, Switzerland, June 1993. Springer-Verlag.

[Her89] J. Hershberger. An optimal visibility graph algorithm for triangulated simple polygons. *Algorithmica*, 4:141–155, 1989.

[JS87] B. Joe and R.B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT*, 27:458–472, 1987.

[KJR82] S. Kotz, N.L. Johnson, and C.B. Read, editors. *Encyclopedia of statistical sciences.* John Wiley & Sons, New York, 1982.

[Knu69] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1969.

[Knu76] D.E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, April–June 1976.

[KS83] M. Komlos and E. Szemeredi. Limit distributions for the existence of Hamiltonian cycles in a random graph. *Discrete Math.*, 43:55–63, 1983.

[Las96] M.J. Laszlo. *Computational Geometry and Computer Graphics in C++.* Prentice-Hall, Inc., 1996.

[Lau90] J.-P. Laumond. Connectivity of plane triangulations. *Inform. Process. Lett.*, 34:87–96, 1990.

[L'E94] P. L'Ecuyer. Uniform random number generation. *Ann. Oper. Res.*, 53:77–120, 1994.

[MRSW92] J.S.B. Mitchell, G. Rote, G. Sundaram, and G. Woeginger. Counting convex polygons in planar point sets. To appear, 1992.

[Nis90] T. Nishizeki. Planar graph problems. In G. Tinhofer, E. Mayr, H. Noltemeier, and M. Syslo in cooperation with R. Albrecht, editors, *Comput. Graph Theory*, pages 53–68. Springer, 1990.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994. ISBN 0-521-44592-2.

[OV91] J. O'Rourke and M. Virmani. Generating random polygons. Technical Report 11, CS Dept. Smith College, MA, USA, 1991.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[SB94] J.A. Shuffelt and H.J. Berliner. Generating Hamiltonian circuits without backtracking from errors. *Theoretical Comput. Sci.*, 132:347–375, 1994.

[Sch94] Otfried Schwarzkopf. A manual of Ipe. Technical report, Vakgroep Informatica, Universiteit Utrecht, the Netherlands, 1994.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1983.

[SZ93] J. Snoeyink and C. Zhu. Generating random monotone polygons. Technical Report 93-28, CS Dept. The University of British Columbia, BC, USA, 1993.

[vLS82] J. van Leeuwen and A.A. Schoone. Untangling a travelling salesman tour in the plane. In J.R. Mühlbacher, editor, *Proc. 7th Conf. Graphtheoretic Concepts in Comput. Sci. (WG 81) (Linz 1981)*, pages 87–98, München, 1982. Hanser.

[ZO95] T.C. Zhao and M. Overmars. Forms Library. *http://bragg.phys.uwm.edu/xforms*, 1995.

[ZSSM96] C. Zhu, G. Sundaram, J. Snoeyink, and J.S.B. Mitchell. Generating random polygons with given vertices. *Comput. Geom.: Theory Applic.*, to appear 1996.

Curriculum Vitae

Nachname: AUER

Vornamen: Thomas Alexander

Geburtsdatum: 8. September 1972

Nationalität: Österreich

wohnhaft in: Klammstr. 12, 6020 Innsbruck, Österreich

Vater: Dr. Auer Gerhard, Beamter,
wohnhaft in: Klammstr. 12, 6020 Innsbruck

Mutter: Auer Hildegard, Hausfrau,
wohnhaft in: Klammstr. 12, 6020 Innsbruck

Schulausbildung: Volksschule Innsbruck-Allerheiligen von 1979 – 1983
Akademisches Gymnasium Innsbruck von 1983 – 1991
Matura mit Auszeichnung 1991

Beginn des Studiums der Computerwissenschaften
im Oktober 1991
1. Diplomprüfung abgelegt am 22. Dezember 1993
Wechsel zum Studium der Angewandten Informatik
im März 1995