The Limited Workspace Model for Geometric Algorithms

by

Bahareh Banyassady

A dissertation submitted to the
Department of Mathematics and Computer Science
in conformity with the requirements for
the degree of Doctor of Natural Sciences

Freie Universität Berlin
Berlin, Germany
2018

Supervisor: Prof. Wolfgang Mulzer

Second reviewer: Prof. Marcel Roeloffzen

Date of defense: April 26, 2019

© Copyright by Bahareh Banyassady, November 2018. All rights reserved.

Abstract

Space usage has been a concern since the very early days of algorithm design. The increased availability of devices with limited memory or power supply – such as smartphones, drones, or small sensors – as well as the proliferation of new storage media for which write access is comparatively slow and may have negative effects on the lifetime – such as flash drives – have led to renewed interest in the subject. As a result, the design of algorithms for the *limited workspace model* has seen a significant rise in popularity in computational geometry over the last decade.

In this setting, we typically have a large amount of data that needs to be processed. Although we may access the data in any way and as often as we like, write-access to the main storage is limited and/or slow. Thus, we opt to use only higher level memory for intermediate data (e.g., CPU registers). Since the application areas of the devices mentioned above – sensors, smartphones, and drones – often handle a large amount of geographic (i.e., geometric) data, the scenario becomes particularly interesting from the viewpoint of computational geometry.

Motivated by these considerations, we investigate geometric problems in the limited workspace model. In this model the input of size n resides in read-only memory, an algorithm may use a workspace of size $s = \{1, \ldots, n\}$ to read and write the intermediate data during its execution, and it reports the output to a write-only stream. The goal is to design algorithms whose running time decreases as s increases, which provides a *time-space trade-off*.

In this thesis, we consider three fundamental geometric problems, namely, computing different types of Voronoi diagrams of a planar point set, computing the Euclidean minimum spanning tree of a planar point set, and computing the k-visibility region of a point inside a polygonal domain. Using several innovative techniques, we either achieve the first time-space trade-offs for those problems or improve the previous results.

Acknowledgments

On the way that lead me to this thesis, I was fortunate to encounter people who stimulated and influenced me in different ways. This accomplishment would not have been possible without them.

Foremost, I would like to express my gratitude to my advisor Prof. Wolfgang Mulzer for the continuous support of my Ph.D. research, for his motivation, enthusiasm, and sharing his immense knowledge. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing.

Besides my advisor, I would like to thank the rest of my Ph.D. committee and in particular the second reader of this thesis, Prof. Marcel Roeloffzen, for giving me the chance to benefit from their knowledge in the field. Special thanks go to the experts who were my coauthors in this research project André, Luis, Marcel, Matias, Prosenjit, Stephane and Yeganeh for their passionate participation and input, as well as to my mentor at BMS, Nataša, who inspired me to choose my career path.

My fellow members of the theory group Alexander, Andrei, Aruni, Astrid, Boris, Claudia, Frank, Günter, Heike, Helmut, Heuna, Ita, Jonas, Katharina, Klaus, Kristin, László, Lena, Ludmila, Man-Kwun, Matthias, Max, Nadja, Paul, Romain, Tamara, Tillmann, and Yannik are thanked for their feedback, cooperation, and friendship.

I owe a lot to my friends who supplied me with energy whenever I needed a recharge. Above all, I must express my very profound thankfulness to my parents, Mohammad and Zahra, to my boyfriend, Jan, to my sisters, Rayhaneh and Sarah, and to my brother in law, Kazem, for providing me with unfailing support and continuous encouragement throughout my years of study and writing this thesis.

Contents

Abstract **i**

Acknowledgments **ii**

Contents **iii**

List of Figures **v**

1 Introduction **1**
1.1 The Limited Workspace Model . . . 2
1.2 Related Models . . . 3
1.3 State of the Art . . . 5
1.4 Contributions . . . 7
1.5 Thesis Outline . . . 8
1.6 Publications . . . 8

I Problems on Planar Point Sets **9**

2 Preliminaries and Background on Point Sets **11**
2.1 Convex Hulls . . . 11
2.2 Voronoi Diagrams and Delaunay Triangulations . . . 14
2.3 Higher-Order Voronoi Diagrams . . . 17
2.4 Euclidean Minimum Spanning Trees . . . 23
2.5 Relative Neighborhood Graphs . . . 26

3 Voronoi Diagrams **31**
3.1 A Constant Workspace Algorithm for NVD and FVD . . . 31
3.2 A Time-Space Trade-Off for NVD and FVD . . . 35
3.2.1 Finding a Single Edge of Some Cells . . . 35
3.2.2 Finding All the Edges According to Their Incident Cells . . . 37

3.3 A Time-Space Trade-Off for the Family of HVDs 41
3.3.1 Relation Between Two Consecutive HVDs 41
3.3.2 The Recursive Procedure 44
3.3.3 Obtaining a Time-Space Trade-Off 47

4 Euclidean Minimum Spanning Trees **51**
4.1 Computing the Relative Neighborhood Graph 51
4.1.1 All the Incident Edges to Some Sites 51
4.1.2 Finding All the Edges of RNG 53
4.1.3 Edges of RNG in Sorted Order of Length 54
4.2 A Simple Time-Space Trade-Off for EMST 55
4.2.1 Structure of Face-Cycles 55
4.2.2 The Algorithm 57
4.3 Improvement via a Compact Representation of RNGs 59
4.3.1 The s-net Structure 59
4.3.2 Maintaining the s-net 62

II Problems on Polygonal Domains **65**

5 Preliminaries and Background on Polygonal Domains **67**
5.1 Visibility Region 68
5.2 k-Visibility Region 71
5.3 Sorting 76
5.4 Selection 77

6 k-Visibility Region **79**
6.1 Geometry of the k-visibility region 79
6.2 A Constant Workspace Algorithm for $V_k(P, q)$ 83
6.3 Time-Space Trade-Offs for $V_k(P, q)$ 85
6.3.1 Processing All the Vertices of the Polygon 86
6.3.2 Processing Only the Critical Vertices of the Polygon 90
6.4 Variants and Extensions 95

7 Conclusions **97**

Bibliography **99**

List of Figures

1.1 Memory cells in the limited workspace model . . . 2

2.1 Convex hull of a subset of the plane and of a finite planar set of points . . 11
2.2 Jarvis' gift-wrapping algorithm . . . 12
2.3 Convex hull algorithm in the streaming model . . . 13
2.4 An arbitrary triangulation of a planar set of sites . . . 14
2.5 The Delaunay triangulation of a planar set of sites . . . 15
2.6 The nearest site Voronoi diagram . . . 15
2.7 The farthest site Voronoi diagram . . . 16
2.8 The duality of Delaunay triangulation and nearest site Voronoi diagram . 16
2.9 Finding a single edge of a cell of NVD . . . 17
2.10 The Voronoi diagram of order 2 . . . 18
2.11 Illustration of the k-cells. . . . 18
2.12 Illustration of the new and the old k-vertices . . . 19
2.13 Illustration of the k-edges . . . 19
2.14 Property (I) of HVDs for corresponding sites of two adjacent k-cells . . . 20
2.15 Property (II) of HVDs for the k-edges lying inside a $(k+1)$-cell . . . 21
2.16 Property (III) of HVDs for the vertices in two consecutive diagrams . . . 21
2.17 An arbitrary spanning tree and the Euclidean minimum spanning tree . . 23
2.18 Kruskal's algorithm for EMST . . . 24
2.19 The relation between EMST and DT . . . 24
2.20 A path between endpoints of an edge e in the subgraph $\mathrm{DT}_{<e}$ . . . 25
2.21 The empty lens property . . . 26
2.22 Edges of the relative neighborhood graph . . . 27
2.23 An illustration of $\mathrm{EMST}(S) \subseteq \mathrm{RNG}(S) \subseteq \mathrm{DT}(S)$ . . . 27
2.24 Applying Kruskal's algorithm on the RNG . . . 28
2.25 The half-edges in RNG and their head and tail endpoints . . . 28
2.26 Face-cylces of RNG_i . . . 29
2.27 Predecessor and successor of a half-edge of RNG_i . . . 30

3.1 A starting ray from p intersecting the cell of p in FVD 32
3.2 Finding the starting ray in FVD 32
3.3 Finding the first edge of a cell of NVD and a cell of FVD 33
3.4 Finding the next edges of a cell of NVD and a cell of FVD 34
3.5 Finding the lines spanned by an edge of a cell of NVD, for s cells 36
3.6 Finding the endpoints of an edge of a cell of NVD, for s cells 36
3.7 The big and small cells in NVD 37
3.8 The first phase of the NVD algorithm using $O(s)$ cells of workspace . . . 38
3.9 The second phase of the NVD algorithm using $O(s)$ cells of workspace . . 39
3.10 The third phase of the NVD algorithm using $O(s)$ cells of workspace . . . 40
3.11 The relevant and non-relevant k-half-edges 41
3.12 The $(k+1)$-intervals assigned to the relevant k-half-edges 42
3.13 Finding the first $(k+1)$-half-edge of the interval assigned to a k-half-edge 43
3.14 Finding $(k+1)$-half-edges using s half-edges from VD^k or VD^{k+1} 44
3.15 The k-half-edges incident to big and small cells of VD^k 46
3.16 The allocated buffers for the processor Voro-k 47
3.17 The pipeline between all the processors and their allocated memory buffers 49

4.1 Necessity of the second phase of the RNG algorithm 52
4.2 Finding all the neighbors of s sites in the graph RNG 53
4.3 Constructing RNG by finding all the neighbors of all the sites 54
4.4 The components and the face-cycles containing endpoints of an edge . . . 56
4.5 Finding the next edge of a given edge on a face-cycle of RNG_i 56
4.6 Finding the predecessor and the successor of an edge in RNG_i 57
4.7 Traversing the face-cycles of RNG_i containing endpoints of an edge 58
4.8 The s-net 59
4.9 The set H containing the batch edges, their successors and the net-edges . 60
4.10 The auxiliary graph G with the compressed edges 61
4.11 Adding the batch edges using the component structure of G 62
4.12 Distributing the new net-edges on the face-cycles 62

5.1 A simple polygon and a polygon with holes 67
5.2 Visible points and the visibility region 68
5.3 The chords defined by reflex vertices 68
5.4 The weak visibility region 69
5.5 The visible chain between the shadows of two visible reflex vertices 70
5.6 Edge-to-edge visiblity 70
5.7 k-visible points 71
5.8 k-visibility region 72
5.9 Comparing the k-visibility region and the $(k-1)$-visibility region 72
5.10 The rank of edges in the edge list of a ray 73
5.11 The start/ end critical vertices and non-critical vertices 73
5.12 The chains on the boundary of the polygon 74
5.13 Two notions of the k-visibility region in a polygon and in the plane 75

6.1 Updating the chain list of a rotating ray 80
6.2 Edge list of a ray through start and end vertices 80
6.3 The k-visible edges in the edge list of the rays through critical vertices . . 81
6.4 The rank $(k + 1)$ edge on the rays before and after a start/end vertex . . 82
6.5 The window on a ray though a start/end vertex 82
6.6 Finding the $(k + 1)$-edge on a ray using the one on the previous ray . . . 84
6.7 The elements in the balanced binary search tree T 86
6.8 Updating T in a counterclockwise sweep of the ray 87
6.9 Reporting the k-visible sunchains in a cone 88
6.10 Insering new chains into T 91
6.11 Removing chains from T 92
6.12 The k-visibility region in a polygon with holes 95
6.13 The k-visibile subsets of a set of line segments 96

CHAPTER 1

Introduction

Memory constraints have been studied since the introduction of computers, initially motivated by the high cost of memory (see, for example, Pohl [Poh69]). The first computers often had limited memory compared to the available processing power. As hardware progressed, this gap narrowed and the amount of memory available to computers grew exponentially. Thus, other concerns became more important, and the focus of algorithms research shifted away from memory constraints. Therefore, nowadays, many algorithms have been developed with little or no regard to the amount of memory used.

On the other hand, in recent years, we have seen an explosive growth of small distributed devices such as tracking devices and wireless sensors. These gadgets are small, have only limited energy supply, are easily moved, and should not be too expensive. To accommodate these needs, the amount of memory on them is tightly budgeted. Thus, programs that are oblivious to space resources are not suitable for such a setting. Furthermore, even though memory became drastically cheaper than before, input data sizes are growing rapidly. Hence, memory constraints became again an important issue for these new devices as well as for huge datasets that have become available through cloud computing. This poses a significant challenge to software developers and algorithm designers: how to create useful and efficient programs in the presence of strong memory constraints?

An easy way to model algorithms with memory constraints is to assume that the input is stored in a read-only memory. This is appealing for several reasons. From a practical viewpoint, writing to external memory is often a costly operation, e.g., if the data resides on a read-only medium such as a DVD or on hardware where writing is slow and wears out the material, such as flash memory. Similarly, in concurrent environments, writing operations may lead to race conditions. Thus, it is useful to limit or simply disallow writing operations on the input media.

From a theoretical viewpoint, keeping the working memory separate from the (read-only) input memory has also advantages. Basically, it allows for a more detailed accounting of the space requirements of an algorithm and for a better understanding of the required resources. In fact, this is exactly the approach taken by computational complexity theory. Here, one defines complexity classes that model *sublinear* space requirements, such as the complexity class of problems that use a logarithmic amount of space [AB09].

With this situation in mind, we focus on designing algorithms that need only a limited number of cells of the read-write memory, while the input resides in read-only memory, and the output is usually written sequentially to a write-only stream. In the following chapter, we will introduce our model more formally.

1.1 The Limited Workspace Model

The present notion of a *limited workspace algorithm* was introduced to the computational geometry community by Tetsuo Asano [Asa08]. Initially, the model postulated a workspace that consists of a constant number of cells [AMRW11, AMW11]. Over the years, this was extended to also allow more workspace in exchange for a better running time, which is known as time-space trade-off. In the following, we describe the most general variant of the model which we will refer to as *the limited workspace model*. Studying geometric algorithms in the limited workspace model constitutes the focus of this thesis.

The model is similar to the standard word random access machine, word RAM, in which the memory is organized as a sequence of *cells* that can be accessed in constant time via their *addresses*. Each memory cell stores a single data word [Knu97]. In contrast to the standard word RAM, the limited workspace model distinguishes two kinds of cells: (i) *read-only cells* that store the input; and (ii) *read-write cells* that constitute the algorithm's *workspace*. A cell of the workspace can store either an integer of $O(\log n)$ bits, a pointer to some input cell, or a root of some polynomial of bounded degree that depends on a fixed number of input variables (for example, the intersection point of two lines, each passing through two input points). Here, n denotes the input size (measured in the number of cells). We denote the number of cells of the workspace by s.

Typically, the output is larger than the workspace. Thus, we assume that the output is written sequentially to a dedicated write-only stream. Once a data item is written to the output stream, it cannot be modified or even accessed again by the algorithm; see Figure 1.1. It usually depends on the algorithmic problem at hand how exactly the output should be structured. As usual, the running time of an algorithm on a given input is measured as the number of elementary operations that it performs. The space usage is counted as the number of cells in the workspace. Note that the input does not contribute to this count, but any other memory consumption does (like memory implicitly allocated during recursion).

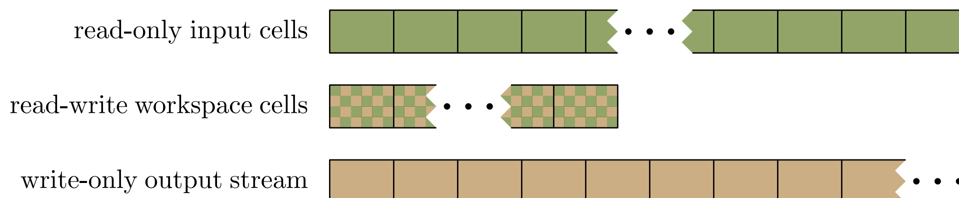

Figure 1.1: The different types of memory cells available in the limited workspace model.

Although the objective is to have algorithms that are fast and at the same time use little – ideally constant – workspace, it is normally not possible to achieve both goals simultaneously. Thus, the aim is to balance the two. Often, this results in a *time-space trade-off*: as more cells of workspace become available, the running time decreases. Now, the precise relationship between the running time and the available space becomes the main focus of our attention. For many problems, this dependency is linear, i.e., by doubling the amount of workspace, the running time can be halved; see for example [ABB$^+$13, HP14, OA17] . However, recent research has uncovered a wide range of possible trade-offs that often interpolate smoothly between the best known results for constant and for $O(n)$ cells of workspace; see for example [DE14, BKLS14]

1.2 Related Models

Designing algorithms that require little working memory is a classic and well-known challenge in theoretical computer science. Over the years, it has been attacked from many different angles. In the following, we briefly survey some widely studied models that restrict the available workspace or the way the data is accessed. See also the survey by Korman [Kor16] on the memory-constrained models.

Streaming algorithms. One of the most restrictive models that has been considered is the *one-pass* or the *streaming* model. In a typical streaming algorithm, the elements of the input can be scanned only once and in a sequential fashion. The underlying motivation originates from the desire to process huge amounts of data, such as traffic data on the internet, and in an online fashion. The goal is to perform some meaningful computation on the input and usually to approximate the solution while the algorithm is using as little workspace as possible to accomplish its task. Of course, the size of the random-access workspace should be sublinear in the size of the input, otherwise there would be no difference to the classic unconstrained model.

A natural extension of the above setting is the *multi-pass* model, in which the input can be scanned a constant number of times; however, still in a sequential fashion and using a limited workspace. In this model, the number of passes that an algorithm makes over the input is accounted as a measure. Basically, one looks for a trade-off between the number of passes and either the size of the workspace or the quality of the approximation [Kor16].

The selection problem has been studied in the multi-pass model since the early work of Munro and Paterson [MP80] and Frederickson [Fre87]. Many further problems have been considered since then (see, e.g., the survey by Muthukrisnan [Mut05]). There are also several results on geometric problems in the streaming model [Ind04, Cha06], mostly dealing with problems concerning clustering and extent measures [AHPV04], but also with classic questions, such as computing convex hulls or low-dimensional linear programming [CC07].

Read-only random-access model. As the name of *read-only random access* model implies, in this model, the input is stored on a read-only randomly-accessible medium. The main difference to the streaming model is that one has random access to each element of the input, while in the streaming model, each element of the input can be read only once (or constant number of times). The extreme version of the multi-pass model that imposes no restriction on the number of passes coincides with the random access model. Research on this model focuses on either computability (i.e., determining whether a particular problem is solvable with a workspace of fixed size), or the design of efficient algorithms whose running time is not much worse compared to the case in which no space constraints exist [Kor16].

In computational geometry, this model was first introduced by Asano [Asa08] under the name of *constant workspace model*. The constant workspace model is the case of the read-only random-access model when the algorithm can use only a constant number of read-write memory cells to accomplish its task. Assuming the size of the workspace as a parameter, we accomplish time-space trade-offs in the read-only random-access model, which is the main focus of this thesis.

In *computational complexity theory*, the algorithms in the constant workspace model have been studied for many years under the name of *log-space* algorithms. The classic complexity class LOGSPACE contains all algorithmic problems that can be solved with a workspace which has only a logarithmic number (in the input size) of bits [AB09, Gol08]. The research on LOGSPACE has led to several surprising insights [Imm88, Sze87, Sav70], perhaps most recently the *st*-connectivity algorithm by Reingold [Rei08] for undirected graphs, an unexpected application of expander graphs. The focus in computational complexity is mainly on what can be done *in principle* in LOGSPACE. Obtaining LOGSPACE-algorithms with a low running time is usually a secondary concern.

In-place and restore models. Unlike in the previous models, the *in-place* model assumes that the input resides in a memory that can be read and written arbitrarily. The algorithm may use the input array as working space and may rearrange the elements of the input or even modify them during the computation. The algorithm may use only a constant number of additional memory cells. This means that small, multi-purpose data structures can be encoded through appropriate permutation of the input, and some algorithms in this model have even achieved a running time comparable to those in unconstrained settings. However, it is still challenging to encode complex data structures using the input elements, and therefore, the model severely restricts the algorithmic options at our disposal.

The classic example of an in-place algorithm is heap-sort which, in addition to the input array, requires only constantly many cells of workspace for the loop and for the array indices. In computational geometry, in-place algorithms have been developed for many problems, e.g., computing the convex hull and computing the Voronoi diagram of a given planar point set [BIK$^+$02, BCC04, CC08, CC10].

A more restrictive version of the in-place model, which is called the *restore* model, has been introduced by Chan *et al.* [CMR14]. In the *restore* model, the algorithms

are allowed to modify the input array during the computation, but it is required that the original input permutation is restored by the end of the computation. Thus, it retains at least one of the advantages of the read-only model, namely, that the input is not destroyed after the execution of the algorithm. This can play an important role in certain applications [CMR14]. In particular, this model can potentially be useful in the design of in-place algorithms, when one encounters subproblems that need to be solved by subroutines, and these subroutines must leave the array in its original state by the time they finish, see [CMR14] for further details.

Succinct data structures. In *succinct data structures*, the goal is to minimize the precise number of bits that are needed to represent the data, getting as close to the entropy bound as possible [Nav16, Jac88]. At the same time, one would like to retain the ability to support the desired data structure operations efficiently. Although this kind of approach drastically reduces the memory needed, in many cases a workspace of $\Omega(n)$ bits are still necessary, and also the input data structure cannot be read-only since it may need to be used as a working space. In computational geometry, succinct data structures have been developed for classic problems like range searching, point location, or nearest neighbor search [He13].

To bring our model into perspective, we compare it with the related models. Unlike the typical viewpoint from computational complexity theory, our goal is to find the best running time that can be achieved with a given space budget. In contrast to streaming algorithms, we may read the input repeatedly and with random access. Unlike in-place algorithms, our input resides in read-only memory, and the workspace can potentially contain arbitrary data. When analyzing the space usage of an algorithm, we typically ignore constant factors and lower order terms, whereas these play a crucial role in succinct data structures.

1.3 State of the Art

As mentioned above, the initial investigations on geometric algorithms with limited workspace focused on the case that only a constant number of workspace cells are available, but by now, we have time-space trade-offs for most problems that interpolate between the constant and the linear workspace regime.

Table 1.1 summarizes the best known algorithms for some geometric problems in the limited workspace model. We have categorized them into four groups: problems on data sets, e.g., sorting; problems on point sets, e.g., Voronoi diagram; and problems on polygonal domains, e.g., visibility, which itself is partitioned into two other categories. Indeed, the intricate web of relationships between the three problems (i) triangulation of a simple polygon; (ii) balanced partition of a simple polygon; and (iii) the shortest path in a simple polygon induced us to put them in an individual group. See the recent survey by Banyassady *et al.* [BKM18a] for more details.

Problem	Running Time	Space	Source
shortest path in a tree	n	1	[AMW11]
all nearest larger neighbors	$n \log_s n$	s	[AK13]
sorting	$n^2/(s \log n) + n \log s$	s	[AEK13]
convex hull of a point set	$n^2/(s \log n) + n \log s$	s	[DE14]
triangulation of a point set	$n^2/s + n \log s$	s	[AMRW11, ABOS17]
Voronoi diagram/Delaunay triangulation	$n^2 \log s/s$	s	[KMvR+17, BKM+18b] [This thesis]
Voronoi diagrams of order 1 to K ($K \leq \sqrt{s}$)	$n^2 K^6 \text{polylog}(s)/s$	s	[BKM+18b] [This thesis]
Euclidean minimum spanning tree	$n^3 \log s/s^2$	s	[AMRW11, BBM18][This thesis]
triangulation of a simple polygon	n^2/s	s	[OA17, AKP+16]
balanced partition of a simple polygon	n^2/s	s	[OA17]
shortest path in a simple polygon	n^2/s	s	[AMW11, HP16, OA17]
triangulation of a monotone polygon	$n \log_s n$	s	[AK13]
visibility in a simple polygon	$n^2/2^s + n \log^2 n$	s	[BKLS14]
k-visibility in a polygonal domain	$n^2/s + n \log s$	s	[BBB+18] [This thesis]
weak visibility in a simple polygon	n^2	1	[Abr13]
minimum link path in a simple polygon	n^2	1	[Abr13]
convex hull of a simple polygon	$n^2 \log n/2^s$	s^*	[BKL+15]
convex hull of a simple polygon	$n^{1+1/\log s}$	s^{**}	[BKL+15]
common tangents of two disjoint polygons	n	1	[Abr15, AW16]

Table 1.1: A selection of problems and the best known running times in the limited workspace model. The O-notation has been omitted in the bounds. If the space usage is given as s, then s may range from 1 to n. For s^*, it may range from 1 to $o(\log n)$, and for s^{**} it ranges from $\log n$ to n. The running times for k-visibility and for the convex hull of a simple polygon have been simplified.

1.4 Contributions

In this chapter, we briefly explain the problems that we consider in this thesis and the results that we obtain to solve those problems in the limited workspace model. In all the following problems, we assume that the input is given in a read-only array of size $O(n)$ cells, and there is an additional read-write workspace of $O(s)$ cells, where $s = \{1, \ldots, n\}$ is a parameter of the model.[1] An algorithm may use this workspace to write the intermediate data, and it reports the output on a write-only stream. Such an algorithm is called an s-workspace algorithm.

Computing variants of Voronoi diagrams for a point set. Let S be a planar set of n point-*sites* in general position. For $k \in \{1, \ldots, n-1\}$, the Voronoi diagram *of order* k for S is obtained by subdividing the plane into *cells* such that points in the same cell have the same set of nearest k neighbors in S. The (*nearest site*) *Voronoi diagram* (NVD) and the *farthest site Voronoi diagram* (FVD) are the particular cases of $k = 1$ and $k = n - 1$, respectively. We develop a deterministic s-workspace algorithm for computing NVD and FVD for the set S that runs in $O\big((n^2/s)\log s\big)$ time. Moreover, we generalize our s-workspace algorithm so that for any given $K \in \{1, \ldots, O(\sqrt{s})\}$, we can compute the family of all higher-order Voronoi diagrams of order $k = 1, \ldots, K$ for S in $O\big(\frac{n^2K^5}{s}(\log s + K\log K)\big)$ total deterministic time or $O\big(\frac{n^2K^5}{s}(\log s + K\,2^{O(\log^* K)})\big)$ total expected time.

Computing the Euclidean minimum spanning tree of a point set. Let S be a planar set of n point-*sites* in general position. Consider the complete graph G on S where the weight of each edge is the Euclidean distance between its endpoints. The *Euclidean minimum spanning tree* (EMST) for S is defined as the spanning tree of G which has the minimum weight among all the spanning trees of G. We provide an s-workspace algorithm that computes the EMST for S in total $O\big((n^3/s^2)\log s\big)$ deterministic time.

Computing the k-visibility region of a point inside a polygonal domain. Let P be a simple polygon with n vertices, and let $q \in P$ be a point in P. Let $k \in \{0, \ldots, n-1\}$. A point $p \in P$ is *k-visible* from q if and only if the line segment pq crosses the boundary of P at most k times. The *k-visibility region* of q in P is the set of all points that are k-visible from q. We study the problem of computing the k-visibility region of q in P in the limited workspace model. We present an s-workspace algorithm that reports the k-visibility region of q in P in $O(cn/s + c\log s + \min\{\lceil k/s\rceil n, n\log\log_s n\})$ expected time. Here, $c \in \{1, \ldots, n\}$ is the number of *critical vertices* of P for q, where the k-visibility region of q may change. We generalize this result for polygons with holes and for sets of non-crossing line segments.

[1] The assumption that we have $O(s)$ cells instead of exactly s cells of workspace is for the sake of a simple presentation. Thus, when describing our algorithms, we can ignore constant factors in the space usage. The precise constant is a function that only depends on the implementation of the algorithms.

1.5 Thesis Outline

After the introduction chapter, the thesis is divided into two parts. In Part I we investigate geometric problems on point sets. In Chapter 2, we present preliminaries and background on problems that concern point sets, as well as the notations and definitions that are used in Part I. Next, in Chapter 3, we provide our algorithms to compute the nearest site Voronoi diagram, farthest site Voronoi diagram, and the family of higher order Voronoi diagrams for a given planar set of points. Last, in Chapter 4, we describe a simple time-space trade-off for computing the Euclidean minimum spanning tree for a given planar set of points, and we introduce *s-nets* which are compact representation of planar graphs. Finally, we explain how to use the s-nets to speed up our algorithm.

Part II is devoted to problems on polygonal domains, and it consists of two chapters. In Chapter 5, we review preliminaries and background on problems that deal with polygonal domains, as well as definitions and notations that are used in Part II. Furthermore, we explain some techniques that are applied in our algorithms in the next chapter. In Chapter 6, we tackle the problem of computing the k-visibility region of a point in a polygonal domain. First, we describe a constant workspace algorithm for this problem. Then, we use the same ideas to obtain a time-space trade-off which can be easily described and understood. Last, we explain how to improve the running time of our algorithm with the help of some data structures.

Finally, in Chapter 7, we provide a conclusion about geometric problems in the limited workspace model and some suggestions for further research in this topic.

1.6 Publications

The results that are covered in this thesis have appeared in the following publications.

[BKM+18b] Bahareh Banyassady, Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Improved time-space trade-offs for computing voronoi diagrams. *Journal of Computational Geometry (JoCG)*, 7(2):19–45, 2018.
[*A preliminary version appeared in proceedings of STACS 2017.*]

[BBM18] Bahareh Banyassady, Luis Barba, and Wolfgang Mulzer. Time-space trade-offs for computing Euclidean minimum spanning trees. In *Proceedings of 13th Latin American Theoretical Informatics Symposium (LATIN)*, pages 108–119, 2018.
[*Co-winner of the Alejandro López-Ortiz best paper award.*]

[BBB+18] Yeganeh Bahoo, Bahareh Banyassady, Prosenjit Bose, Stephane Durocher, and Wolfgang Mulzer. A time-space trade-off for computing the k-visibility region of a point in a polygon. *Journal of Theoretical Computer Science (TCS)*, 2018.
[*A preliminary version appeared in proceedings of WALCOM 2017.*]

Part

Problems on Planar Point Sets

Chapter 2

Preliminaries and Background on Point Sets

In this chapter, we introduce basic notations and preliminaries on geometric problems in the limited workspace model that deal with point sets in the plane. Some of the most basic geometric problems that concern point set in the plane are convex hulls, Voronoi diagrams, Euclidean minimum spanning trees, and some related structures. These problems have also been an early focus in the study of the limited workspace model. Here, we will give an overview of known results on these problems.

2.1 Convex Hulls

One of the most basic problems in computational geometry is the computation of planar *convex hulls*. In the following we define the convex hull and briefly present some algorithms for computing it.

Definition. A subset $\mathcal{S}$ of the plane is called *convex* if and only if for any pair of points $p, q \in \mathcal{S}$ the line segment $\overline{pq}$ is completely contained in $\mathcal{S}$. The *convex hull* of a set $\mathcal{S}$ is the smallest convex set $\mathcal{S}'$ that contains $\mathcal{S}$. More precisely, it is the intersection of all convex sets that contain $\mathcal{S}$ [BCvKO08]. See Figure 2.1a.

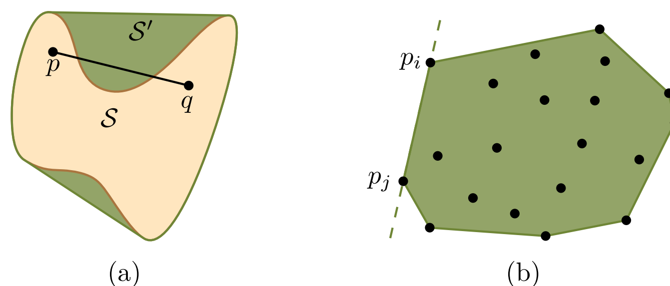

Figure 2.1: (a) A non-convex subset of the plane, $\mathcal{S}$, and its convex hull, $\mathcal{S}'$. (b) The convex hull of a finite set of points in the plane.

Here, we only consider the problem of computing the convex hull of a finite set of points $S = \{p_1, \ldots, p_n\}$ in the plane, which is denoted by $\mathrm{CH}(S)$. It can easily be observed that $\mathrm{CH}(S)$ is a convex polygon and that the line segment $\overline{p_i p_j}$, for any $p_i, p_j \in S$, is an edge of $\mathrm{CH}(S)$ if and only if all the points of $S \setminus \{p_i, p_j\}$ lie to the right of the directed line through p_i and p_j [BCvKO08]. See Figure 2.1b. These observations help us to understand the geometry of the problem and to design several algorithms to compute $\mathrm{CH}(S)$, i.e., to list its vertices along $\partial\,\mathrm{CH}(S)$ starting from an arbitrary one.[1] It is well-known that $\mathrm{CH}(S)$ can be computed in $O(n \log n)$ time when $O(n)$ cells of workspace are available; see the book by de Berg *et al.* for more details [BCvKO08].

Constant Workspace. In one of the first papers that investigated geometric problems with limited workspace, Asano *et al.* [AMRW11] observed that the convex hull of S can be found in $O(n^2)$ time when $O(1)$ cells of workspace are available. This is through a straightforward application of Jarvis' classic gift-wrapping algorithm [Jar73].

The essence of the algorithm is as follows: we scan S to find its leftmost point, and we denote this point by p_{leftmost}. This takes $O(n)$ time, and it uses only $O(1)$ cells of workspace. We output p_{leftmost} as a point on $\partial\mathrm{CH}(S)$, and we store it as the current point p_{current}. Then, we select the point $p_{\text{next}} \in S$ such that all the points in S are on the left side of the directed line through p_{current} and p_{next}. This point may be found in $O(n)$ time by comparing the polar angles of all the points with respect to p_{current} taken for the center of polar coordinates, and using only $O(1)$ cells of workspace for p_{current} and p_{next}; see Figure 2.2. By definition, p_{next} is the next counterclockwise point on $\partial\mathrm{CH}(S)$, thus we output p_{next}. Then, we store p_{next} in p_{current}, and we repeat the above procedure in order to find the next counterclockwise points on $\partial\mathrm{CH}(S)$, one after another, until we reach p_{leftmost} again, i.e., until we select the point p_{leftmost} as p_{next}. Clearly, we have to repeat this procedure $h = O(n)$ times, where h is the number of vertices on $\partial\mathrm{CH}(S)$, which makes a total running time of $O(n^2)$. The algorithm stores only a constant number of indices as well as the variables p_{leftmost}, p_{current} and p_{next}.

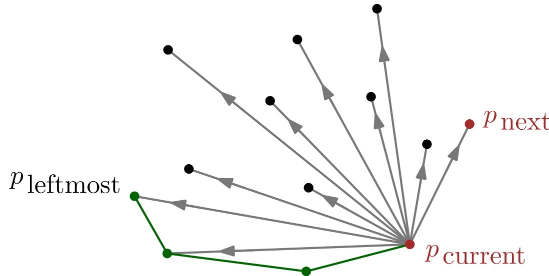

Figure 2.2: An intermediate step of the algorithm in [AMRW11] for a given set of points S. At the current step, three edges of $\mathrm{CH}(S)$ have been reported, and now the algorithm selects p_{next} as the next counterclockwise point on $\partial\,\mathrm{CH}(S)$ by comparing the polar angles of all the points with respect to p_{current}.

[1] Throughout this thesis, for any finite set $\mathcal{S}$, we denote the boundary of $\mathcal{S}$ by $\partial\mathcal{S}$.

Time-Space Trade-offs. As an extension of the constant workspace algorithm for computing CH(S), we first mention the trade-off presented by Chan and Chen [CC07] in the multi-pass model, which can also be used in the limited workspace model. Their algorithm runs in $O(n^2/s + n \log s)$ time, and it uses $O(s)$ cells of workspace. The algorithm imitates the classical Graham's scan which takes a lexicographic sorting[2] of the points of S and builds their upper-hull (and lower-hull) by testing repeatedly whether three consecutive points make a right turn or not.

In the limited workspace model, it is not possible to store the lexicographic sorting of the points of S. Instead, Chan and Chen present a clever subroutine to select a batch of s lexicographically consecutive points of S in $O(n)$ time using $O(s)$ cells of workspace, see Chapter 5.3 for details. Such a batch of s points forms a vertical slab containing s points of S. Their algorithm selects the s points in the leftmost slab, called σ, and sorts them in $O(s \log s)$ time. Then, using any of the known $O(s \log s)$ time convex hull algorithms, it constructs the upper-hull of the points in σ. Finally, among the points on the upper-hull of σ, it eliminates the ones which do not appear on CH(S).

This is done in a Graham-scan fashion by doing a right turn check for every point on the right of σ with the last segment of the tentative upper-hull of σ. Whenever a point with a left turn is detected (i.e., a point above the last segment) the left turn is resolved by pruning the points on the tentative upper-hull in the reverse order; see Figure 2.3. This results in finding the edge of CH(S) which intersects the right wall of σ and takes $O(n)$ time. The algorithm outputs the remaining points on the upper-hull of σ, since they belong to CH(S). Then, it proceeds to the next slab containing s points and it performs the same procedure as above.

Since there are $O(n/s)$ slabs and since processing each slab takes $O(n + s \log s)$ time, the total running time of the algorithm is $O(n^2/s + n \log s)$. The space usage for storing the points in the current slab as well as some constant amount of information per each such a point is $O(s)$ cells of workspace.

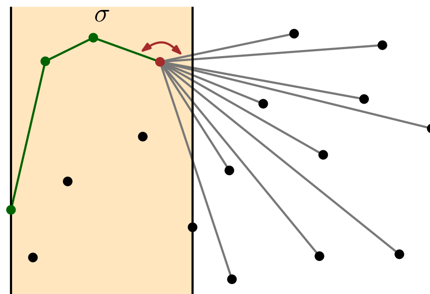

Figure 2.3: An intermediate step of the algorithm in [CC07]. At the current state, the algorithm has found the upper-hull of the points in the slab σ. Now, the last segment of the tentative upper-hull will be eliminated, since it makes a left turn with some of the points on the right of σ.

[2] Lexicographic order of the points of S means that first we sort the points according to their x-coordinate, and if some points have the same x-coordinate, then we sort them by their y-coordinate.

Another time-space trade-off for computing CH(S) was later presented by Darwish and Elmasry [DE14]. Their algorithm is quite similar to the algorithm by Chan and Chen at a high level. The main difference is that the algorithm by Darwish and Elmasry uses a heap data structure in order to recall the points and deal with them. More precisely, they develop a heap data structure, called *adjustable navigation pile*, of size $O(b)$ bits, i.e., $O(b/\log n)$ cells. Their algorithm uses three navigation piles, one to find and extract the points in each slab, and the other two to construct the upper-hull of the points in the slab and to prune it. The algorithm outputs CH(S) in $O(n^2/b + n \log b)$ time using $O(b/\log n)$ cells of workspace.

The underlying adjustable navigation pile is very versatile, and it can also be used to obtain time-space trade-offs for the sorting problem and for computing a triangulation of a planar point set [AEK13,KMvR$^+$17]. Due to the time-space product lower bound of $\Omega(n^2)$ for sorting n elements [Bea91], the achieved trade-offs for sorting and computing convex hull are asymptotically optimal. In Chapter 3, we will use this algorithm as a black box in order to compute the *farthest site Voronoi diagram* for S.

2.2 Voronoi Diagrams and Delaunay Triangulations

The *Voronoi diagram* is a versatile geometric structure that is closely linked to another important structure, the so-called *Delaunay triangulation* [BCvKO08]. In this chapter we introduce the Voronoi diagrams and the Delaunay triangulation in the plane.

Definition. Let $S = \{p_1, \ldots, p_n\}$ be a set of $n \geq 3$ point-*sites* in the plane. We assume general position, meaning that no three sites of S lie on a common line and no four sites of S lie on a common circle.

A *triangulation* $\mathcal{T}$ for S is a maximal plane straight-line graph whose vertex set is S. More precisely, $\mathcal{T}$ has as many edges as possible such that no edge between two vertices can be added to $\mathcal{T}$ without destroying the planarity, i.e., any edge that is not in $\mathcal{T}$ intersects one of the existing edges in its relative interior; see Figure 2.4.

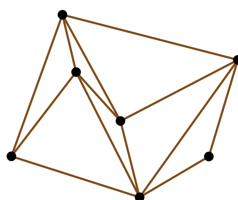

Figure 2.4: An arbitrary triangulation of a planar set of sites in general position.

A triangulation of S is called *Delaunay triangulation*, DT(S), if and only if the circumcircle of any triangle t in DT(S) does not contain any sites of S in its interior; see Figure 2.5. It is well known that DT(S) always exists, and it is uniquely defined under our general position assumptions.

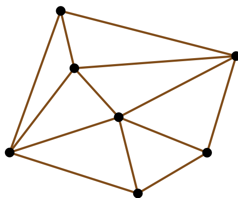

Figure 2.5: The Delaunay triangulation of a planar set of sites in general position.

The (*nearest site*) *Voronoi diagram* for S is obtained by classifying the points in the plane according to their nearest site in S. For each site $p \in S$, the open set of points in $\mathbb{R}^2$ with p as their unique nearest site in S is called the *Voronoi cell* of p. For any two sites $p, q \in S$, the *bisector* $B(p, q)$ of p and q is the line containing all the points in the plane that are equidistant to p and q. The *Voronoi edge* for p, q consists of all the points in the plane with p and q as their only two nearest sites. If it exists, the Voronoi edge for p and q is a subset of $B(p, q)$. Our general position assumption, and the fact that $n \geq 3$, guarantee that each Voronoi edge is an open line segment or a halfline.

Voronoi vertices are the points in the plane that have exactly three nearest sites in S. Again by our general position assumption, every point in $\mathbb{R}^2$ is either a Voronoi vertex or lies on a Voronoi edge or in a Voronoi cell. The Voronoi vertices and the Voronoi edges form the set of vertices and edges of a plane graph whose faces are the Voronoi cells. This graph is called the nearest site Voronoi diagram for S, and it is denoted by $\text{NVD}(S)$; see Figure 2.6. It has $O(n)$ vertices, $O(n)$ edges, and n cells [AKL13, BCvKO08].

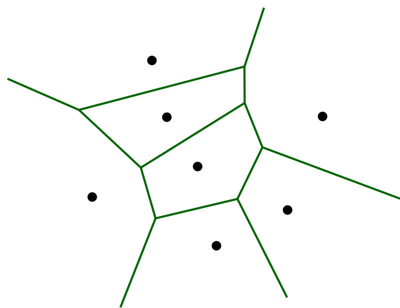

Figure 2.6: The nearest site Voronoi diagram for a planar set of sites.

The *farthest site Voronoi diagram* for S, $\text{FVD}(S)$, is defined analogously. Farthest Voronoi cells, edges, and vertices are obtained by replacing the term "nearest site" by the term "farthest site" in the respective definitions. Again, the farthest Voronoi vertices and edges constitute the vertices and edges of a plane graph, called $\text{FVD}(S)$. As before, it has $O(n)$ vertices and $O(n)$ edges. However, unlike in $\text{NVD}(S)$, in $\text{FVD}(S)$ it is not necessarily the case that all sites in S have a corresponding cell in $\text{FVD}(S)$. Indeed, the sites with non-empty farthest Voronoi cells are exactly the sites on the convex hull of S. Furthermore, all cells in $\text{FVD}(S)$ are unbounded. Hence, $\text{FVD}(S)$, considered as a plane graph, is a tree; see Figure 2.7 [AKL13, BCvKO08].

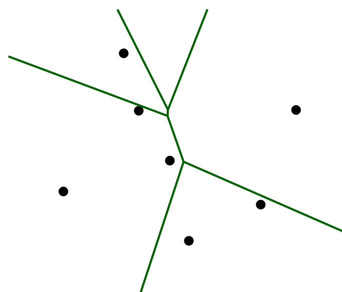

Figure 2.7: The farthest site Voronoi diagram for a planar set of sites.

The Delaunay triangulations and the nearest site Voronoi diagrams are dual structures, i.e., there is an edge connecting two sites in $\mathrm{DT}(S)$ if and only if the cells of that two sites in $\mathrm{NVD}(S)$ share an edge; see Figure 2.8. Due to this property, the algorithms for computing one of these two structures are usually adaptable to compute the other one. There are several algorithms to compute the Delaunay triangulation and the nearest site Voronoi diagram of a set of n planar points in $O(n \log n)$ time provided that $O(n)$ cells of workspace are available.

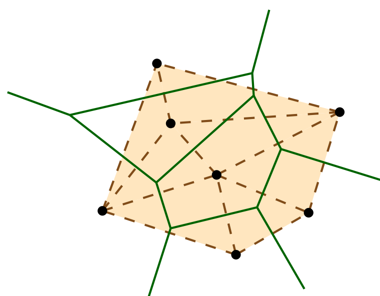

Figure 2.8: Illustration for the duality of DT and NVD for a given set of points.

For $\mathrm{FVD}(S)$, it is also well known that, if we forego memory constraints, we can compute $\mathrm{FVD}(S)$ in $O(n \log n)$ time using $O(n)$ cells of workspace. Note that computing a Voronoi diagram or computing a triangulation means to report each of its edges, exactly once, in an arbitrary order [AKL13, BCvKO08].

Constant Workspace. For computing the Delaunay triangulation and the nearest site Voronoi diagram of S using $O(1)$ cells of workspace, Asano *et al.* [AMRW11] presented an $O(n^2)$-time algorithm. Their algorithm for $\mathrm{NVD}(S)$ simply processes the sites in S sequentially and for each $p \in S$ outputs all the edges of the cell of p in counterclockwise order. To do this, they define a subroutine that computes a single edge of the cell of p in a given direction. This is done in $O(n)$ time by one scan of the input and computing the bisector between any sites in S and p. While scanning the input, their algorithm maintains the *closest* bisector to p in the given direction, which at the end of the scan defines one edge of the cell of p; see Figure 2.9.

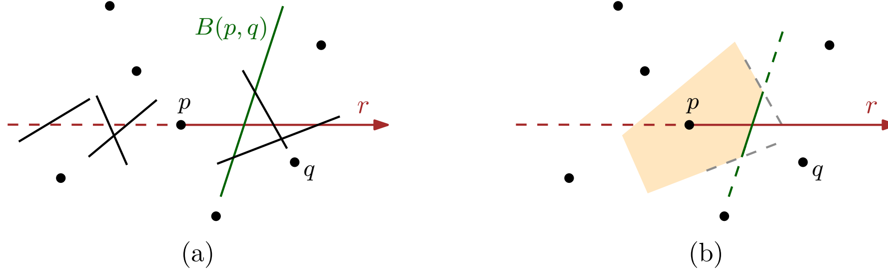

Figure 2.9: An intermediate step of the algorithm in [AMRW11]. For $p \in S$ and a direction r, (a) the bisectors between p and all the points in S. (b) Since $B(p, q)$ intersects r closest to p, a portion of $B(p, q)$ is an edge of the cell of p in NVD(S).

By repeating the same procedure, they find the next counterclockwise edges of the cell of p. Since finding each edge takes $O(n)$ time, and since NVD(S) has $O(n)$ edges, it takes $O(n^2)$ time to compute NVD(S) using $O(1)$ cells of workspace. For DT(S), in a dual way, they provide a subroutine which outputs one edge incident to a point $p \in S$. Then, they find the next counterclockwise edges incident to p and repeat it for all the points in S. This results to an $O(n^2)$ running time algorithm that uses $O(1)$ cells of workspace. They also use this algorithm as a black box in order to compute the *Euclidean minimum spanning tree* for S; see Chapter 2.4.

Time-Space Trade-offs. Korman *et al.* [KMvR+17] gave a randomized time-space trade-off for computing NVD(S) that runs in $O((n^2/s) \log s + n \log s \log^* s)$ expected time, provided that s cells of workspace may be used. The algorithm is based on a space-efficient implementation of the Clarkson-Shor random sampling technique [CS89] that makes it possible to divide the problem into $O(s)$ subproblems with $O(n/s)$ sites each. All subproblems can then be handled simultaneously with the constant workspace method of Asano *et al.* [AMRW11], resulting in the desired running time.

We have developed a deterministic algorithm that provides a better time-space trade-off [BKM+18b]. Our algorithm computes NVD(S) as well as FVD(S) in $O((n^2/s) \log s)$ time, using s cells of workspace, thus saving a $\log^* s$ factor for large values of s compared to the result of Korman *et al.* [KMvR+15]. We believe that our method is simpler and more flexible than the previous methods. The main idea is to obtain NVD(S) and FVD(S) by processing S in *batches* of s sites each, using a special procedure to handle sites whose Voronoi cells have a large number of edges. See Chapter 3 for the full description of the algorithm.

2.3 Higher-Order Voronoi Diagrams

A natural extension of nearest site and farthest site Voronoi diagrams is captured as *higher-order Voronoi diagrams*. In the following we see the definition and some basic properties of higher-order Voronoi diagrams.

Definition. Let $S = \{p_1, \ldots, p_n\}$ be a set of $n \geq 3$ point-*sites* in the plane. We assume general position, meaning that no three sites of S lie on a common line and no four sites of S lie on a common circle. Let $x \in \mathbb{R}^2$ be a point in the plane. The *distance order* for x is the sequence of sites in S ordered according to their distance from x, from closest to farthest. By our general position assumption, there are at most three sites in S with the same distance to x.

For $k \in \{1, \ldots, n-1\}$, a subset Q of S is called a *k-subset* if $|Q| = k$. The Voronoi diagram *of order* k for S is obtained by classifying the points in the plane into cells, edges, and vertices according to the k-subset whose sites achieve the k smallest distances in the distance order of the points. We denote the Voronoi diagram of order k for S by $\mathrm{VD}^k(S)$; see Figure 2.10. Observe the two cases of $\mathrm{VD}^1(S) = \mathrm{NVD}(S)$ and $\mathrm{VD}^{n-1}(S) = \mathrm{FVD}(S)$. We use HVD as the abbreviation of higher-order Voronoi diagram.

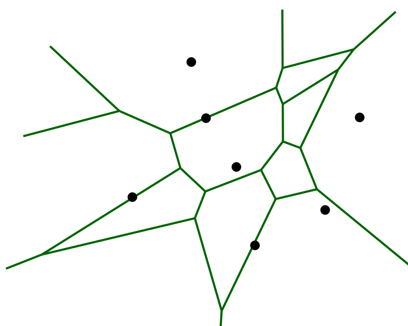

Figure 2.10: The Voronoi diagram of order 2 for a planar set of sites in general position.

We call a cell C of $\mathrm{VD}^k(S)$ a *k-cell*, and we represent it as the k-subset whose sites are the first k sites in the distance order of all the points in C. Let $Q \subset S$ be the *k-subset* representing a k-cell, then we denote this k-cell by $C^k(Q)$; see Figure 2.11. For simplicity, the cell of $p \in S$ in $\mathrm{NVD}(S)$ and $\mathrm{FVD}(S)$ is denoted by $C^1(p)$ and $C^{n-1}(p)$, respectively. As mentioned before, the boundary of a cell $C^k(Q)$ is denoted by $\partial C^k(Q)$.

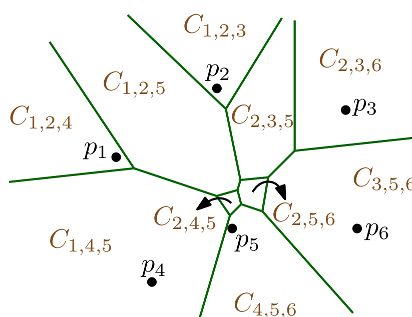

Figure 2.11: The diagram $\mathrm{VD}^k(S)$ for $k = 3$ and $S = \{p_1, \ldots, p_6\}$ and the k-cells are shown. As an example, the k-cell $C_{4,5,6}$ contains all the points in the plane whose first k sites in their distance order form the k-subset $\{p_4, p_5, p_6\}$.

Similarly, we call a vertex v of $\mathrm{VD}^k(S)$ a *k-vertex*. It is known that there exists a disk D_v with center v such that $|\partial D_v \cap S| = 3$ and $|\mathring{D}_v \cap S| \in \{k-2, k-1\}$, where ∂D_v is the boundary and $\mathring{D}_v$ is the interior of D_v. We call v an *old* vertex if $|\mathring{D}_v \cap S| = k-2$, and a *new* vertex if $|\mathring{D}_v \cap S| = k-1$; see Figure 2.12. We represent v by the set $D_v \cap S$, marking the sites on ∂D_v.

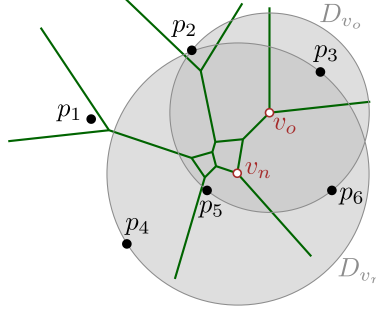

Figure 2.12: The diagram $\mathrm{VD}^k(S)$ for $k = 3$ and $S = \{p_1, \dots, p_6\}$. The interior of the disk D_{v_n} with center v_n contains $k-1$ sites $\{p_5, p_6\}$, so the k-vertex v_n is new. The interior of the disk D_{v_o} with center v_o contains $k-2$ sites $\{p_3\}$, so the k-vertex v_o is old.

Finally, the edges of $\mathrm{VD}^k(S)$ are called *k-edges*. A k-edge e^k is represented by $k+3$ sites of S: the $k-1$ sites closest to e^k, the two sites that come next in the distance order for the points on e^k and are equidistant to e^k, and one more site for each endpoint of e^k, to define the corresponding k-vertices. For each endpoint v of e^k, there are two cases: if v is an old vertex, the third site defining v is among the $k-1$ sites closest to e^k, and if v is a new vertex, the third site is not among those $k-1$ sites; see Figure 2.13.

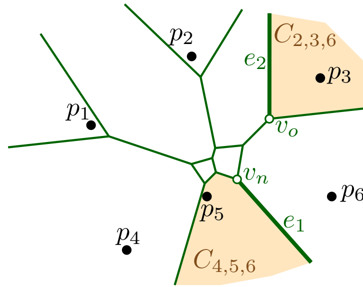

Figure 2.13: The diagram $\mathrm{VD}^k(S)$ for $k = 3$ and $S = \{p_1, \dots, p_6\}$. The k-cell $C_{4,5,6}$ corresponds to the k-subset $\{p_4, p_5, p_6\}$. The k-edge e_1 is represented by the set $\{p_5, p_6\}$ (the $k-1$ sites closest to e_1), the two sites p_3 and p_4 that are equidistant to e_1, and the site p_2 that defines the k-vertex v_n. Since v_n is a new k-vertex, the site p_2 is not among the $k-1$ closest sites to e_1. The k-cell $C_{2,3,6}$ corresponds to the k-subset $\{p_2, p_3, p_6\}$. The k-edge e_2 is represented by the set $\{p_2, p_3\}$ (the $k-1$ sites closest to e_2), the two sites p_5 and p_6 that are equidistant to e_2, and the site p_2 that defines the k-vertex v_o. Since v_o is an old k-vertex, the site p_2 is among the $k-1$ closest sites to e_2.

We actually represent each k-edge with two directed *half-edges*, such that the k-half-edges are oriented in opposing directions, and such that each k-half-edge is *associated* with the k-cell to its left. The order of the endpoints in the representation of e^k encodes the direction of each of the k-half-edges of e^k.[3] The k-half-edge is directed from the *tail* vertex to the *head* vertex.

It is known that the Voronoi diagram of order k for a set of size n is a plane graph of complexity $O(k\,(n-k))$ [AKL13]. In the following we present some other properties of higher-order Voronoi diagrams that we will use in our algorithm in Chapter 3. See for example the paper by Lee [Lee82] and the book by Aurenhammer *et al.* [AKL13] for further details.

Property (I) Let $Q_1, Q_2 \subset S$ be two k-subsets such that the k-cells $C^k(Q_1)$ and $C^k(Q_2)$ are both non-empty and are adjacent (i.e., they share a k-edge e). Then, the set $Q = Q_1 \cup Q_2$ is a $(k+1)$-subset, and $C^{k+1}(Q)$ is a non-empty $(k+1)$-cell; see Figure 2.14 for an illustration.

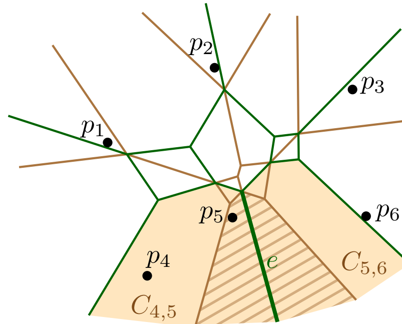

Figure 2.14: For the set $S = \{p_1, \ldots, p_6\}$ and $k = 2$, the two diagrams $\mathrm{VD}^k(S)$ (green) and $\mathrm{VD}^{k+1}(S)$ (beige) are shown. The k-cells $C_{4,5} = C^k(\{p_4, p_5\})$ and $C_{5,6} = C^k(\{p_5, p_6\})$ are non-empty, and they share the k-edge e on their boundaries. The $(k+1)$-subset $Q = \{p_4, p_5\} \cup \{p_5, p_6\} = \{p_4, p_5, p_6\}$ corresponds to a non-empty $(k+1)$-cell (shown hashed) which contains e in its interior.

Property (II) Let $Q \subset S$ be a $(k+1)$-subset such that $C^{k+1}(Q)$ is non-empty. Then, the part of $\mathrm{VD}^k(S)$ restricted to $C^{k+1}(Q)$ is identical to (i.e., has the same vertices and edges as) the part of $\mathrm{FVD}(Q)$ restricted to $C^{k+1}(Q)$. Furthermore, the edges of $\mathrm{FVD}(Q)$ in $C^{k+1}(Q)$ do not intersect the boundary, but their endpoints either lie in the interior of $C^{k+1}(Q)$ or coincide with vertices of $\partial C^{k+1}(Q)$. Hence, for every $(k+1)$-cell C, the number of k-edges in C lies between 1 and $O(k+1)$[4], and these edges form a tree; see Figure 2.15.

[3] Note that for higher order Voronoi diagrams, we denote both a k-half-edge or a k-edge by e^k. This is for sake of simplicity.

[4] More precisely, the number of k-edges in C is bounded by the number of edges in the farthest site Voronoi diagram of the set Q of $k+1$ sites, which is at most $2k-1$ edges [BCvKO08].

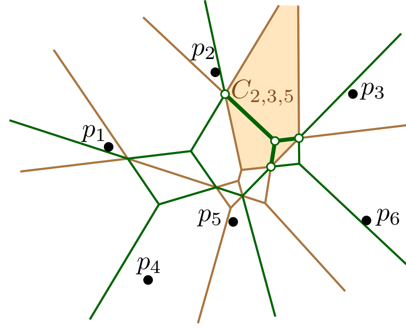

Figure 2.15: For the set $S = \{p_1, \ldots, p_6\}$ and $k = 2$, the two diagrams $\mathrm{VD}^k(S)$ (green) and $\mathrm{VD}^{k+1}(S)$ (beige) are shown. The $(k+1)$-cell $C_{2,3,5} = C^{k+1}(\{p_2, p_3, p_5\})$ is filled with beige. Inside $C_{2,3,5}$, the edges of $\mathrm{VD}^k(S)$ are identical to the edges of $\mathrm{FVD}(\{p_2, p_3, p_5\})$. These edges meet the boundary of $C_{2,3,5}$ only on the vertices of $\partial C_{2,3,5}$.

Property (III) If v is an old k-vertex, then it is also a new $(k-1)$-vertex, and if v is a new k-vertex, then it is also an old $(k+1)$-vertex. In particular, every vertex appears in exactly two Voronoi diagrams of consecutive order; see Figure 2.16. Note that all 1-vertices are new, and all $(n-1)$-vertices are old.

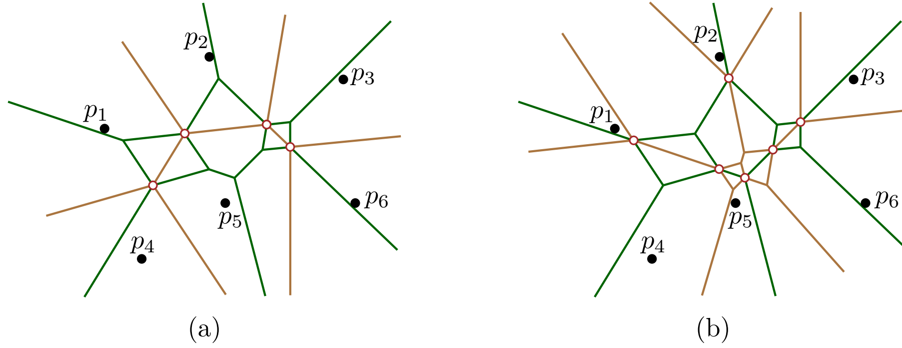

Figure 2.16: The diagram $\mathrm{VD}^k(S)$ (green) for $k = 2$ and $S = \{p_1, \ldots, p_6\}$. (a) The diagram $\mathrm{VD}^{k-1}(S)$ is shown in beige. The empty vertices of $\mathrm{VD}^k(S)$ are old k-vertices, and they also appear in $\mathrm{VD}^{k-1}(S)$ as new $(k-1)$-vertices. (b) The diagram $\mathrm{VD}^{k+1}(S)$ is shown in beige. The empty vertices of $\mathrm{VD}^k(S)$ are new k-vertices, and they also appear in $\mathrm{VD}^{k+1}(S)$ as old $(k+1)$-vertices. Every vertex of $\mathrm{VD}^k(S)$ appears in exactly one of $\mathrm{VD}^{k-1}(S)$ or $\mathrm{VD}^{k+1}(S)$.

For any given $K \in \{1, \ldots, n-1\}$, the *family of higher-order Voronoi diagrams of order* 1 *to* K *for* S contains all $\mathrm{VD}^k(S)$ for $k = 1, \ldots, K$, and it is denoted by $\mathrm{HVD}^{1:K}(S)$. Computing a single diagram $\mathrm{VD}^k(S)$, means to report each of its k-edges, exactly once, in some arbitrary order. Furthermore, computing $\mathrm{HVD}^{1:K}(S)$ is equivalent to compute all $\mathrm{VD}^k(S)$ for $k = 1, \ldots, K$, in increasing order of k.

For computing a single Voronoi diagram of order k, the best known randomized algorithm takes $O\big(n \log n + nk\, 2^{O(\log^* k)}\big)$ time and $O(nk)$ space [Ram99], while the best known deterministic algorithm takes $O\big(n \log n + nk \log k\big)$ time and $O(nk)$ space [Cha00, CT16].[5] For any given $K \in \{1, \ldots, n-1\}$, the family of higher-order Voronoi diagrams of order 1 to K can be computed in $O(nK^2 + n \log n)$ deterministic time using $O\big(K^2(n-K)\big)$ space [AGSS89, Lee82].

Time-space Trade-offs. For computing a specific higher-order Voronoi diagram, without first finding the diagrams of lower order, there are several efficient algorithms in the classic setting, when $\Omega(n)$ cells of workspace are available [CT16, AdBMS98, Ram99]. It would be interesting to extend any of them to obtain a general trade-off, or even an algorithm for constant workspace. However, as of now, for the whole range of k and s, we are not aware of any time-space trade-off. For $s \in O(1)$, one can compute a single Voronoi diagram of order k in $O(n^4)$ time using the naive algorithm that considers the whole arrangement which is obtained by a well-known geometric transformation of the set of n point-sites to a set of planes in three dimensions [CE87].

In this arrangement, computing the Voronoi diagram of order k is equivalent to computing the vertical projection of the k-level of the arrangement. To do this, we consider each vertical line through one of the $O(n^3)$ intersection points of the planes in the arrangement. On a vertical line l through an intersection point x, we count the number of planes that intersect l above the point x. If the point x is the intersection of the kth plane with either the $(k-1)^{\text{th}}$ or $(k+1)^{\text{th}}$, then x is clearly on the k-level of the arrangement. This takes a total of $O(n^4)$ time using $O(1)$ cells of workspace. It is not known to us, if an additional workspace of $O(s)$ cells can be exploited in this naive approach to achieve a better running time, e.g., $O(n^4/s)$.

For the family of higher-order Voronoi diagrams, the situation is different since we can compute each higher-order Voronoi diagram using the previously computed diagrams of the lower levels. Based on this idea, we use our $\text{NVD}(S)$ algorithm as a building block to compute $\text{HVD}^{1:K}(S)$ for a given parameter $K \in O(\sqrt{s})$. Our algorithm reports all the edges of $\text{VD}^1(S), \ldots, \text{VD}^K(S)$ in $O\big(\frac{n^2K^5}{s}(\log s + K\, 2^{O(\log^* K)})\big)$ expected time or in $O\big(\frac{n^2K^5}{s}(\log s + K \log K)\big)$ deterministic time, using a workspace of size $O(s)$ cells.

In this algorithm, to compute edges of a Voronoi diagram of order k, we use edges of the diagram of order $k-1$. However, this needs to be coordinated carefully in order to prevent edges from being reported multiple times, and to not exceed the space budget. Since the edges of all the diagrams are computed simultaneously, or in other words in a pipelined fashion, at any step of our algorithm, we need $O(k)$ cells of workspace to store the state of the sub-algorithm which computes the Voronoi diagram of order k. This makes a total of $O(K^2)$ cells of workspace. Hence, due to the space limit, K has to be in $O(\sqrt{s})$. We will explain the details of this algorithm in Chapter 3.

[5]This algorithm uses the rather involved dynamic planar convex hull structure of Brodal and Jacob [BJ02]. If the reader prefers a more elementary method, we can substitute it by the slightly slower, but much simpler, previous result by the same authors. The running time then becomes $O(n \log n + nk \log k \log\log k)$ [BJ00, CT16].

2.4 Euclidean Minimum Spanning Trees

Another fundamental geometric structure that has been studied in the limited workspace model is the *Euclidean minimum spanning trees* of a planar set of point-sites. In this chapter we define the Euclidean minimum spanning tree and we briefly mention some algorithms for computing it.

Definition. Let $G = (V, E)$ be a graph with vertex set V and edge set E. A *spanning tree* of G is defined as a subgraph G' of G with the vertex set V and the edge set $E' \subset E$, such that G' is a tree, i.e., it has no cycle, and there is a path between every two vertices of G'. It is clear that, if G is not connected, then it has no spanning tree, otherwise it might have several spanning trees and at least one; see Figure 2.17a.

Let $S = \{p_1, \ldots, p_n\}$ be a set of n sites in the plane. Let G_S be the complete and weighted graph with vertex set S, where the edges are weighted with the Euclidean distance of their endpoints. A spanning tree of G_S that has the minimum weight, among all possible spanning trees of G_S, is called the Euclidean minimum spanning tree of S, and is denoted by $\text{EMST}(S)$. Under a general position assumption, i.e., no three sites in S lie on a common line, no four points in S lie on a common circle, and the distance between every pair of points in S is distinct, $\text{EMST}(S)$ is unique; see Figure 2.17b.

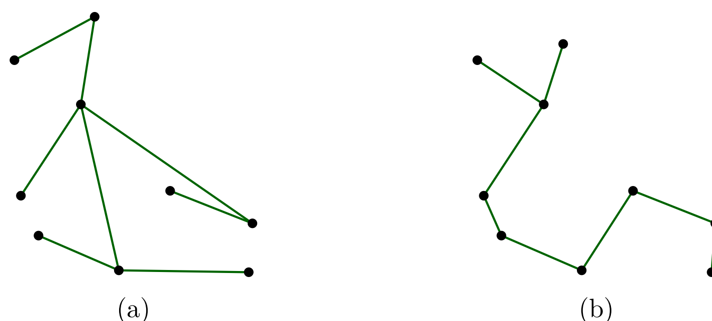

Figure 2.17: For a planar set of sites S, (a) a spanning tree of the complete graph with S as the vertex set. (b) the Euclidean minimum spanning tree of S.

Several classic algorithms are known that compute $\text{EMST}(S)$ in $O(n \log n)$ time using $O(n)$ cells of workspace [BCvKO08], where computing $\text{EMST}(S)$ means to report each of its edges exactly once in an arbitrary order. Recall the classic algorithm by Kruskal to find $\text{EMST}(S)$ [CLRS09]: we start with an empty forest T, and we consider the edges of G_S one by one, by increasing weight. In each step, we insert the current edge e into T if and only if there is no path between its endpoints in T; see Figure 2.18. In the end, T is $\text{EMST}(S)$. Using a *union-find data structure*, we can keep track of vertices of T in each component, and thus, we can determine if there is a path in T between the two vertices or not. Since this data structure provides constant time operations, the time needed for adding the edges of G_S to T is dominated by the time needed for sorting the edges of G_S by their weight.

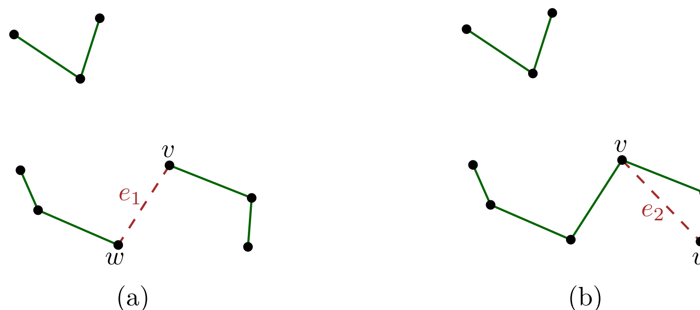

Figure 2.18: An intermediate state of the Kruskal forest T, for a planar set of points. (a) In the current step the algorithm adds e_1 to T, since there is no path between v and w. (b) In the following step, the algorithm checks e_2, and since v and u are connected it does not add e_2 to T.

It is well-known that EMST(S) is a subgraph of DT(S); see Figure 2.19. Thus, it suffices to consider only the edges of DT(S) instead of the edges of the complete graph G_S. Using this, Kruskal's algorithm needs to consider $O(n)$ edges of DT(S) and runs in $O(n \log n)$ time, when $O(n)$ cells of workspace are available [BCvKO08].

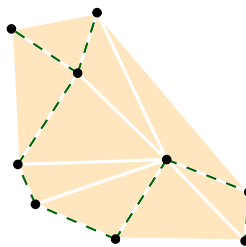

Figure 2.19: An illustration for the fact that EMST(S) is a subgraph of DT(S), for a set of points S. The dashed edges belong to EMST(S).

Constant Workspace. The task of computing EMST(S) was among the first problems to be considered in the limited workspace model. Asano *et al.* [AMRW11] provided an algorithm that reports the edges of EMST(S) using $O(n^3)$ time and $O(1)$ cells of workspace. This is still the fastest algorithm for the problem when only $O(1)$ cells of workspace are available.

Since EMST(S) is a subgraph of DT(S), Asano *et al.* use their constant workspace Delaunay triangulation algorithm (see Chapter 2.2) in order to produce edges of DT(S), one by one. Every time that a new Delaunay edge e is detected, the algorithm pauses the computation of DT(S), and using a subroutine, it determines if e is in EMST(S).

More precisely, they use the *bottleneck shortest path* property of minimum spanning trees: a Delaunay edge $e = pq$ is not in EMST(S) if and only if DT(S) has a path between p and q consisting only of the edges with length less than $|pq|$ [Epp00]. Let

$DT_{<e}$ be the subgraph of $DT(S)$ with the edges of length less than $|e|$. Therefore, to decide if e is in $EMST(S)$ or not, it must be determined whether there is a path between p and q in $DT_{<e}$. Since any subgraph of $DT(S)$ is planar, if such paths exist, then one of those paths together with e forms a face. Thus, by walking from p along the boundary of that face, q must be encountered. If not, such a path does not exist. In the former case e is reported as an edge of $EMST(S)$; see Figure 2.20.

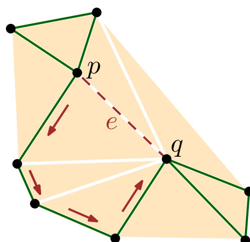

Figure 2.20: The subgraph $DT_{<e}$ for a planar set of sites S and an edge $e = pq$ of $DT(S)$. To decide if e belongs to $EMST(S)$, we check the existence of a path from p to q in $DT_{<e}$. This is done by walking along the face of $DT_{<e}$ that is intersected by e starting from p.

To perform each step of the walk on $DT_{<e}$, they use one iteration of the Delaunay triangulation algorithm. This subroutine receives the current edge pq, and it computes only the next clockwise Delaunay edge incident to p in $O(n)$ time using $O(1)$ cells of workspace. The algorithm may repeat this procedure for edges incident to p, until it finds the first edge with length less than $|e|$. This concludes one step of the walk. Each walk generates a subset of the edges of $DT(S)$, and each edge is generated at most twice. Thus, $O(n^2)$ is the total running time to decide if an edge e of $DT(S)$ is in $EMST(S)$. The required space is constant.

Since $DT(S)$ has $O(n)$ edges, and since it takes $O(n^2)$ time to decide membership in $EMST(S)$, the total time to find all the edges of $EMST(S)$ is $O(n^3)$. Furthermore, the overhead for computing all edges of $DT(S)$ is $O(n^2)$, which is negligible compared to the remainder of the algorithm. The space bound is immediate. With a slight modification in the algorithm, we can report the edges of EMST by increasing length: we repeatedly compute the whole diagram $DT(S)$, and each time we find the shortest edge $e \in DT(S)$ whose membership in EMST has not been checked. After checking if $e \in$ EMST, we again compute the whole $DT(S)$ to find the next shortest edge. This causes an overhead of $O(n^3)$ which is just a constant factor in the total running time of the algorithm.

Time-Space Trade-offs. Using the immediate idea of the constant workspace algorithm by Asano *et al.* [AMRW11] and combining it with our time-space trade-off for computing $NVD(S)$ (more precisely, a modified version of the algorithm that can compute the Delaunay triangulation) [BKM+18b] does not give us a better trade-off than the running time of $O((n^3/s)\log s)$ and space usage of $O(s)$ for computing $EMST(S)$. This bound is clearly not optimal when $s = O(n)$ cells of workspace are available.

Based on the idea of combining the two algorithms and by applying some new techniques, we develop a time-space trade-off that provides a smooth transition between the $O(n^3)$-time algorithm with constant cells of workspace [AMRW11] and the $O(n \log n)$-time algorithm using a workspace of $O(n)$ cells [BCvKO08]. Our algorithm computes EMST(S) in $O((n^3/s^2) \log s)$ time using $O(s)$ cells of workspace [BBM18].

This algorithm uses the workspace in two different ways: we check s edges in parallel for membership in EMST(S). Furthermore, we introduce *s-nets* which are compact representations of planar graphs in $O(s)$ cells of workspace. Applying s-nets, one can speed up Kruskal's MST algorithm on S by better exploiting the additional workspace.[6] The s-net structure seems to be of independent interest as it provides a compact way to represent planar graphs that could be utilized by other algorithms that deal with such graphs. We will explain the details of this algorithm in Chapter 4.

2.5 Relative Neighborhood Graphs

The *relative neighborhood graph* is an interesting geometric structure which is related to both Euclidean minimum spanning tree and the Delaunay triangulation for a set of point-sites in the plane. In this chapter, first, we define the relative neighborhood graph, and then, we illustrate the relation between these three graphs. We also explain who to exploit this property.

Definition. Let $S = \{p_1, \ldots, p_n\}$ be a set of n point-sites in the plane. For two sites $u, v \in S$, we define the *lens* of u and v as the intersection of the disk centered at u and passing through v with the disk centered at v and passing through u. The lens of u and v is called *empty* if and only if it contains no sites of $S \setminus \{v, v\}$. In other words, the two sites u and v has the *empty lens* property, if there is no site $w \in S \setminus \{u, v\}$ such that both $|uw|$ and $|vw|$ are shorter than $|uv|$, where $|uv|$ denotes the Euclidean distance between u and v; see Figure 2.21.

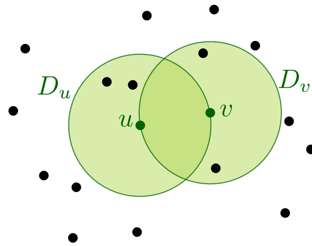

Figure 2.21: A set of sites S and two sites u and v in S. The disks D_u and D_v have radius $|uv|$ and are centered at u and v, respectively. The two sites u and v satisfy the empty lens property since $D_u \cap D_v$ is empty of other sites of S.

[6] Although the spirit of the algorithm is the same, in [BBM18], the walks are performed in the *relative neighborhood graph* of S instead of DT(S). This is critical, since DT(S) is not of bounded degree.

The *relative neighborhood graph* of S is the undirected graph with vertex set S obtained by connecting two sites $u, v \in S$ with an edge if and only if the lens of u and v is empty [Tou80]. This graph is denoted by $\mathrm{RNG}(S)$. One can show that a plane embedding of $\mathrm{RNG}(S)$ is obtained by drawing the edges as straight line segments between the corresponding sites in S; see Figure 2.22.

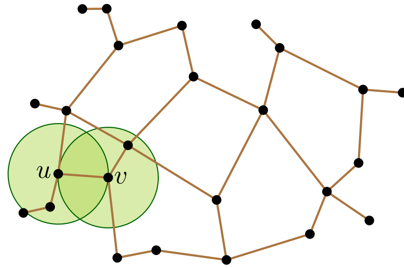

Figure 2.22: The $\mathrm{RNG}(S)$ and an edge uv in $\mathrm{RNG}(S)$. The lens of u and v is empty.

By definition, $\mathrm{RNG}(S)$ is a subgraph of $\mathrm{DT}(S)$.[7] Furthermore, each vertex in $\mathrm{RNG}(S)$ has at most six neighbors, so $\mathrm{RNG}(S)$ is a bounded degree graph. The graph $\mathrm{RNG}(S)$ has $O(n)$ edges. We will denote the number of those edges by m. Given S, we can compute $\mathrm{RNG}(S)$, meaning that we can report each of its edges exactly once, in $O(n \log n)$ time using $O(n)$ cells of workspace [Tou80, JT92, MM17].

It is well-known that $\mathrm{EMST}(S)$ is a subgraph of $\mathrm{RNG}(S)$ [BCvKO08]. In particular, this implies that $\mathrm{RNG}(S)$ is connected; see Figure 2.23. Additionally, this indicates that we can compute $\mathrm{EMST}(S)$ in Kruskal's algorithm using edges of $\mathrm{RNG}(S)$. However, since both $\mathrm{RNG}(S)$ and $\mathrm{DT}(S)$ have $O(n)$ edges, the running time of Kruskal's algorithm in the classic setting (when $O(n)$ cells of workspace are available) does not improve if we use edges of $\mathrm{RNG}(S)$ instead of $\mathrm{DT}(S)$. Nevertheless, unlike $\mathrm{DT}(S)$, the degree of vertices in $\mathrm{RNG}(S)$ is bounded, and thus, using $\mathrm{RNG}(S)$ for finding $\mathrm{EMST}(S)$ in the limited workspace model has some advantages over using $\mathrm{DT}(S)$.

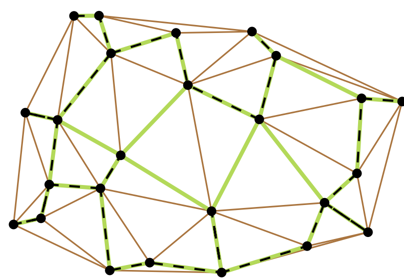

Figure 2.23: An illustration of the fact $\mathrm{EMST}(S) \subseteq \mathrm{RNG}(S) \subseteq \mathrm{DT}(S)$. The dashed black edges belong to $\mathrm{EMST}(S)$ and are a subset of the green edges which represent $\mathrm{RNG}(S)$. All these edges forms a subset of the edges of the underlying graph $\mathrm{DT}(S)$.

[7] If $e = uv$ is in $\mathrm{EMST}(S)$, then the lens of u and v is empty, which means that the smallest circle passing through both u and v is also empty of other sites of s. Thus, e belongs to $\mathrm{DT}(S)$.

In order to ensure a unique EMST for the given set S, while determining EMST(S) using RNG(S), we assume the general position mentioned in Chapter 2.4, by which the edge lengths in RNG(S) are pairwise distinct. We define $E_R = e_1, \ldots, e_m$ to be the sorted sequence of RNG(S) edges, in increasing order of length. For $i \in \{1, \ldots, m\}$, we define RNG$_i$ to be the subgraph of RNG(S) with vertex set S and edge set $\{e_1, \ldots, e_{i-1}\}$. Thus, to check if e_i belongs to EMST(S), the algorithm by Kruskal checks if the endpoints of e_i lie on the same component of RNG$_i$ or not; see Figure 2.24.

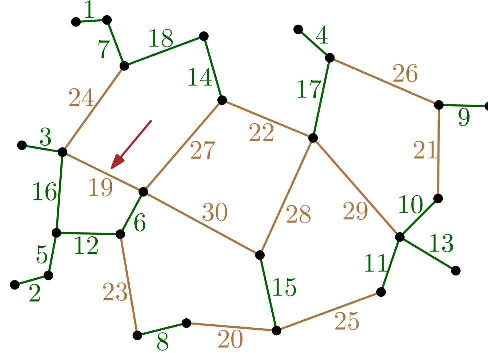

Figure 2.24: The RNG for a set of sites S. The labels represent the indices of the edges in the sorted sequence E_R. The subgraph RNG$_{19}$ is shown in black. The edge e_{19} does not belong to EMST(S) since its endpoints lie on the same component of RNG$_{19}$.

We represent each edge $e_i \in E_R$ by two directed *half-edges*. The two half-edges are oriented in opposite directions such that the face incident to a half-edge lies on its left. We call the endpoints of a half-edge the *head* and the *tail* such that the half-edge is directed from the tail endpoint to the head endpoint. Furthermore, directed half-edges will be denoted as $\overrightarrow{e}$ and undirected edges as e; see Figure 2.25.[8]

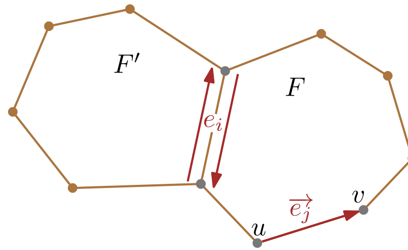

Figure 2.25: A schematic drawing of the faces F, F' of RNG(S). The two half-edges that correspond to the edge e_i are oriented such that the face incident to each of lies on its respective left. The sites v and u are the head and the tail endpoints of the half-edge $\overrightarrow{e_j} = \overrightarrow{uv}$, respectively.

[8]For simplicity we have used similar notations for a k-half-edge and a k-edge in higher order Voronoi diagrams. However, in the content of relative neighborhood graphs, we need to distinguish a half-edge $\overrightarrow{e}$ from an edge e.

Using the concept of half-edges, we define the *face-cycle* in a planar graph for each face in the graph or the outer face of each connected components of the graph. More precisely, for $i \in \{1, \ldots, m\}$, a *face-cycle* in RNG_i is the circular sequence of consecutive half-edges such that (i) they bound either a face in RNG_i or an outer face in a connected component of RNG_i; and (ii) every two consecutive half-edges e and e' in a face-cycle share an endpoint which is the head vertex of e and the tail vertex of e'.

The definition implies that all the half-edges in a face-cycle are oriented in the same direction and the face (or outer face) incident to the half-edges lies on their left. Note that every half-edge lies on only one face-cycle; however, every site of S might be on several face-cycles; see Figure 2.26. Furthermore, the *partial relative neighborhood graph* RNG_i can be represented as a collection of face-cycles.

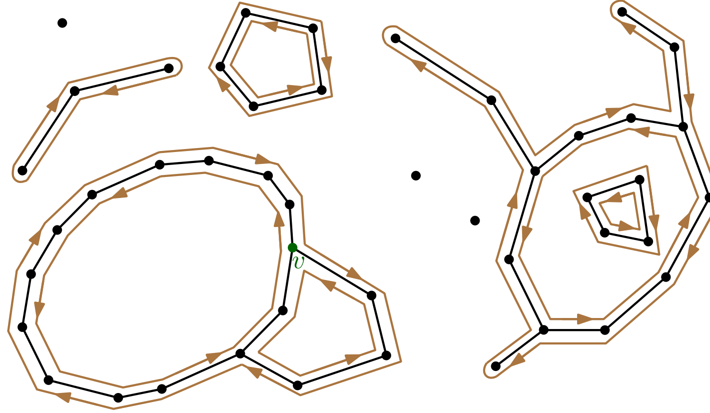

Figure 2.26: A schematic drawing of RNG_i for a planar set S of sites is in black. The face-cycles of RNG_i are in beige. All the half-edges of each face-cycle are directed according to the arrows on the corresponding cycle. The site $v \in S$ is on three face-cycles of RNG_i. Each of the six half-edges incident to v are only on one face-cycle.

Let $j \geq i \geq 1$. We define the *predecessor* and the *successor* in RNG_i for a half-edge $\overrightarrow{e_j}$ with head w as follows: the predecessor $\overrightarrow{p_j}$ of $\overrightarrow{e_j}$ is the half-edge in RNG_i which has w as its head and is the first half-edge encountered in a counterclockwise sweep from $\overrightarrow{e_j}$ around w. The successor $\overrightarrow{s_j}$ of $\overrightarrow{e_j}$ is the half-edge in RNG_i which has w as its tail and is the first half-edge encountered in a clockwise sweep from $\overrightarrow{e_j}$ around w; see Figure 2.27 for an illustration. Note that, if there is no edge incident to w in RNG_i, we set both p_j and s_j to *Null*.

Let $i > j \geq 1$. For the half-edge $\overrightarrow{e_j}$ in RNG_i that lies on the face-cycle F, we define the *next* edge on F as follows: the next edge of $\overrightarrow{e_j}$ is the half-edge on F whose tail endpoint is the head endpoint of $\overrightarrow{e_j}$. Note that the next edge for a half-edge $\overrightarrow{e_j}$ is defined with respect to each diagram RNG_i, where $i > j$ and thus $\overrightarrow{e_j} \in \mathrm{RNG}_i$. However, the predecessor and successor of $\overrightarrow{e_j}$ are defined with respect to each diagram RNG_i, where $i \leq j$, meaning that $\overrightarrow{e_j} \notin \mathrm{RNG}_i$.

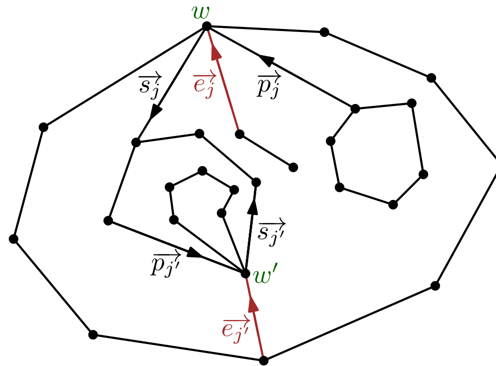

Figure 2.27: For $j, j' \geq i \geq 1$, a schematic drawing of RNG_i and the half-edges $\overrightarrow{e_j}$ with the head w and $\overrightarrow{e_{j'}}$ with the head w' are shown. For $\overrightarrow{e_j}$ the predecessor and the successor are $\overrightarrow{p_j}$ and $\overrightarrow{s_j}$, respectively. For $\overrightarrow{e_{j'}}$ the predecessor and the successor are $\overrightarrow{p_{j'}}$ and $\overrightarrow{s_{j'}}$, respectively.

Time-Space Trade-offs. Using a similar technique as the one for computing $\mathrm{NVD}(S)$, we obtain a same time-space trade-off for computing $\mathrm{RNG}(S)$, i.e., an s-workspace algorithm with $O\big((n^2/s)\log s\big)$ running time. Furthermore, since $\mathrm{EMST}(S)$ is a subgraph of $\mathrm{RNG}(S)$, we use our $\mathrm{RNG}(S)$ algorithm as a blackbox in Kruskal's algorithm to establish a time-space trade-off for computing $\mathrm{EMST}(S)$. As we have mentioned in the previous chapter, the resulting s-workspace algorithm computes $\mathrm{EMST}(S)$ in $O\big((n^3/s^2)\log s\big)$ time. In Chapter 4, we will describe these algorithms in detail.

Voronoi Diagrams

CHAPTER 3

In this chapter, we describe a deterministic s-workspace algorithm for computing NVD and FVD for a given planar set S of n point-sites that runs in $O((n^2/s)\log s)$ time. Moreover, we generalize our s-workspace algorithm such that, for any given $K \in O(\sqrt{s})$, it computes $\mathrm{HVD}^{1:K}(S)$ in $O(\frac{n^2K^5}{s}(\log s + K\,2^{O(\log^* K)}))$ total expected time or in $O(\frac{n^2K^5}{s}(\log s + K\log K))$ total deterministic time.

In Chapter 3.1, we restate the approach by Asano *et al.* [AMRW11] for computing $\mathrm{NVD}(S)$ with a constant size workspace, and we show that the same technique can be used to compute $\mathrm{FVD}(S)$. In Chapter 3.2, we introduce a new time-space trade-off for computing $\mathrm{NVD}(S)$ and $\mathrm{FVD}(S)$. Finally, in Chapter 3.3, we use the s-workspace algorithm from Chapter 3.2 as a building block in a new pipelined algorithm in order to compute $\mathrm{HVD}^{1:K}(S)$.

3.1 A Constant Workspace Algorithm for NVD and FVD

For the given planar set $S = \{p_1, \dots, p_n\}$ of n point-sites, stored in a read only array, our task is to compute $\mathrm{NVD}(S)$ and $\mathrm{FVD}(S)$ using only a constant number of cells of workspace. We summarize the properties of $\mathrm{FVD}(S)$ that are relevant to our algorithms in the following two facts. More details can be found, e.g., in the book by Aurenhammer, Klein, and Lee [AKL13]. See Figure 3.1 for an illustration.

Fact 3.1. *Let S be a set of n point-sites in the plane in general position, and let $p \in S$. The cell $C^{n-1}(p)$ is not empty if and only if p lies on $\partial\,\mathrm{CH}(S)$. In this case, the farthest Voronoi cell of p is unbounded. Furthermore, if $r, l \in S$ are the two adjacent sites of p on $\partial\,\mathrm{CH}(S)$, then the cells $C^{n-1}(p)$ and $C^{n-1}(r)$ share an unbounded edge which is a subset of the bisector $B(p, r)$, and analogously the cells $C^{n-1}(p)$ and $C^{n-1}(l)$ share an unbounded edge which is a subset of the bisector $B(p, l)$.*

Fact 3.2. *Let S be a set of n point-sites in the plane in general position. Let $l, p, r \in S$ be three consecutive sites on $\partial\,\mathrm{CH}(S)$ in counterclockwise order, and let c be the intersection of the bisectors $B(p, l)$ and $B(p, r)$. Then, the ray from p toward c intersects $\partial C^{n-1}(p)$ (not necessarily at c).*

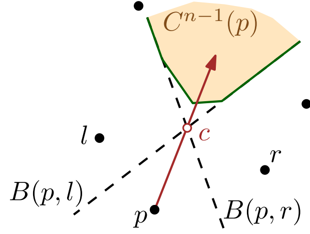

Figure 3.1: An illustration of Facts 3.1 and 3.2: The sites $l, p, r \in S$ are consecutive on $\partial\,\mathrm{CH}(S)$. The boundary $\partial C^{n-1}(p)$ contains a subset of $B(p,l)$ and of $B(p,r)$, Fact 3.1. The ray from p toward $c = B(p,l) \cap B(p,r)$ intersects $\partial C^{n-1}(p)$, Fact 3.2.

In the following two lemmas, we show how to find a single edge of a given cell of NVD(S) or of FVD(S). Then, in Theorem 3.5, we repeatedly use the procedure of finding a single edge of a cell in order to find, first, all the edges of the same cell, and then the edges of all the other cells of NVD(S) and FVD(S).

Lemma 3.3. *Let S be a set of n point-sites in the plane in general position. Suppose that S is given in a read-only array. For any $p \in S$, in $O(n)$ time and using $O(1)$ cells of workspace, we can determine whether $C^{n-1}(p)$ is not empty. If so, we can also find a ray that intersects $\partial C^{n-1}(p)$.*

Proof. By Fact 3.1, it suffices to check whether p lies inside CH(S). This can be done with simple *gift-wrapping*: pick an arbitrary site $q \in S \setminus \{p\}$. Scan through S and find the sites p_{cw} and p_{ccw} in S which make, respectively, the largest clockwise angle and the largest counterclockwise angle with the ray pq, such that both angles are at most π. Both p_{cw} and p_{ccw} are easily obtained in $O(n)$ time using $O(1)$ cells of workspace. If the cone $p_{\mathrm{cw}}pp_{\mathrm{ccw}}$ that contains q has an opening angle larger than π, then p is inside CH(S) and consequently $C^{n-1}(p)$ is empty; see Figure 3.2a. Otherwise, p is on $\partial\,\mathrm{CH}(S)$, with p_{cw} and p_{ccw} as its two neighbors; see Figure 3.2b. By Fact 3.2, the ray from p through $B(p, p_{\mathrm{cw}}) \cap B(p, p_{\mathrm{ccw}})$ intersects $\partial C^{n-1}(p)$. □

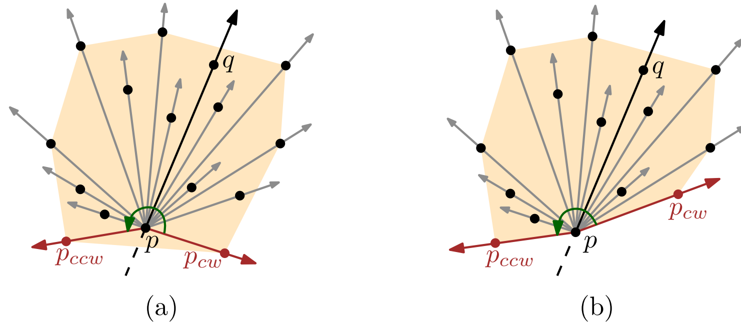

Figure 3.2: An illustration of Lemma 3.3. (a) The counterclockwise angle $p_{cw}pp_{ccw}$ is larger than π and p is inside CH(S). (b) The counterclockwise angle $p_{cw}pp_{ccw}$ is smaller than π and p is on $\partial\,\mathrm{CH}(S)$.

Lemma 3.4. *Let S be a set of n point-sites in the plane in general position that is stored in a read-only array. Suppose we are given a site $p \in S$ and a ray γ that emanates from p and intersects $\partial C^1(p)$. Then, we can report an edge e of $C^1(p)$ that intersects γ in $O(n)$ time, using $O(1)$ cells of workspace. An analogous statement holds for* FVD(S).

Proof. Among all bisectors $B(p, p')$, for $p' \in S \setminus \{p\}$, we find a bisector $B^* = B(p, p^*)$ that intersects γ closest to p.[1] We can find B^* by scanning the sites of S and maintaining a closest bisector in each step. By the definition of NVD(S), it follows that the edge e is a subset of B^*. To find the portion of B^* that forms a Voronoi edge in NVD(S), we do a second scan of S. For each $p' \in S \setminus \{p, p^*\}$, we check where $B(p, p')$ intersects B^*. Each such intersection cuts a piece from B^* that cannot appear in NVD(S), namely the part of B^* that is closer to p' than to p. After scanning all the sites of S, the remaining portion of B^* is exactly e. Since the current piece of B^* in each step is connected, we need to maintain only the at most two endpoints. Overall, we can find an edge e of $C^1(p)$ that intersects γ in $O(n)$ time using $O(1)$ cells of workspace; see Figures 3.3a and 3.3b for an illustration.

The procedure for FVD(S) is analogous, but we take $B^* = B(p, p^*)$ to be the bisector intersecting γ farthest from p, and we cut from B^* the pieces that are closer to p than to any other site in $S \setminus \{p, p^*\}$. Note that here γ is the given ray that intersects $\partial C^{n-1}(p)$. We conclude that an edge e of $C^{n-1}(p)$ that intersects γ can be found in $O(n)$ time using $O(1)$ cells of workspace; see Figures 3.3a and 3.3c. □

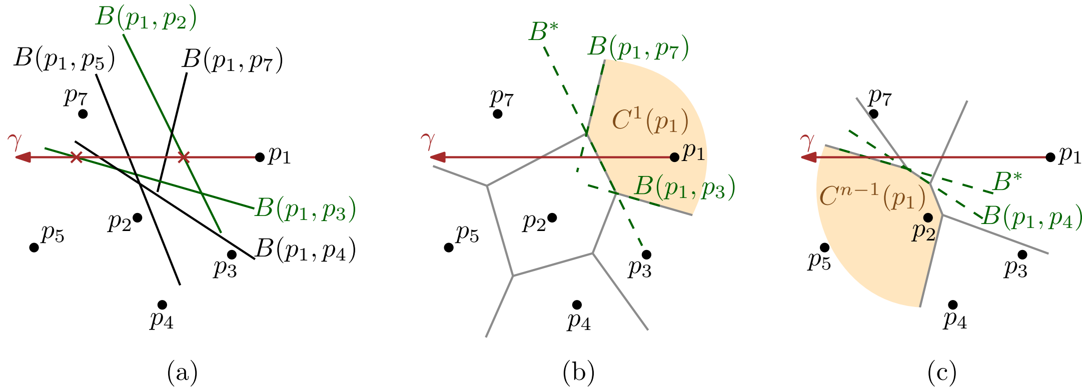

Figure 3.3: An illustration of Lemma 3.4. (a) Among the bisectors between p_1 and all the other sites, the bisector $B(p_1, p_2)$ intersects the ray γ closest to p_1 and the bisector $B(p_1, p_3)$ intersects the ray γ farthest from p_1. (b) A portion of $B^* = B(p_1, p_2)$ is an edge of $C^1(p_1)$ and its endpoints are defined by the bisectors $B(p_1, p_3)$ and $B(p_1, p_7)$. (c) A portion of $B^* = B(p_1, p_3)$ is an edge of $C^{n-1}(p_1)$ and its endpoint is defined by the bisector $B(p_1, p_4)$.

[1] If γ happens to intersect a vertex of $C^1(p)$, there are two such bisectors. Otherwise, B^* is unique.

Theorem 3.5. *Suppose we are given a planar set of n point-sites $S = \{p_1, \dots, p_n\}$ in general position, stored in a read-only array. We can find all the edges of* NVD(S) *in* $O(n^2)$ *time, using* $O(1)$ *cells of workspace. The same holds for* FVD(S).

Proof. First, we restate the strategy that was proposed by Asano *et al.* [AMRW11] for NVD(S), and then we show how to adapt it for FVD(S).

We go through the sites in S. In step i, we process $p_i \in S$ to detect all edges of $C^1(p_i)$. For this, we need a ray γ to apply Lemma 3.4. We choose γ as the ray from p_i to an arbitrary site of $S \setminus \{p_i\}$. This ensures that γ intersects $\partial C^1(p_i)$. Now we use Lemma 3.4 to find an edge e of $C^1(p_i)$ that intersects γ. We consider the ray γ' from p_i through the endpoint of e that lies to the left of γ (if it exists), and we apply Lemma 3.4 to find the adjacent edge e' of e in $C^1(p_i)$.[2] The ray γ' hits both e and e', so we perform a symbolic perturbation to γ' so that only e' is hit. We repeat this procedure to find further edges of $C^1(p_i)$, in counterclockwise direction; see Figure 3.4a. This continues until we return to e or until we find an unbounded edge of $C^1(p_i)$. In the latter case, we start again from the right endpoint of e (if it exists), and we find the remaining edges of $C^1(p_i)$ in clockwise direction.

Since each edge of NVD(S) is incident to two Voronoi cells, this process will detect each edge twice. To avoid repetitions, whenever we find an edge e of $C^1(p_i)$ with $e \subseteq B(p_i, p_j)$, we report e if and only if $i < j$. Since NVD(S) has $O(n)$ edges, and reporting one edge takes $O(n)$ time and $O(1)$ cells of workspace, the result follows.

For FVD(S), the procedure is almost the same. However, when we process each site $p_i \in S$, we first check if $C^{n-1}(p_i)$ is non-empty, using Lemma 3.3. If so, the algorithm from the lemma also gives us a ray γ that intersects $\partial C^{n-1}(p_i)$. From here, we proceed exactly as for NVD(S) to find the remaining edges of $C^{n-1}(p_i)$; see Figure 3.4b. □

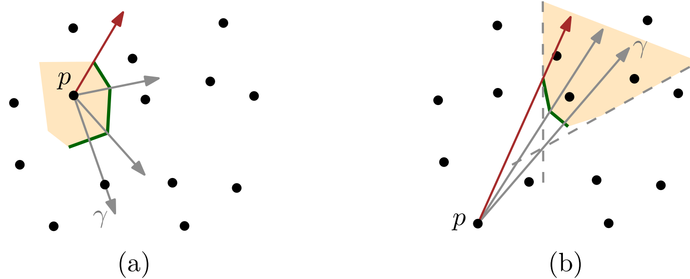

Figure 3.4: For $p \in S$, the starting ray γ emanates from p to (a) another arbitrary site of S to find an edge of $C^1(p)$, (b) the intersection of the bisectors between p and its neighbors on CH(S) to find an edge of $C^{n-1}(p)$. The next rays, in both (a) and (b), emanate from p through the left endpoint of the last computed edge.

[2]The bisector that defines the left endpoint of e is also the bisector that is spanned by e'. Thus, the first scan of the input in Lemma 3.4, for finding the line spanned by e', is not strictly necessary. However, since we must scan the input anyway to determine the endpoint of e', we chose to present the algorithm as doing two scans. This keeps the presentation more uniform at the expense of only a constant factor in the running time. The same comment also applies to our algorithms in Chapters 3.2 and 3.3

3.2 A Time-Space Trade-Off for NVD and FVD

In this chapter, we adapt the algorithm from Chapter 3.1 to a time-space trade-off for computing NVD(S) and FVD(S). Suppose we have $O(s)$ cells of workspace at our disposal, for some $s \in \{1, \ldots, n\}$. As before, we are given a planar set of n point-sites $S = \{p_1, \ldots, p_n\}$ in general position in a read-only array, and we would like to report all edges of NVD(S) or FVD(S) as quickly as possible. While the algorithm from Chapter 3.1 needs two passes over the input to find a single edge of the Voronoi diagram, the idea now is to exploit the additional workspace in order to find s edges of the Voronoi diagram in parallel, using two passes.

3.2.1 Finding a Single Edge of Some Cells

In the following lemma, we show how to find simultaneously a single edge for s different cells of NVD(S) or of FVD(S), using $O(s)$ cells of workspace.

Lemma 3.6. *Suppose we are given a set $V = \{v_1, \ldots, v_s\}$ of s sites in S, and for each $i = 1, \ldots, s$, a ray γ_i emanating from v_i such that γ_i intersects $\partial C^1(v_i)$. Then, we can report, for each $i = 1, \ldots, s$, an edge e_i of $C^1(v_i)$ that intersects γ_i, in $O(n \log s)$ total time, using $O(s)$ cells of workspace. An analogous statement holds for* FVD(S).

Proof. The algorithm has two phases. In the first phase, for $i = 1, \ldots, s$, we find the bisector B_i^* that contains e_i, and in the second phase, for $i = 1, \ldots, s$, we find e_i, i.e., the portion of B_i^* that is in NVD(S).

The first phase proceeds as follows: we group S into *batches* $Q_1, Q_2, \ldots, Q_{n/s}$ of s consecutive sites (according to the order in the input array).[3] We start with considering the union of the set V and the first batch Q_1, and we compute NVD($V \cup Q_1$). Since $|V \cup Q_1| \leq 2s$, this takes $O(s \log s)$ time using $O(s)$ cells of workspace. Now, for $i = 1, \ldots, s$, we find the edge e_i' of NVD($V \cup Q_1$) that intersects γ_i closest to v_i, and we store the bisector B_i' that contains e_i'. This can be done in total time $O(|V \cup Q_1|)$, since each ray originates in a unique Voronoi cell, and since we can simply traverse the whole diagram NVD($V \cup Q_1$) to find the intersection points; see Figure 3.5a.

Then, for $j = 2, \ldots, n/s$, we again compute NVD($V \cup Q_j$). For $i = 1, \ldots, s$, we find the edge in NVD($V \cup Q_j$) that intersects γ_i closest to v_i, in total time $O(|V \cup Q_j|)$. We update B_i' to the bisector that contains this edge if and only if its intersection with γ_i is closer to v_i than for the current B_i'. We claim that after all batches $Q_1, \ldots, Q_{n/s}$ have been scanned, B_i' is the desired bisector B_i^*; see Figure 3.5. To see this, let $B_i^* = B(v_i, p)$, for a site $p \in S \setminus \{v_i\}$. Then, for the batch Q_j with $p \in Q_j$, the Voronoi diagram NVD($V \cup Q_j$) contains an edge on B_i^*. This is because $V \cup Q_j \subset S$ and NVD(S) has an edge of B_i^*. Furthermore, by definition, the bisector between no other site and v_i intersects γ_i closer to v_i than B_i^*.

[3] For ease of explanation, we assume that $n/s \in \mathbb{N}$. If $n/s \notin \mathbb{N}$ we group S into $\lceil n/s \rceil$ batches. Each such a batch has exactly s elements, except for the last batch $Q_{\lceil n/s \rceil}$ which may have fewer than s elements.

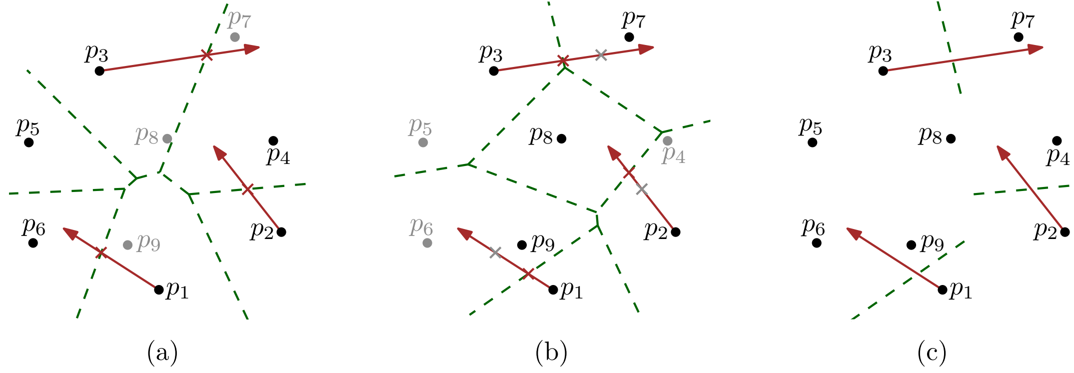

Figure 3.5: For $V = \{p_1, p_2, p_3\} \subset S = \{p_1, \ldots, p_9\}$ and the given starting rays from the sites in V: (a) the intersection of the rays with $\mathrm{NVD}(V \cup Q_1)$ where $Q_1 = \{p_4, p_5, p_6\}$; (b) the intersection of the rays with $\mathrm{NVD}(V \cup Q_2)$ where $Q_2 = \{p_7, p_8, p_9\}$; (c) the resulting closest bisectors to the sites in V after scanning all the batches of sites in S.

In the second phase, we again group S into batches $Q_1, \ldots, Q_{n/s}$ of size s. We again compute $\mathrm{NVD}(V \cup Q_1)$. For $i = 1, \ldots, s$, we find the portion of B_i^* inside the cell of v_i in $\mathrm{NVD}(V \cup Q_1)$, and we store it in e_i; see Figure 3.6a. Then, for $j = 2, \ldots, n/s$, we compute $\mathrm{NVD}(V \cup Q_j)$, and for $i = 1, \ldots, s$, we update the endpoints of e_i to the intersection of the current e_i and the cell of v_i in $\mathrm{NVD}(V \cup Q_j)$. This is done in total $O(|V \cup Q_j|)$ time by traversing the whole diagram. After processing Q_j, there is no site in $V \cup \bigcup_{m=1}^{j} Q_m$ that is closer to e_i than v_i, i.e., no site whose bisector with v_i cuts a bigger portion of e_i. Thus, at the end of the second phase, e_i is the edge of $C^1(v_i)$ that intersects γ_i; see Figure 3.6. Due to the properties of the Voronoi diagram, throughout the algorithm, e_i is a connected subset of B_i^* (i.e., a ray or a line segment), and it can be described with $O(1)$ cells of workspace.

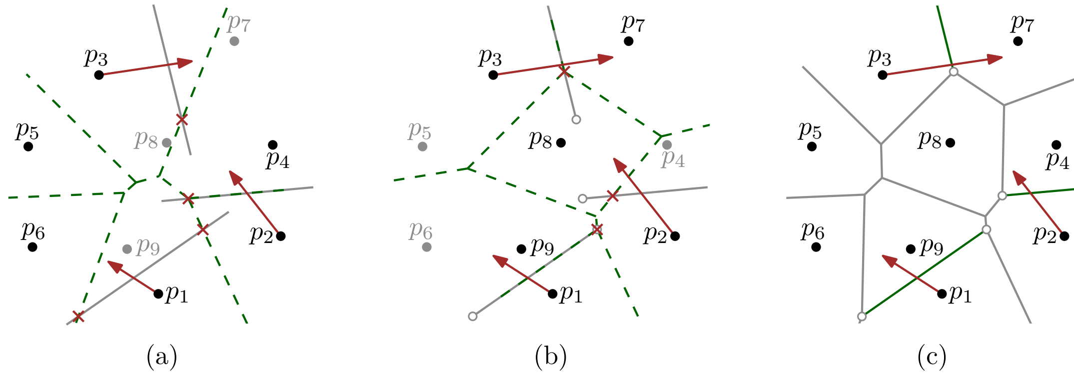

Figure 3.6: The endpoints of the bisectors obtained from the first phase in the cells of (a) $\mathrm{NVD}(V \cup \{p_4, p_5, p_6\})$; and (b) further in $\mathrm{NVD}(V \cup \{p_7, p_8, p_9\})$. (c) The edge on the cell of each site of V that intersects the corresponding ray.

In total, we construct $O(n/s)$ Voronoi diagrams, each with at most $2s$ sites. Since we have $O(s)$ cells of workspace available, it takes $O(s \log s)$ time to compute a single Voronoi diagram. Thus, the total running time is $O(n \log s)$. At each point in time, we have $O(s)$ sites in workspace and a constant amount of information for each site, including the Voronoi diagram of these sites, so the space bound is not exceeded. The proof for FVD(S) is analogous. □

3.2.2 Finding All the Edges According to Their Incident Cells

Now we describe our time-space trade-off algorithm. At each point in time, we have a set V of s sites in workspace. We use Lemma 3.6 to produce a new edge for each site in V. Once all edges for a site $v \in V$ have been found, we discard v from V and replace it with a new site from S (we say that v has been processed completely). We stop this process as soon as all but fewer than s sites have been processed completely. At this point, we do not use Lemma 3.6 any longer. This is because Lemma 3.6 needs two passes of the input to find a single new edge for each site in V. Thus, if there is a cell with many edges, too many passes will be necessary; see Figure 3.7 for an example. To avoid this, we will need a different method for finding the edges of the remaining cells, see below. We call these remaining cells *big* and the other cells *small*. By definition, all small cells have $O(n/s)$ edges, but big cells may have a lot more edges (even though this does not have to be the case).

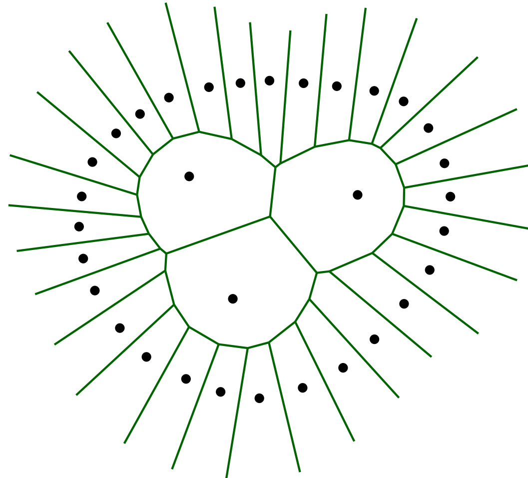

Figure 3.7: An example of an NVD that has three cells with too many edges, while all the other cells are of constant complexity, i.e., they have three or four edges.

In order to avoid doubly reporting edges, our algorithm is split into three *phases*. In the first phase, we process the whole input to identify the big cells (no edge is reported in this phase). The second phase scans the input again and reports all edges incident to at least one small cell. The third phase reports edges incident to two big cells.

First phase. The aim of this phase is to find the big cells. We describe how we use Lemma 3.6 in more detail. We scan all sites with non-empty Voronoi cells. For NVD(S), since all sites have a non-empty cell, we can scan them sequentially. The starting ray is constructed in the same way as in Theorem 3.5. For FVD(S), by Fact 3.2, we need to find the sites on the convex hull of S. For this, we use the algorithm of Darwish and Elmasry [DE14] that reports the sites on the convex hull of S in clockwise order in $O(n^2/s \log n + n \log s)$ time using $O(s)$ cells of workspace; see Chapter 2.1 for more details. We run the Darwish-Elmasry algorithm until s sites on the convex hull have been identified. Then, we suspend the convex hull computation and process those sites. Whenever more sites are needed, we simply resume the convex hull algorithm. Since the convex hull is reported in clockwise order, we know the two neighbors for each site on the convex hull, and using Fact 3.2, we can find a starting ray for each such a site.

At each point in time, our Voronoi algorithm has s sites from S with non-empty cells in memory. We apply Lemma 3.6 to compute one edge on the cell of each such site. After that, we iteratively update the rays of all sites in memory to find the next edge of each cell, as in Theorem 3.5. Whenever all edges of a cell have been found, we remove the corresponding site from memory, and we replace it with the next relevant site; see Figure 3.8. Since (1) the Voronoi diagram of S has $O(n)$ edges, (2) in each iteration we produce s edges, and (3) each edge is produced at most twice, it follows that after $O(n/s)$ iterations, fewer than s sites remain in memory. All other sites of S must have been processed.

Thus, after the first phase, we have identified all big cells (those that have not been processed fully). Since there are at most s of them, we can store the corresponding sites explicitly in a table $\mathcal{B}$. We sort those sites according to their indices, so that membership in $\mathcal{B}$ can be tested in $O(\log s)$ time.

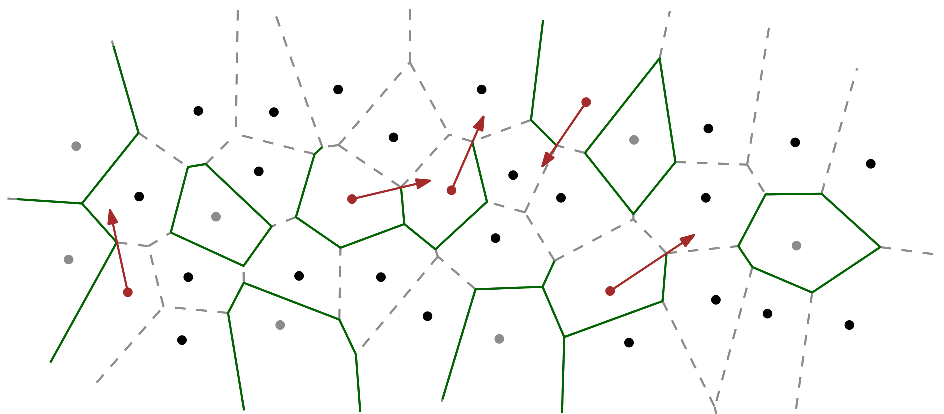

Figure 3.8: Illustration of the algorithm after nine iterations of Lemma 3.6 for a set S of $n = 35$ sites and workspace of size $O(\log n)$ cells. The green segments are the edges of NVD(S) that have already been found. The gray and the red sites represent the sites which have been fully processed and those which are currently in the workspace, respectively.

Second phase. The second phase is very similar to the first one.[4] Pick s sites to process; repeatedly use Lemma 3.6 to find edges for each site; once all edges of a site v have been found, replace v with the next site; continue until only big cells remain. The main difference now is that we report some Voronoi edges (making sure that every edge is reported exactly once). More precisely, suppose that we discover a Voronoi edge e while scanning the cell C_i of a site v_i, and that e is also incident to the cell C_j of the site v_j. Then, we report e if and only if one of the following conditions holds (see Figure 3.9):
(i) both C_i and C_j are small and $i < j$; or
(ii) C_i is small and C_j is big.

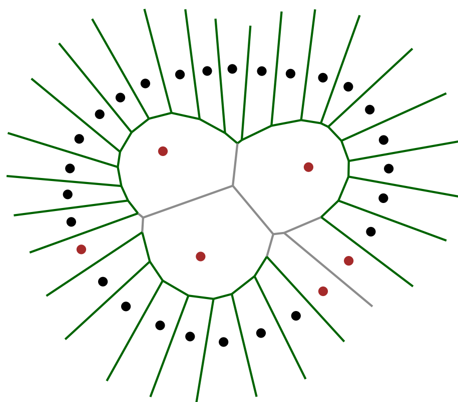

Figure 3.9: An illustration of the algorithm at the end of the second phase. The red sites correspond to the big cells. The edges between every two big cells have not been reported (gray edges). All the other edges have been reported exactly once (black edges).

Third phase. The purpose of the third phase is to report every Voronoi edge that is incident to two big cells. For this, we compute the Voronoi diagram of the sites of big cells, in $O(s \log s)$ time. Let $E_{\mathcal{B}}$ denote the set of its edges. The edges of $E_{\mathcal{B}}$ that are also present in the Voronoi diagram of S need to be reported (the edges may need to be truncated); see Figure 3.10.

In order to determine which edges of $E_{\mathcal{B}}$ remain in the diagram, we proceed similarly as in the second scan of Lemma 3.6: in each step, we compute the Voronoi diagram $\mathcal{V}$ of $\mathcal{B}$ and a batch of s sites from S. For each edge e of $E_{\mathcal{B}}$, we check whether e is cut off in $\mathcal{V}$. If so, we update the endpoints of e to the intersection of e and the cell for one of the sites defining e. After all edges have been checked, we continue with the next batch of s sites from S. After processing all the sites of S, the remaining $O(s)$ edges in $E_{\mathcal{B}}$ that have not become empty constitute all the edges of the Voronoi diagram of S that are incident to two big cells. In contrast to Lemma 3.6, we report $O(s)$ edges that are not necessarily incident to s different cells.

[4] Indeed, these two phases could be merged into one. However, as we will see below, it is not straightforward to do so for higher-order Voronoi diagrams. Thus, for consistency, we split the two phases even for $k = 1$ and $k = n - 1$.

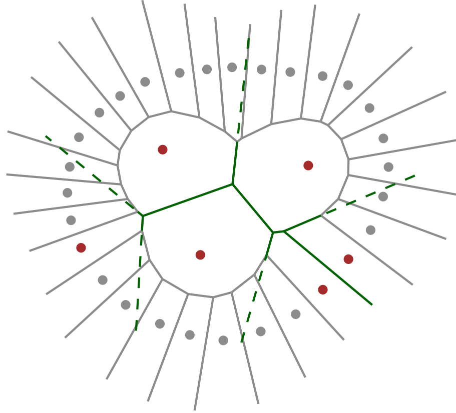

Figure 3.10: An illustration for the observation that the edges in $\mathrm{NVD}(S)$ which are between two big cells are a subset of the (possibly truncated) edges in $\mathrm{NVD}(\mathcal{B})$. The sites corresponding to the big cells are shown in red.

Theorem 3.7. *Let $S = \{p_1, \ldots, p_n\}$ be a planar set of n point-sites in general position, stored in a read-only array. Let s be a parameter in $\{1, \ldots, n\}$. We can report all edges of $\mathrm{NVD}(S)$ in $O\big((n^2/s)\log s\big)$ time, using $O(s)$ cells of workspace. An analogous result holds for $\mathrm{FVD}(S)$.*

Proof. Lemma 3.6 guarantees that the edges reported in the second phase are part of $\mathrm{NVD}(S)$. Also, conditions (i) and (ii) ensure that no edge is reported twice. Clearly, if an edge $e \in \mathrm{NVD}(S)$ is incident to two big cells, the same edge (possibly a superset) must be present in $\mathrm{NVD}(\mathcal{B})$. For the reverse inclusion, first note that since $\mathcal{B} \subset S$, an edge incident to two big cells that is not present in $\mathrm{NVD}(\mathcal{B})$ cannot be present in $\mathrm{NVD}(S)$. Furthermore, for each edge e of $\mathrm{NVD}(\mathcal{B})$, we consider all sites of S and remove only the portions of e that cannot be present in $\mathrm{NVD}(S)$.

Finally, we need to analyze the running time. The most expensive part of the algorithm lies in the $O(n/s)$ invocations of Lemma 3.6 during the first and the second phase. Other than that, creating the table $\mathcal{B}$ needs $O(s \log s)$ time, and we perform $O(n)$ lookups in $\mathcal{B}$, two for each edge of $\mathrm{NVD}(S)$. Each lookup needs $O(\log s)$ time, so $O(n \log s)$ time in total. The third phase does a single scan over the input, and it computes a Voronoi diagram for each batch of s sites, which totally takes $O(n \log s)$ time. Thus, the running time of the algorithm is $O\big((n^2/s)\log s\big)$.

At each point during the algorithm, we store only s sites that are currently being processed (along with a constant amount of information attached to each such site), the table $\mathcal{B}$ of at most s sites, the batch of s sites being processed (and the associated Voronoi diagram). All of this can be stored using $O(s)$ cells of workspace, as claimed.

For $\mathrm{FVD}(S)$, the approach is analogous. The only difference is that now we must also find the convex hull of S. With the algorithm of Darwish and Elmasry [DE14], this takes $O\big((n^2/s)\log s\big)$ time for $O(s)$ cells of workspace, so the asymptotic running time does not increase. □

3.3 A Time-Space Trade-Off for the Family of HVDs

We now consider the computation of higher-order Voronoi diagrams where we have $O(s)$ cells of workspace at our disposal, for some $s \in \{1, \ldots, n\}$. We are given an integer $K \in O(\sqrt{s})$, and we would like to report $\mathrm{HVD}^{1:K}(S)$, the family of higher-order Voronoi diagrams of order 1 to K for S. For this, we generalize our approach from Chapter 3.2, and we combine it with a recursive procedure. More precisely, for $k = 1, \ldots, K-1$, we compute the edges of $\mathrm{VD}^{k+1}(S)$ by using previously computed edges of $\mathrm{VD}^{k}(S)$. To make efficient use of the available memory, we perform the computation of the diagrams $\mathrm{VD}^1(S), \mathrm{VD}^2(S), \ldots, \mathrm{VD}^K(S)$ in a pipelined fashion, so that in each stage, the necessary edges of the previous Voronoi diagrams are at our disposal and the total memory usage remains $O(s)$ cells.

3.3.1 Relation Between Two Consecutive HVDs

To generate all $(k+1)$-half-edges using k-half-edges, for some $k = 1, \ldots, K$, our high-level idea is as follows: let e be a k-half-edge. By Property (II) of higher-order Voronoi diagrams (refer to Chapter 2.3), the k-half-edge e lies inside a $(k+1)$-cell C. We will see that we can use e as a starting ray to report all $(k+1)$-half-edges incident to C, similar to Lemma 3.6. However, if we repeat this procedure for every k-half-edge that lies inside a $(k+1)$-cell, again by Property (II) of higher-order Voronoi diagrams, we may report a $(k+1)$-half-edge $\Omega(k)$ times. This will lead to problems when we combine the procedures for computing the Voronoi diagrams of different orders. To avoid this, we do the following: we call a k-half-edge *relevant* if its head vertex lies on the boundary of the $(k+1)$-cell that contains it; see Figure 3.11 for an example.

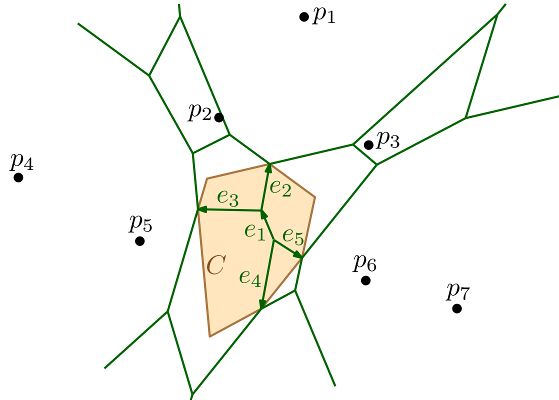

Figure 3.11: The diagram $\mathrm{VD}^k(S)$ for $k = 3$ and $S = \{p_1, \ldots, p_7\}$. The k-half-edges $e_1, \ldots, e_5$ lie in the $(k+1)$-cell C defined by the sites $\{p_2, p_3, p_5, p_6\}$. The head vertex of e_1 does not lie on ∂C. Hence, e_1 is not a relevant k-half-edge. On the other hand, the head vertices of e_2, e_3, e_4, e_5 lie on ∂C, and thus, they are all relevant. None of the opposite half-edges of $e_1, \ldots, e_5$ is a relevant k-half-edge.

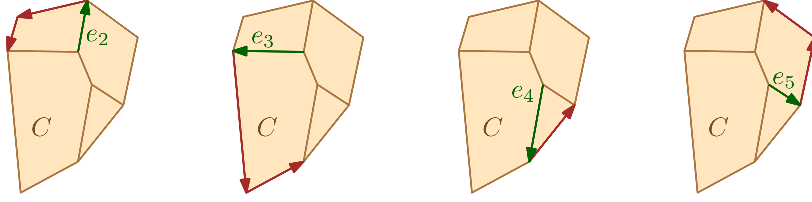

Figure 3.12: The relevant k-half-edges e_2, e_3, e_4, e_5 lying in the $(k+1)$-cell C and the $(k+1)$-interval assigned to each of these k-half-edges.

For each $(k+1)$-cell C, we partition the boundary of C into *intervals* of $(k+1)$-half-edges between two consecutive head vertices of relevant k-half-edges that lie inside C. Each such $(k+1)$-interval is assigned to the relevant k-half-edge of its clockwise endpoint; see Figure 3.12. Note that by the k-interval containing e we refer to the interval of k-half-edges (on the k-cell) that e belongs to, and this is different from the $(k+1)$-interval assigned to e.

Our algorithm goes through all k-half-edges. If the current k-half-edge e is not relevant, the algorithm does nothing. Otherwise, it reports the half-edges of the $(k+1)$-interval assigned to e. This ensures that every $(k+1)$-half-edge is reported exactly once. The following lemma describes an algorithm that takes s different k-half-edges. For each such k-half-edge e, the algorithm either determines that e is not relevant or finds the first edge of the $(k+1)$-interval assigned to e.

Lemma 3.8. *Suppose we are given s different k-half-edges $e_1^k, \ldots, e_s^k$ represented by the k-subsets $E_1, \ldots, E_s$ of S. There is an algorithm that, for $i = 1, \ldots, s$, either determines that e_i^k is not relevant, or finds e_i^{k+1}, the first $(k+1)$-half-edge of the $(k+1)$-interval assigned to e_i^k. The algorithm takes total expected time $O(n \log s + nk\, 2^{O(\log^* k)})$ or total deterministic time $O(n \log s + nk \log k)$ and uses $O(sk^2)$ cells of workspace.*

Proof. Our algorithm proceeds analogously to Lemma 3.6. First, we inspect all k-half-edges e_i^k. By Property (III) of higher-order Voronoi Diagrams, if the head vertex v of e_i^k is an old k-vertex, then v is not a vertex of $\mathrm{VD}^{k+1}(S)$, and it lies in the interior of a $(k+1)$-cell, so e_i^k is not relevant. Otherwise, v is a new k-vertex and an old $(k+1)$-vertex, so it appears on the boundary of a $(k+1)$-cell. In this case, we need to determine the first $(k+1)$-half-edge of the $(k+1)$-interval assigned to e_i^k. Let I be the set of all indices i such that e_i^k is relevant.

To determine the first $(k+1)$-half-edge of each $(k+1)$-interval, we process the sites in S in batches of size sk. In each iteration, we pick a new batch Q of sk sites. Then, we construct $\mathrm{VD}^{k+1}(\bigcup_{i \in I} E_i \cup Q)$ in $O(sk \log(sk) + sk^2\, 2^{O(\log^* k)})$ expected time [Ram99] or in $O(sk \log(sk) + sk^2 \log k)$ deterministic time [Cha00, CT16]. Note that $\bigcup_{i \in I} E_i \cup Q$ contains $O(sk)$ sites, so the diagram $\mathrm{VD}^{k+1}(\bigcup_{i \in I} E_i \cup Q)$ has complexity $O(sk^2)$. By construction, the head vertex of each e_i^k with $i \in I$ belongs to the resulting diagram, and we can find each head vertex in $O(\log(sk^2)) = O(\log(sk))$ time by using a point location structure [BCvKO08].

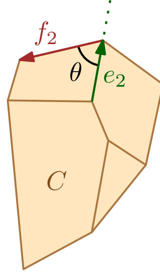

Figure 3.13: The $(k+1)$-half-edge f_2 is incident to the head vertex of e_2 and lies to the left of the directed line spanned by e_2. Among all such edges, f_2 makes the smallest angle θ with e_2.

Thus, we iterate over all batches, and for each e_i^k, we determine the edge f_i^{k+1} that appears in one of the resulting diagrams such that (i) f_i^{k+1} is incident to the head vertex of e_i^k; (ii) f_i^{k+1} is to the left of the directed line spanned by e_i^k; and (iii) among all such edges, f_i^{k+1} makes the smallest angle with e_i^k; see Figure 3.13. We need $O(n/sk)$ iterations to find f_i^{k+1} for all e_i^k.

Now, for each $i \in I$, the desired $(k+1)$-half-edge e_i^{k+1} is a subset of f_i^{k+1}. This is because, by Property (I) of higher-order Voronoi diagrams (refer to Chapter 2.3), there is one site which is different in the second $(k+1)$-cell incident to e_i^{k+1}, and this site exists in one of the batches. Thus, to find the other endpoint of e_i^{k+1}, as in Lemma 3.6, we perform a second scan over S in batches of sk sites. As before, for each batch Q, we construct $\mathrm{VD}^{k+1}(\bigcup_{i\in I} E_i \cup Q)$ and we check, for each $i \in I$, where f_i^{k+1} is cut-off in the new diagram. After scanning all the sites of S, we have the desired endpoint of e_i^{k+1}. This is because the endpoint of e_i^{k+1} is defined by one more site of S, and this site exists in one of the batches. Finally, we orient e_i^{k+1} such that the $(k+1)$-cell containing e_i^k lies to the left of it.

It follows that we can simultaneously process the given s edges of $\mathrm{VD}^k(S)$ in $O(n/sk)$ iterations. Each of these iterations takes $O(sk\log(sk) + sk^2\, 2^{O(\log^* k)})$ expected time or $O(sk\log(sk) + sk^2 \log k)$ deterministic time. Thus, we get $O(n\log s + nk\, 2^{O(\log^* k)})$ total expected time or $O(n\log s + nk\log k)$ total deterministic time. Note that the term $n\log(sk)$ is substituted by $n\log s$, since $n\log(sk) = n\log s + n\log k$, and since $n\log k$ is dominated by nk in the total running time. In each iteration of the algorithm, we have stored an intermediate Voronoi diagram of complexity $O(sk^2)$, in addition to some constant amount of information per k-half-edge e_i^k. Thus, the total space usage is $O(sk^2)$ cells of workspace. □

The algorithm from Lemma 3.8 is actually more general. If, instead of a k-half-edge e_i^k that lies inside a $(k+1)$-cell C, we have a $(k+1)$-half-edge e_i^{k+1} that lies on the boundary of C, the same method of processing S in batches of size sk allows us to find the next $(k+1)$-half-edge incident to C in counterclockwise order from e_i^{k+1}. Corollary 3.9 notes that these two kinds of edges can be handled simultaneously. See Figure 3.14 for an illustration.

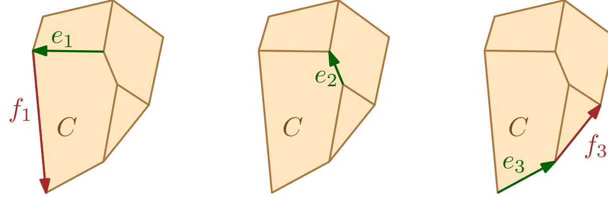

Figure 3.14: An illustration of Corollary 3.9. The $(k+1)$-half-edge f_1 is the first edge in the interval assigned to the relevant k-half-edges e_1. The k-half-edge e_2 is not relevant, thus f_2 is null. The $(k+1)$-half-edge f_3 is the counterclockwise successor of e_3 on C.

Corollary 3.9. *Let e_i denote either a k-half-edge or a $(k+1)$-half-edge. Suppose we are given s such half-edges $e_1, \dots, e_s$. Then, in total $O(n \log s + nk\, 2^{O(\log^* k)})$ expected time or $O(n \log s + nk \log k)$ deterministic time and using $O(sk^2)$ cells of workspace, we can find a sequence $f_1, \dots, f_s$ of $(k+1)$-half-edges such that, for each $i = 1, \dots, s$, we have*

1. *if e_i is a relevant k-half-edge, then f_i is the first $(k+1)$-half-edge of the $(k+1)$-interval assigned to e_i;*
2. *if e_i is a k-half-edge that is not relevant, then f_i is* null*;*
3. *if e_i is a $(k+1)$-half-edge adjacent to the $(k+1)$-cell C, then f_i is the counterclockwise successor of e_i on C.*[5]

3.3.2 The Recursive Procedure

In the following lemma, we assume that edges of $\mathrm{VD}^k(S)$, for some $k = 1, \dots, K-1$, are at hand, and we describe an algorithm which finds all the edges of $\mathrm{VD}^{k+1}(S)$ by iterative use of Corollary 3.9. We equip our algorithm with a procedure, similar to the one in Chapter 3.2, that first distinguishes the big and the small $(k+1)$-cells and then reports the $(k+1)$-half-edges incident to each of these cells. This prevents spending too much time on the $(k+1)$-cells with many incident edges.

Lemma 3.10. *Using two scans over all k-half-edges, we can report all $(k+1)$-half-edges in batches of size at most $2s$ such that each $(k+1)$-half-edge is reported exactly once. This takes $O(\frac{n^2 k}{s}(\log s + k\, 2^{O(\log^* k)}))$ expected time or $O(\frac{n^2 k}{s}(\log s + k \log k))$ deterministic time, using $O(sk^2)$ cells of workspace.*

Proof. The algorithm consists of three phases analogous of the ones introduced in Chapter 3.2: in the first phase, we aim at finding the big $(k+1)$-cells. Let e_i denote either a k-half-edge or a $(k+1)$-half-edge. To find the big $(k+1)$-cells we keep s such half-edges $e_1, \dots, e_s$ in memory. At the beginning of the first phase, $e_1, \dots, e_s$ are all k-half-edges. In each iteration of the algorithm, we apply Corollary 3.9 to these half-edges, in order

[5] Note that f_i is always the counterclockwise successor of e_i, even if the containing $(k+1)$-intervals of f_i and e_i are different. This occurs only when e_i is the last $(k+1)$-half-edge on a $(k+1)$-interval of C.

to obtain s new $(k+1)$-half-edges $f_1, \dots, f_s$. Now, for each $i = 1, \dots, s$, three cases can apply: (i) f_i is *null*, i.e., e_i was not relevant. In the next iteration, we replace e_i with a fresh k-half-edge from the input; (ii)/(iii) f_i is not *null*. Now we need to determine whether f_i is the last $(k+1)$-half-edge of the $(k+1)$-interval containing it. For this, we check whether the head vertex of f_i is an old $(k+1)$-vertex. (ii) If f_i is not the last $(k+1)$-half-edge of the $(k+1)$-interval containing it, i.e., if its head vertex is a new $(k+1)$-vertex, we set e_i to f_i for the next iteration. (iii) If f_i is the last $(k+1)$-half-edge of $(k+1)$-interval containing it, we set e_i to a fresh k-half-edge from the input. We repeat this procedure until there are no fresh k-half-edges left to process.[6]

The remaining $(k+1)$-half-edges in the working memory are incident to the *big* $(k+1)$-cells. For each such cell, we store the *center of gravity* of its defining sites in an array $\mathcal{B}^{k+1}$, sorted according to lexicographic order. We emphasize that in the first phase, we do not report any $(k+1)$-half-edge.

In the second phase, we repeat the same procedure as in the first phase, but now that we know the big $(k+1)$-cells, we can report the $(k+1)$-edges. In order to avoid repetitions, we only report (i) every $(k+1)$-half-edge incident to a small $(k+1)$-cell; and (ii) the opposite direction of every $(k+1)$-half-edge e incident to a small $(k+1)$-cell, if the $(k+1)$-cell on the right of e is a big $(k+1)$-cell. See Figure 3.15.

The conditions (i) and (ii) guarantee that the number of $(k+1)$-half-edges that are reported in each iteration is at most $2s$. We use $\mathcal{B}^{k+1}$ to identify if a cell C is a big cell, by locating the center of gravity of the defining sites of C in $\mathcal{B}^{k+1}$ with a binary search; see below for details.

In the third phase, we report every $(k+1)$-half-edge e that is incident to a big $(k+1)$-cell, while the $(k+1)$-cell on the right of e is also a big $(k+1)$-cell. Let $\{\mathcal{B}^{k+1}\}$ denote the sites that define the big $(k+1)$-cells. We construct $\mathrm{VD}^{k+1}(\{\mathcal{B}^{k+1}\})$ in the working memory. Then, we go through the sites in S in batches of size sk, adding the sites of each batch to $\mathrm{VD}^{k+1}(\{\mathcal{B}^{k+1}\})$. While doing this, as in the algorithm for Lemma 3.7, we keep track of how the edges of $\mathrm{VD}^{k+1}(\{\mathcal{B}^{k+1}\})$ are cut by the corresponding cell in the new diagrams. In the end, we report all $(k+1)$-edges of $\mathrm{VD}^{k+1}(\{\mathcal{B}^{k+1}\})$ that are not empty. By *report*, we mean report two $(k+1)$-half-edges in opposing directions, which is again at most $2s$ half-edges. As we explained in the algorithm in Lemma 3.7, these $(k+1)$-half-edges cover all the $(k+1)$-half-edges whose left and right cells are both big $(k+1)$-cells.

Regarding the running time, the first and the second phase of the algorithm consist of $O(nk/s)$ applications of Corollary 3.9 which takes $O(\frac{n^2k}{s}(\log s + k\, 2^{O(\log^* k)}))$ total expected time or $O(\frac{n^2k}{s}(\log s + k \log k))$ total deterministic time. Furthermore, creating the array $\mathcal{B}^{k+1}$ to represent the big cells takes $O(sk + s\log s)$ steps: we compute the center of gravity of the defining sites for each big $(k+1)$-cell in $O(k)$ steps. Then we sort these center points in lexicographic order in $O(s \log s)$ steps. A query in $\mathcal{B}^{k+1}$ takes $O(k + \log s)$ time: given a query $(k+1)$-cell C, we compute the center of gravity for its

[6] We actually repeat this procedure for one more iteration after there are no fresh k-half-edges left to process. This extra iteration guarantees that there are no k-half-edges left in the working memory and all of them have been removed or replaced by $(k+1)$-half-edges.

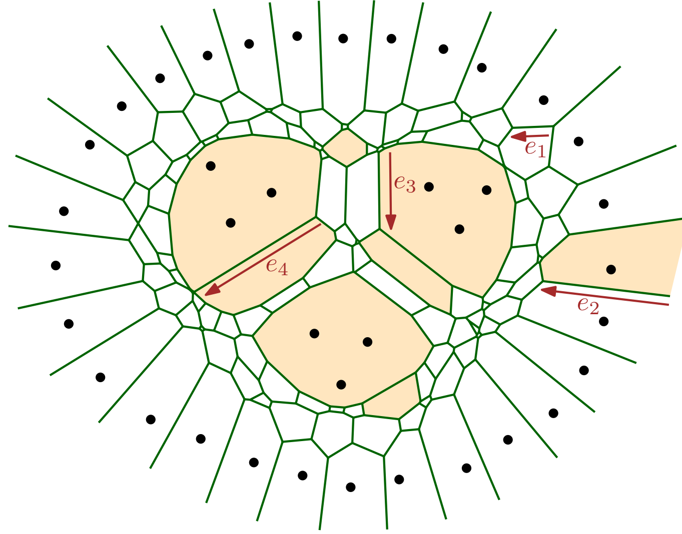

Figure 3.15: The VD^3 for a set of sites and the big 3-cells (filled). In the second phase, when the algorithm finds the half-edges e_1 and e_2, it reports these half-edges as well as the opposite half-edge of (only) e_2. The algorithm may also detect e_3 and e_4, while it processes the big-cells partially. However, in such a case it reports neither them nor their opposite half-edges. Note that in this phase e_3 is reported exactly once, and it occurs when the opposite half-edge of e_3 is detected by the algorithm.

defining sites in $O(k)$ time. Then we use binary-search in $\mathcal{B}^{k+1}$ to find a big $(k+1)$-cell with the same center of gravity. Aurenhammer [Aur90] showed that these centers are pairwise distinct, so that a $(k+1)$-cell can be uniquely identified by the center of gravity of its defining sites.[7]

The algorithm performs at most two queries in $\mathcal{B}^{k+1}$ per $(k+1)$-half-edge, for a total of $O(nk)$ edges. Thus, the total time for the queries is $O(nk^2 + nk \log s)$. In the third phase of the algorithm, constructing a $(k+1)$-order Voronoi diagram of $O(sk)$ sites takes $O(sk \log s + sk^2\, 2^{O(\log^* k)})$ expected time or $O(sk \log s + sk^2 \log k)$ deterministic time. We repeat it $O(n/sk)$ times, which takes $O(n \log s + nk\, 2^{O(\log^* k)})$ expected time or $O(n \log s + nk \log k)$ deterministic time in total.

Overall, the running time of the algorithm simplifies to $O(\frac{n^2k}{s}(\log s + k\, 2^{O(\log^* k)}))$ total expected time or $O(\frac{n^2k}{s}(\log s + k \log k))$ total deterministic time. The algorithm uses a workspace of $O(sk^2)$ cells for running Corollary 3.9, storing big $(k+1)$-cells, and constructing Voronoi diagrams with $O(sk)$ sites. □

[7] To be precise, Aurenhammer [Aur90, Theorem 1] showed the following: take the standard lifting of S onto the unit paraboloid and compute the center of gravity for each subset of $k+1$ lifted points. Call the resulting point set R. Then, the vertical projection of the lower convex hull of R is dual to $\mathrm{VD}^{k+1}(S)$. In particular, the vertices of the projection are the centers of gravity of the defining sites for the cells of $\mathrm{VD}^{k+1}(S)$. Therefore, they must be pairwise distinct; otherwise, they could not all appear on the lower convex hull.

3.3.3 Obtaining a Time-Space Trade-Off

In order to use Lemma 3.8 to compute half-edges of a higher order Voronoi diagram, we need to have all the half-edges of the previous order Voronoi diagram at hand. However, the space constraint does not allow us to store them. To resolve this problem, we find the k-half-edges for all $k = 1, \dots, K$ simultaneously, as follows: for a parameter s' (that we will define later), we compute s' different 1-edges and report every 1-edge as two 1-half-edges in opposing directions. Then, we apply Lemma 3.10 (with parameter s') in a pipelined fashion to obtain the k-half-edges for $k = 2, \dots, K$. In each iteration, the algorithm from Lemma 3.10 consumes at most s' different k-half-edges from the previous order and produces at most $2s'$ new $(k+1)$-half-edges to be used at the next order.

Thus, if we have between s' and $3s'$ new k-half-edges available in a buffer, then we can use them one by one whenever the algorithm for computing $(k+1)$-half-edges in Lemma 3.10 requires such a new k-half-edge. Whenever the size of the buffer falls below s', we run the algorithm for the previous order until the number of k-half-edges in the buffer is again between s' and $3s'$. Applying this idea for all the orders $k = 1, \dots, K-1$, we need to store $K-1$ buffers, each containing up to $3s'$ half-edges for the corresponding diagram. Since a k-half-edge is represented by $O(k)$ sites from S, the buffer for k-half-edges requires $O(s'k)$ cells of workspace. We call this the *output buffer* and we denote it by $\mathcal{O}^k$.

Furthermore, for each k, we need to store $O(s')$ half-edges that reflect the current state of the corresponding algorithm. This requires $O(s'k)$ cells of workspace. This is called the *private workspace* and is denoted by $\mathcal{P}^k$. Finally, for the algorithm that is currently active, we need $O(s'k^2)$ cells of workspace to compute the Voronoi diagram of order k for the next batch of $O(s'k)$ sites from S (see Lemma 3.10). Since this workspace is used by all the algorithms, it is called the *common workspace* and is denoted by $\mathcal{C}$; see Figure 3.16. More details follow.

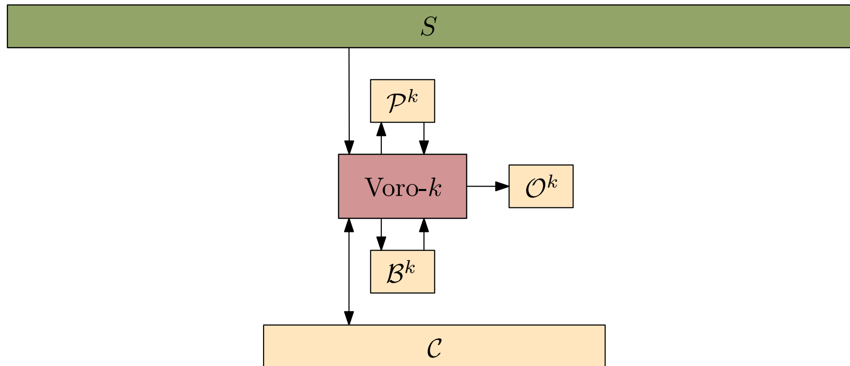

Figure 3.16: The read-only array S contains the given set of sites. The processor Voro-k computes $\mathrm{VD}^k(S)$ with the help of the allocated buffers $\mathcal{P}^k, \mathcal{B}^k, \mathcal{O}^k$ and the common workspace C. The direction of the arrows indicates reading from or writing to the memory cells, for processor Voro-k.

Theorem 3.11. *Let $S = \{p_1, \dots, p_n\}$ be a set of n planar point-sites in general position that is stored in a read-only array. Let s be a parameter in $\{1, \dots, n\}$ and $K \in O(\sqrt{s})$. There is an algorithm that reports all the edges of* $\mathrm{VD}^1(S), \dots, \mathrm{VD}^K(S)$ *in expected time* $O(\frac{n^2K^5}{s}(\log s + K\, 2^{O(\log^* K)}))$ *or in deterministic time* $O(\frac{n^2K^5}{s}(\log s + K \log K))$*, using a workspace of $O(s)$ cells.*

Proof. We compute the half-edges of $\mathrm{VD}^1(S), \dots, \mathrm{VD}^K(S)$ in a pipelined fashion. The algorithm simulates having K processors, each one computing a Voronoi diagram of different order. For $k = 1, \dots, K$, let Voro-k be the processor in charge of computing the Voronoi diagram of order k. We emphasize that the algorithm is sequential, but the analogy of K processors helps our exposition. Set the parameter $s' = s/K^2$. The first processor Voro-1 uses the algorithm of Theorem 3.7 with space parameter s' to compute the 1-half-edges. For $k \geq 2$, the processor Voro-k runs the algorithm from Lemma 3.10 with space parameter s' to compute the k-half-edges. Recall that Lemma 3.10 requires $O(s'k^2)$ cells of workspace to compute the intermediate Voronoi diagrams of order k for a set of $O(s'k)$ sites. However, when Voro-k does not compute a diagram, it needs only a state of $O(s'k)$ cells of workspace.

Therefore, all the processors can share a common workspace $\mathcal{C}$ of size $O(s'k)$. At any point in time, $\mathcal{C}$ is used by a single processor Voro-k to compute edges of $\mathrm{VD}^k(S)$, for some $k \in \{1, \dots, K\}$. For each processor Voro-k, the local state and the other variables needed by the processor, i.e., the edges that are currently being processed by the algorithm for computing $\mathrm{VD}^k(S)$, are stored in a private workspace $\mathcal{P}^k$. In addition, Voro-k has an array $\mathcal{B}^k$ to store the big k-cells. Whenever an edge of $\mathrm{VD}^k(S)$ would be reported, the processor Voro-k instead inserts it into an output buffer $\mathcal{O}^k$. Each of these local arrays $\mathcal{O}^k, \mathcal{P}^k, \mathcal{B}^k$ should be able to store $O(s')$ half-edges and cells of $\mathrm{VD}^k(S)$. Since we need $O(k)$ sites to represent a k-half-edge or a k-cell, the total space requirement for all processors is $O(s'K^2) = O(s)$. Note that our requirement that $K = O(\sqrt{s})$ is crucial in ensuring that the space constraints are not exceeded.

We simulate the parallel execution of the processors with *stages*. For $k = \{1, \dots, K\}$, the goal of each stage k is to report the k-half-edges and also to identify the big $(k+1)$-cells. In stage 0 the goal is only to identify the big 1-cells. This is done by processor Voro-1 which performs the first phase of Theorem 3.7 in order to find the $O(s')$ big cells of $\mathrm{VD}^1(S)$, and to store them in $\mathcal{B}^1$. Now we know the big 1-cells.

In stage 1, using the processor Voro-1, we perform the second phase and the third phase of Theorem 3.7 to find and also report all the half-edges of $\mathrm{VD}^1(S)$ in batches of at most $2s'$ half-edges.[8] The processor Voro-1 stores the batches of the 1-half-edges in the buffer $\mathcal{O}^1$. Whenever there are at least s' half-edges in $\mathcal{O}^1$, we pause Voro-1 while we store its state in $\mathcal{P}^1$, and we start Voro-2 to perform the first phase of Lemma 3.10 with the $\mathcal{O}^1$ as input. The processor Voro-2 takes a batch of s' half-edges from $\mathcal{O}^1$, stores them in $\mathcal{P}^2$, and then it processes them as long as there are at least s' half-edges in $\mathcal{P}^2$. Otherwise, it inserts a new batch of s' half-edges from $\mathcal{O}^1$ into $\mathcal{P}^2$. Whenever the buffer $\mathcal{O}^1$ falls below s' half-edges, we stop running Voro-2 and instead we resume

[8] Note that the 1-half-edges are the only edges reported in stage 1.

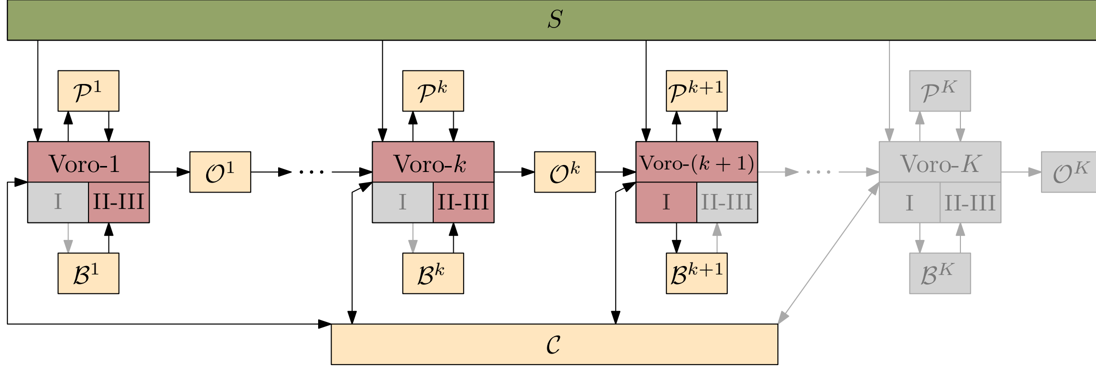

Figure 3.17: For $k' = 1, \ldots, K$, the processor Voro-k' and its allocated memory cells $\mathcal{P}^{k'}$, $\mathcal{O}^{k'}$, $\mathcal{B}^{k'}$ as well as the common workspace $\mathcal{C}$ of all the processors are shown. The direction of the arrows indicates reading from or writing to memory cells. The roman number I, II and III on each processor Voro-k' refer to the first, second, and third phase of the algorithm run by Voro-k'. The figure shows the algorithm in *stage* k. The gray boxes and arrows show the current inactive parts of the algorithm. In this stage, the algorithm reads data from $\mathcal{B}^1, \ldots, \mathcal{B}^k$ and writes into $\mathcal{B}^{k+1}$, and as a result, it reports all the k-half-edges and it identifies all the big $(k+1)$-cells.

Voro-1 until $\mathcal{O}^1$ has at least s' half-edges. Then, we continue running Voro-2 with a similar procedure until Voro-2 has consumed all 1-half-edges (this is the end of the first phase of Lemma 3.10 for the processor Voro-2 and the end of the third phase of Theorem 3.7 for the processor Voro-1). The current half-edges in $\mathcal{P}^2$ represent the big cells of $\mathrm{VD}^2(S)$, and we store them in $\mathcal{B}^2$. This concludes the description of stage 1, in which the 1-half-edges are reported and the big 2-cells are identified.

In general, in stage k of the algorithm, we have identified the big cells $\mathcal{B}^1, \ldots, \mathcal{B}^k$ of the first k diagrams, and we want to use Voro-$(k+1)$ to identify the big cells of $\mathrm{VD}^{k+1}(S)$. For this, all processors Voro-$1, \ldots,$ Voro-k perform the second and the third phase of Theorem 3.7 and Lemma 3.10 in a pipelined fashion to generate all half-edges of $\mathrm{VD}^1(S), \ldots, \mathrm{VD}^k(S)$, and to store them in the buffers $\mathcal{O}^1, \ldots, \mathcal{O}^k$. Furthermore, the processor Voro-$(k+1)$ uses $\mathcal{O}^k$ as an input of the first phase of Lemma 3.10 to generate $\mathcal{B}^{k+1}$ for being used in the next stage; see Figure 3.17. Stage K is similar, but we do not need to determine the big cells of order $K+1$.

By running the K stages of the algorithm, we compute all the Voronoi half-edges and add them to the corresponding output buffers. The edges are computed more than once. Therefore, in order to make sure that they are written into the output memory only once, we report them only the first time they are inserted into the output buffers. For the half-edges of $\mathrm{VD}^k(S)$, this happens in stage k of the algorithm. Thus, we can be certain that every half-edge of each diagram $\mathrm{VD}^1(S), \ldots, \mathrm{VD}^K(S)$ is reported exactly once and in order of their containing diagrams. In other words, all the k-half-edges are reported before all the $(k+1)$-half-edges.

Regarding the running time, in each stage $k = 1, \ldots, K$, we have to compute all diagrams $\mathrm{VD}^1(S), \ldots, \mathrm{VD}^k(S)$ using Lemma 3.10. This takes

$$\sum_{k'=1}^{k} O\Big(\frac{n^2 k'}{s'}\big(\log s' + k'\, 2^{O(\log^* k')}\big)\Big) = O\Big(\frac{n^2 k^2}{s'}\big(\log s' + k\, 2^{O(\log^* k)}\big)\Big)$$

expected time in stage k. The running time for stage 0 is negligible. The complete algorithm takes

$$\sum_{k=1}^{K} O\Big(\frac{n^2 k^2}{s'}\big(\log s' + k\, 2^{O(\log^* k)}\big)\Big) = O\Big(\frac{n^2 K^3}{s'}\big(\log s' + K\, 2^{O(\log^* K)}\big)\Big)$$

expected time for all stages 1 to K. This is $O\Big(\frac{n^2 K^5}{s}\big(\log s + K\, 2^{O(\log^* K)}\big)\Big)$ expected time in terms of s, since $s' = s/K^2$. The analysis for the deterministic running time is completely analogous, replacing the term $2^{O(\log^* k)}$ by $\log k$, which gives us the deterministic time of $O\big(\frac{n^2 K^5}{s}(\log s + K \log K)\big)$. □

CHAPTER 4

Euclidean Minimum Spanning Trees

In this chapter, we describe two s-workspace algorithms for computing the EMST for a given set S of n point-sites in the plane, where the more advanced one runs in $O\big((n^3/s^2)\log s\big)$ time. Our main idea is to apply Kruskal's MST algorithm on RNG(S), using a compact representation of planar graphs, called an *s-net*, which allows us to manipulate the component structure of RNG during the execution of the algorithm.

In Chapter 4.1, we describe an algorithm to compute the edges of RNG(S) (sorted by length) using $O(s)$ cells of workspace. In Chapter 4.2, we explain a simple time-space trade-off for computing EMST(S). Finally, in Chapter 4.3, we introduce the s-net structure, and we equip our algorithm with this tool in order to obtain a better time-space trade-off for computing EMST(S).

4.1 Computing the Relative Neighborhood Graph

For the given set $S = \{p_1, \ldots, p_n\}$, the first goal is to compute RNG(S) in the limited workspace model. In this chapter, we explain our s-workspace algorithm for computing edges of RNG(S) in an arbitrary order and then later in sorted order according to their length. We produce edges of RNG(S) with a similar technique as the one that we have used in the context of Voronoi diagrams in Chapter 3.

4.1.1 All the Incident Edges to Some Sites

In the following lemma, we explain how to compute all the edges of RNG(S) that are incident to a batch of s sites of S using $O(s)$ cells of workspace. Note that the same technique cannot be used for computing all the edges of DT(S) that are incident to s sites of S using only $O(s)$ cells of workspace. This is due to the fact that the vertices in DT(S) are not of bounded degree.

Lemma 4.1. *Let S be a planar set of n point-sites in general position, stored in a read-only array. Given a set $V \subseteq S$ of s sites, we can compute for each $u \in Q$ the at most six neighbors of u in* RNG(S) *in total time $O(n \log s)$ and using $O(s)$ cells of workspace.*

Proof. The algorithm has two phases. In the first phase, for each site in V, we find a superset of its neighbors in RNG(S), and this superset has a size of at most six. In the second phase, we check which of these candidates from the first phase are the actual neighbors in RNG(S) of the sites in V.

The first phase proceeds in $\lceil n/s \rceil$ *steps*. In each step, we process a *batch* of s sites of $S = Q_1 \cup \ldots \cup Q_{\lceil n/s \rceil}$, and we produce at most six candidates for each site of V to be its neighbors in RNG(S). In the first step, we take the first batch $Q_1 \subseteq S$ of s sites, and we compute RNG($V \cup Q_1$). Because both V and Q_1 have at most s sites, we can do this in $O(s \log s)$ time using standard algorithms. For each $u \in V$, we remember the at most six neighbors of u in RNG($V \cup Q_1$). Notice that for each pair $u \in V, v \in Q_1$, if the edge uv is not in RNG($V \cup Q_1$), then the lens of u and v is non-empty. Therefore, there is a site in $V \cup Q_1$ that lies in the lens of u and v, and thus, it is a witness to certify that uv is not an edge of RNG(S). Let N_1 be the set containing all neighbors in RNG($V \cup Q_1$) of all sites in V. Storing N_1, the set of candidate neighbors requires $O(s)$ cells of workspace.

Then, in each step $j = 2, \ldots, O(n/s)$, we take the next batch $Q_j \subseteq S$ of s sites, and we compute RNG($V \cup Q_j \cup N_{j-1}$) in $O(s \log s)$ time using $O(s)$ cells of workspace. For each $u \in V$, we store the set of at most six neighbors of u in this computed graph. Additionally, we let N_j be the set containing all neighbors in RNG($V \cup Q_j \cup N_{j-1}$) of all sites in V. Note that N_j, the set of candidate neighbors, consists of $O(s)$ sites as each site in V has a degree of at most six in the computed graph. At this step, we do not need to store N_{j-1} anymore.

After $\lceil n/s \rceil$ steps we are left with at most six candidate neighbors for each site in V. As mentioned above, for a pair $u \in V, v \in S$, if v is not among the candidate neighbors of u, then, at some point in the construction, there was a site witnessing that the lens of u and v is non-empty. Therefore, only the sites which are in the set of candidate neighbors can define edges of RNG(S). However, all the candidate neighbors are not necessarily the neighbors in RNG(S) of sites in V. See Figure 4.1 for an example.

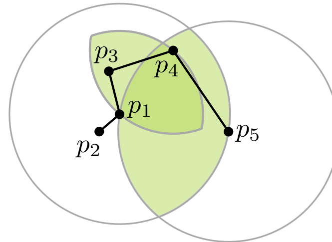

Figure 4.1: For $S = \{p_1, \ldots, p_5\}$, the set of neighbors of p_1 in RNG(S) is $\{p_2, p_3\}$. Consider that p_3 and p_4 are processed in some steps before p_5. After processing p_3 and p_4, the set of the candidate neighbors of p_1 does not contain p_4 because p_3 lies in their lens. This results in adding p_5 to the candidate neighbors of p_1 in one of the following steps. Since p_4 is the only witness site in the lens of p_1 and p_5, the site p_5 will not be removed from the set of candidate neighbors of p_1 until the end of the first phase.

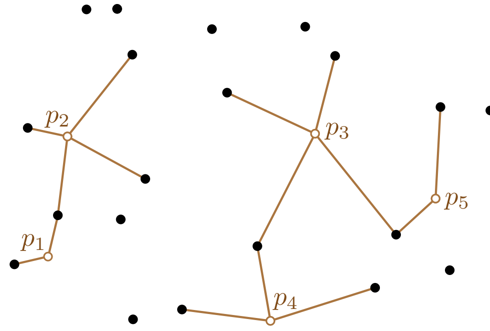

Figure 4.2: For a set of sites S and $V = \{p_1, \ldots, p_5\}$, the neighbors in $\text{RNG}(S)$ of all the sites in V are generated.

In the second phase, to obtain the edges of $\text{RNG}(S)$ incident to the sites in V, we go again through the entire set $S = Q_1 \cup \ldots \cup Q_{\lceil n/s \rceil}$ in batches of size s: in the first step, we start with all the sites in V and their candidate neighbors in $N_{\lceil n/s \rceil}$, and we construct $\text{RNG}(V \cup Q_1 \cup N_{\lceil n/s \rceil})$. For each $u \in V$ and for each candidate neighbor v of u in $N_{\lceil n/s \rceil}$, we check if v is still a neighbor of u in this computed graph. We remove v from the set of candidate neighbors of u, if it does not pass this test. We denote the trimmed set of candidate neighbors of all the sites in V by N'_1. By this procedure, the candidate neighbors in $N_{\lceil n/s \rceil}$ for which there is a witness site in Q_1 will not appear in the updated set of candidate neighbors, N'_1.

Then, in each step $j = 2, \ldots, O(n/s)$, we construct the graph $\text{RNG}(V \cup Q_1 \cup N'_{j-1})$. Again, for each site $u \in V$, we remove its candidate neighbors in N'_{j-1} if there is a witness site in Q_2 lying in their corresponding lens. We denote the trimmed set of the candidate neighbors of all the sites in V by N'_j. In this step, we do not need to store N'_{j-1} anymore. After going through all the batches, the candidates that maintained the empty-lens property throughout define the edges of $\text{RNG}(S)$ incident to the sites in V; see Figure 4.2. Note that in all the steps, N'_j contains at most six candidate neighbors incident to each site of V, and thus, its size is $O(s)$ cells of workspace.

Since the algorithm takes $O(s \log s)$ time per step, and since the number of steps is $2 \times \lceil n/s \rceil$, the total running time of the algorithm is $O(n \log s)$. The space requirement for storing the set of candidate neighbors as well as the intermediate RNGs are $O(s)$ cells of workspace. □

4.1.2 Finding All the Edges of RNG

Through repeated application of Lemma 4.1, we can compute all the edges of $\text{RNG}(S)$, in some arbitrary order, using a workspace of $O(s)$ cells.

Theorem 4.2. *Suppose we are given a planar set of n point-sites $S = \{p_1, \ldots, p_n\}$ in general position, stored in a read-only array. Let s be a parameter in $\{1, \ldots, n\}$. We can compute* $\text{RNG}(S)$ *in total time $O((n^2/s) \log s)$, using $O(s)$ cells of workspace.*

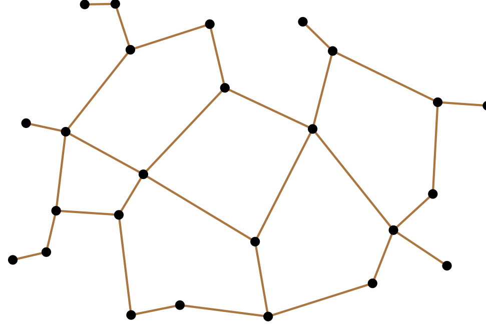

Figure 4.3: The relative neighborhood graph for the set of sites S is generated by producing all the incident edges in $\mathrm{RNG}(S)$ to every batch of s sites of S.

Proof. In the first step of the algorithm, we take the set V of the first s sites of S, and we apply Lemma 4.1 on V to find all the neighbors in $\mathrm{RNG}(S)$ of all the sites in V. Whenever we find a neighbor p_j of a site p_i in $\mathrm{RNG}(S)$, we report the edge p_ip_j only if $i < j$. This guarantees that the edge p_ip_j of $\mathrm{RNG}(S)$ is reported only once. Then in each of the following steps, we take the next batch of s sites of S and repeat the same procedure. We continue this procedure until all the sites in S are processed, i.e., for $O(n/s)$ steps; see Figure 4.3.

Lemma 4.1 guarantees that all the reported edges belong to $\mathrm{RNG}(S)$ and all the edges of $\mathrm{RNG}(S)$ are reported exactly once. Regarding the running time of the algorithm, $O(n/s)$ invocations of Lemma 4.1 take a total of $O\big((n^2/s)\log s\big)$ time. The space requirement is immediate. □

4.1.3 Edges of RNG in Sorted Order of Length

In the following lemma, we provide a technique, that is taken from the work of Chan and Chen [CC07], to produce edges of $\mathrm{RNG}(S)$ in sorted order of length. A similar technique will be used in Chapter 6 in the context of k-visibility regions. Note that having edges of $\mathrm{RNG}(S)$ in sorted order is necessary only in the algorithm in Chapter 4.3, where we introduce the s-net structure. More precisely, in order to update the s-net efficiently, we must add the edges of $\mathrm{RNG}(S)$ one by one in their sorted order. Nevertheless, this procedure is also exploited in our simple algorithm in Chapter 4.2 with the aim of reporting edges of $\mathrm{EMST}(S)$ in their sorted order of length instead of an arbitrary order.

Lemma 4.3. *Let S be a planar set of n point-sites in general position stored in a read-only array. Let $s \in \{1, \dots, n\}$ be a parameter. Let $E_R = e_1, e_2, \dots, e_m$ be the sequence of edges in $\mathrm{RNG}(S)$ sorted by increasing length. Let $i \geq 1$. Given e_{i-1} (or $\perp$, if $i = 1$), we can find the next s edges $e_i, \dots, e_{i+s-1}$ in E_R using $O\big((n^2/s)\log s\big)$ time and $O(s)$ cells of workspace.*[1]

[1] Naturally, if $i + s - 1 > m$, we report the edges $e_i, \dots, e_m$.

Proof. The algorithm in Theorem 4.2 generates all the edges of RNG(S) in total time $O((n^2/s)\log s)$. As we have seen, each step of this algorithm produces a batch of $O(s)$ edges of RNG(S), using Lemma 4.1. Now after each step of this algorithm, instead of reporting the edges, we select the edges $e_i, \dots, e_{i+s-1}$ among them, and we store these edges in the workspace. This can be done with a trick by Chan and Chen [CC07].[2] More precisely, when the algorithm produces $O(s)$ new edges of RNG(S), we store the edges that are longer than e_{i-1} in an array A of size $O(s)$. Whenever A contains more than $2s$ elements, we use a linear time selection procedure to remove all the edges of rank larger than s [CLRS09]. This needs $O(s)$ operations per step of the algorithm in Theorem 4.2, giving a total of $O(n)$ time for selecting the edges. In the end, we have $e_i, \dots, e_{i+s-1}$ in A, albeit not in sorted order. Thus, we sort the final A in $O(s \log s)$ time. The running time for selecting the edges and sorting them is dominated by the time needed to compute all the edges of RNG(S). The space usage for generating the edges and also for selecting and sorting them is bounded by $O(s)$ cells of workspace. Thus, the claim follows. □

4.2 A Simple Time-Space Trade-Off for EMST

The algorithm in Theorem 4.2 for producing edges of RNG(S), together with the techniques from the constant workspace algorithm by Asano et al. [AMRW11] for identifying edges of EMST(S), leads to a simple time-space trade-off for computing EMST(S) that we will explain in this chapter.

4.2.1 Structure of Face-Cycles

Recall that a partial relative neighborhood graph RNG_i is represented as a collection of face-cycles. Asano *et al.* [AMRW11] have observed that, to run Kruskal's algorithm on RNG(S), it suffices to know the structure of the face-cycles of RNG_i, for $i \in \{1, \dots, m\}$. The following observation restates this idea.

Observation 4.4. *Let $i \in \{1, \dots, m\}$. The edge $e_i \in E_R$ belongs to* EMST(S) *if and only if there is no face-cycle F in* RNG_i *such that both endpoints of e_i lie on F.*

Proof. Let u and v be the endpoints of e_i. If there is a face-cycle F in RNG_i that contains both u and v, then e_i clearly does not belong to EMST(S); see Figure 4.4a. Conversely, suppose there is no face-cycle in RNG_i containing both u and v. Thus, every two face-cycles F_u and F_v such that u lies on F_u and v lies on F_v must be distinct. Therefore, F_u and F_v must belong to two different connected components of RNG_i. This is because, if they were in the same component of RNG_i, due to planarity of RNG(S), there would be a face-cycle in RNG_i that both u and v lie on, contradicting the distinctness of F_u and F_v; see Figure 4.4b. This implies that e_i is an edge of EMST(S). □

[2] This technique will be explained in detail in Chapter 5.3.

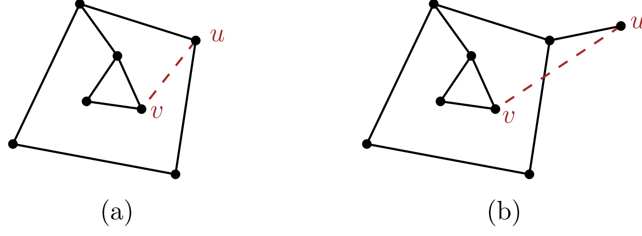

Figure 4.4: A schematic drawing of RNG_i. (a) The edge $uv \notin \text{EMST}(S)$ since there is a face-cycle such that both u and v lie on it. (b) A counter-example for the case that no face-cycle contains both u and v; however, u and v are in the same connected component of RNG_i. The edge uv contradicts the planarity of $\text{RNG}(S)$.

Observation 4.4 tells us that we can identify edges of $\text{EMST}(S)$ if we can determine the face-cycles of each RNG_i that contain the endpoints of e_i, for $i \in \{1, \ldots, m\}$. To accomplish this task, we use the next lemma to traverse the face-cycles.

Lemma 4.5. *Let $i, j \in \{1, \ldots, m\}$ and $i > j$. Suppose we are given the half-edges $\overrightarrow{e_i}, \overrightarrow{e_j} \in E_R$ as well as the at most six edges incident to the head of $\overrightarrow{e_j}$ in* $\text{RNG}(S)$*. Let F be the face-cycle of* RNG_i *that $\overrightarrow{e_j}$ lies on. We can find the next half-edge of $\overrightarrow{e_j}$ on F in $O(1)$ time using $O(1)$ cells of workspace.*

Proof. Let $\overrightarrow{f_j}$ be the next half-edge of $\overrightarrow{e_j}$ on F. Let w be the head of $\overrightarrow{e_j}$. By comparing the length of the edges incident to w in $\text{RNG}(S)$ with $|e_i|$, we identify the incident half-edges of w in RNG_i, in $O(1)$ time. Then, among them we pick the half-edge $\overrightarrow{f_j}$ which has the smallest clockwise angle with $\overrightarrow{e_j}$ around w and has w as its tail. This takes $O(1)$ time using $O(1)$ cells of workspace; see Figure 4.5. □

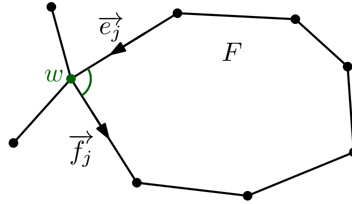

Figure 4.5: For $i > j$, a schematic drawing of a face-cycle F of RNG_i, and $\overrightarrow{e_j}$ on F with the head vertex w, as well as the other edges of RNG_i incident to w. The edge $\overrightarrow{f_j}$ which has the smallest clockwise angle with $\overrightarrow{e}$ is the next edge of $\overrightarrow{e_j}$ on F.

Lemma 4.6. *Let $i, j \in \{1, \ldots, m\}$ and $i \leq j$. Suppose we are given the half-edges $\overrightarrow{e_i}, \overrightarrow{e_j} \in E_R$ as well as the at most six edges incident to the head of $\overrightarrow{e_j}$ in* $\text{RNG}(S)$*. We can find the predecessor and the successor of $\overrightarrow{e_j}$ in* RNG_i *in $O(1)$ time and $O(1)$ cells of workspace.*

Proof. Let $\overrightarrow{p_j}$ be the predecessor and $\overrightarrow{s_j}$ be the successor of $\overrightarrow{e_j}$ in RNG_i. Let w be the head of $\overrightarrow{e_j}$. By comparing the length of the edges incident to w in RNG(S) with $|e_i|$, we identify the incident half-edges of w in RNG_i in $O(1)$ time. Then, among them we pick the half-edge which has w as its head and makes the smallest counterclockwise angle with $\overrightarrow{e_j}$ around w. This is $\overrightarrow{p_j}$. Similarly, we pick the half-edge which has w as its tail and makes the smallest clockwise angle with $\overrightarrow{e_j}$. This is $\overrightarrow{s_j}$. Finding $\overrightarrow{p_j}$ and $\overrightarrow{s_j}$ takes $O(1)$ time using $O(1)$ cells of workspace; see Figure 4.6. □

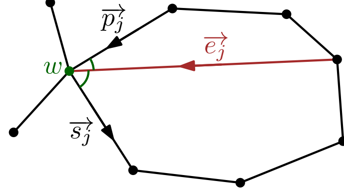

Figure 4.6: For $i \leq j$, a schematic drawing of RNG_i (in black) and a half-edge $\overrightarrow{e_j}$ with the head w. The half-edge $\overrightarrow{s_j}$ has the smallest clockwise angle with $\overrightarrow{e_j}$.

4.2.2 The Algorithm

From our observations so far, we can derive a simple time-space trade-off for computing EMST(S). In Theorem 4.7, we simulate Kruskal's algorithm on RNG(S). For this, we take batches of s edges of RNG(S), and we report the edges of EMST(S) among them. To determine whether an edge e_i of RNG(S) is in EMST(S), we apply Observation 4.4, i.e., we determine whether the endpoints of e_i are on two distinct face-cycles of the corresponding RNG_i.

Theorem 4.7. *Let S be a planar set of n point-sites in general position stored in a read-only array. Let $s \in \{1, \dots, n\}$ be a parameter. We can output all the edges of* EMST(S)*, in sorted order, in $O\big((n^3/s)\log s\big)$ time using $O(s)$ cells of workspace.*

Proof. Let $E_R = e_1, \dots, e_m$ be the edges of RNG(S) sorted by length. In the first iteration, we use Lemma 4.3 to find the batch $e_1, \dots, e_s$ of the first s edges in E_R in $O\big((n^2/s)\log s\big)$ time. For each edge e_i, $i \in \{1, \dots, s\}$, we consider its both half-edges. Then, we perform $2s$ parallel walks starting from the head vertex of each half-edge $\overrightarrow{e_i}$. In the first step of the walks, using Lemma 4.1, we find the at most six incident edges to the head of each half-edge $\overrightarrow{e_i}$. Then, using Lemma 4.6, we identify the predecessor $\overrightarrow{p_i}$ and the successor $\overrightarrow{s_i}$ of $\overrightarrow{e_i}$ in RNG_i (if they exist). By following the successor of each half-edge, we perform one step of the walk for each half-edge of the batch in parallel. Note that the walk that starts from the head of $\overrightarrow{e_i}$ takes place in RNG_i.

Next, in the second step of the parallel walks, we consider the head vertices of all the successors $\overrightarrow{s_i}$. First, we use Lemma 4.1 to find the at most six incident edges to the head of each $\overrightarrow{s_i}$. Then, applying Lemma 4.5, we find the next half-edge of $\overrightarrow{s_i}$, and we advance each half-edge along its face-cycle in RNG_i as one step of the parallel walks. We proceed the parallel walks by finding the next edge on the face-cycles in each step.

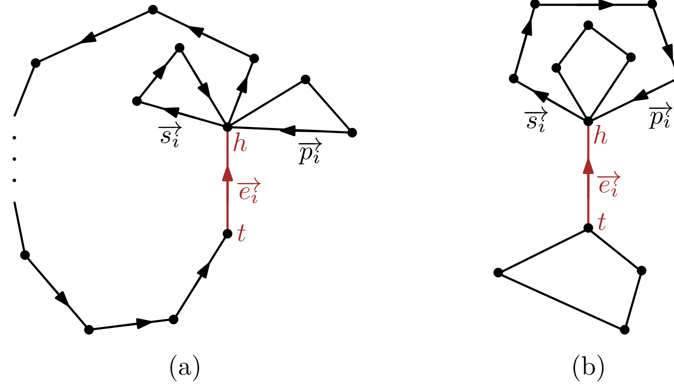

Figure 4.7: A schematic drawing of RNG_i and the half-edge $\overrightarrow{e_i}$ with the head h and the tail t. (a) The vertices h and t are on the same face-cycle of RNG_i since by traversing the face-cycle starting from $\overrightarrow{s_i}$ we encounter t. (b) The vertices h and t are on different face-cycles of RNG_i since by traversing the face-cycle starting from $\overrightarrow{s_i}$ we encounter $\overrightarrow{p_i}$, meaning that we will not reach t.

A walk that started from the head h of $\overrightarrow{e_i}$ continues until it either encounters the tail t of $\overrightarrow{e_i}$ or until it arrives at $\overrightarrow{p_i}$. In the former case, we have found a face-cycle that both endpoints of e_i lie on and thus, by Observation 4.4, e_i is not in $\mathrm{EMST}(S)$; see Figure 4.7a. In the latter case, there is no face-cycle in RNG_i that contains both h and t. This is because, by definition of $\overrightarrow{s_i}$ and $\overrightarrow{p_i}$, all the incident edges of h in RNG_i lie in the counterclockwise cone from $\overrightarrow{p_i}$ to $\overrightarrow{s_i}$ around h. Therefore, by planarity of RNG_i, all the face-cycles that contain h lie inside the face-cycle that starts with $\overrightarrow{s_i}$ and ends at $\overrightarrow{p_i}$. Hence, none of those face-cycles encounters t and, by Observation 4.4, e_i is an edge of $\mathrm{EMST}(S)$; see Figure 4.7b. In this case, we report e_i, and we also abort the walk which was started from the opposite half-edge of $\overrightarrow{e_i}$. This prevents an edge of $\mathrm{EMST}(S)$ to be reported twice.

In the next iteration of the algorithm, we again use Lemma 4.3 to find the next batch of s edges in E_R. Similarly, we perform $2s$ parallel walks for the half-edges in this batch, in order to find the edges which belong to $\mathrm{EMST}(S)$.

Since there are $O(n)$ half-edges in $\mathrm{RNG}(S)$, it takes $O(n)$ steps in each iteration to conclude all the walks, where each step of the walks takes $O(n \log s)$ time. It follows that we can process a single batch of edges in $O(n^2 \log s)$ time which dominates the time needed for finding a batch of s edges of $\mathrm{RNG}(S)$. We have $O(n/s)$ batches, so the total running time of the algorithm is $O((n^3/s) \log s)$. The algorithm uses $O(s)$ cells of workspace for finding and storing a batch of s edges as well as a constant number of cells per edge to perform each walk. $\square$

Note that, in this algorithm, it is not essential to process edges of $\mathrm{RNG}(S)$ in sorted order of length. Thus, we can simply apply Lemma 4.1 to produce edges of $\mathrm{RNG}(S)$. However, by using Lemma 4.3 we are able to report edges of $\mathrm{EMST}(S)$ in sorted order of length, although the total running time of the algorithm will not be affected.

4.3 Improvement via a Compact Representation of RNGs

Theorem 4.7 is clearly not optimal: for the case of linear space $s = n$, we get a running time of $O(n^2 \log n)$, although we know that it should take $O(n \log n)$ time to find EMST(S). Can we do better? The bottleneck in Theorem 4.7 is the time needed to perform the walks in the partial relative neighborhood graphs RNG_i. In particular, such a walk might take up to $\Omega(n)$ steps, leading to a running time of $\Omega(n^2 \log s)$ for processing a single batch of s edges of RNG(S). To avoid this, we will maintain a compressed representation of the partial relative neighborhood graphs that allows us to reduce the number of steps in each walk to $O(n/s)$.

4.3.1 The s-net Structure

Let $i \in \{1, \ldots, m\}$. An *s-net* N for RNG_i is a collection of s half-edges, called *net-edges*, in RNG_i that has the following two properties: (i) each face-cycle in RNG_i with at least $\lfloor n/s \rfloor + 1$ half-edges contains at least one net-edge; and (ii) for any net-edge $\overrightarrow{e} \in N$, let F be the face-cycle of RNG_i that contains $\overrightarrow{e}$. Then on F, between the head of $\overrightarrow{e}$ and the tail of the next net-edge, there are at least $\lfloor n/s \rfloor$ and at most $2\lfloor n/s \rfloor$ other half-edges. Note that the next net-edge on F after $\overrightarrow{e}$ could possibly be $\overrightarrow{e}$ itself. In particular, this implies that face-cycles with less than $\lfloor n/s \rfloor$ edges contain no net-edges; see Figure 4.8.

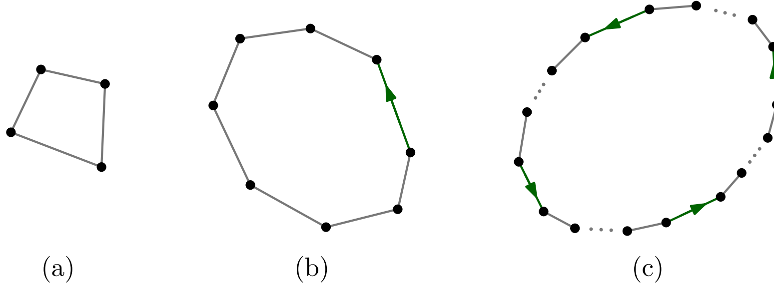

Figure 4.8: A schematic drawing of an s-net for RNG_i. (a) A small face-cycle with no net-edges. (b) a face-cycle with more than $\lfloor n/s \rfloor$ and less than $2\lfloor n/s \rfloor$ half-edges which contains one net-edge. (c) A big face-cycle with four net-edges.

The following observation records two important statements about s-nets.

Observation 4.8. *Let $i \in \{1, \ldots, m\}$, and let N be an s-net for* RNG_i. *Then,*

(N1) N has $O(s)$ half-edges;

(N2) let $\overrightarrow{f}$ be a half-edge of RNG_i*, and let F be the face-cycle that contains it. Then, it takes at most $2\lfloor n/s \rfloor$ steps along F from the head of $\overrightarrow{f}$ until we encounter the tail of either a net-edge or $\overrightarrow{f}$ itself.*

Proof. Property (ii) of the definition of an s-net implies that only face-cycles of RNG_i with at least $\lfloor n/s \rfloor + 1$ half-edges contain net-edges. Furthermore, on these face-cycles, we can uniquely charge $\Theta(n/s)$ half-edges to each net-edge, again by property (ii). Since the face-cycles of RNG_i have $O(n)$ half-edges in total, we conclude the first statement which says $|N| = O(s)$.

For the second statement, we first note that if F contains less than $2\lfloor n/s \rfloor$ half-edges, the claim holds trivially. Otherwise, by property (i), F contains at least one net-edge. Now from property (ii) it is follows that there are at most $2\lfloor n/s \rfloor$ half-edges between every two consecutive net-edges on F. Thus, in a walk on F starting from $\overrightarrow{f}$, we reach a net-edge in at most $2\lfloor n/s \rfloor$ steps. □

Due to statement (*N1*) of Observation 4.8, an s-net can be stored in $O(s)$ cells of workspace. This makes the concept of s-net useful in our s-workspace algorithm. Therefore, we can exploit the s-net in order to speed up the processing of a single batch. The next lemma shows how this is done.

Lemma 4.9. *Let $i \in \{1, \dots, m\}$. Suppose we are given $E_{i,s} = e_i, \dots, e_{i+s-1}$, a batch of s edges in E_R. Furthermore, we have an s-net N for* RNG_i *in our workspace. Then, we can determine which edges from $E_{i,s}$ belong to* $\mathrm{EMST}(S)$ *in $O((n^2/s) \log s)$ time using $O(s)$ cells of workspace.*

Proof. Let H be a set of half-edges, which contains all net-edges from N, as well as, for each *batch-edge* $e_j \in E_{i,s}$, the successor of the two half-edges of e_j in RNG_i; see Figure 4.9. By definition, we have $|H| = O(s)$, and thus, it takes $O(n \log s)$ time to compute H. This is done by using Lemma 4.1 for finding the incident edges of the head of each e_j and Lemma 4.6 for identifying the successors of each e_j.

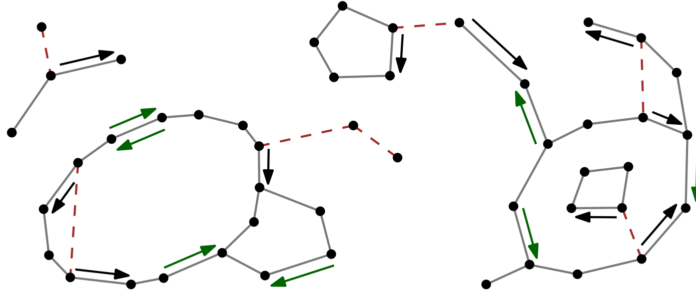

Figure 4.9: A schematic drawing of RNG_i with a batch of edges in E_R (dashed red segments). The directed segments represents the half-edges in H. The net-edges are in green and the successors of the batch edges are in black.

Now starting from the half-edges in H, we perform parallel walks through the face-cycles of RNG_i, one walk per each half-edge, and each such a walk proceeds until it encounters the tail of a half-edge in H (including the starting half-edge itself). In each step of these walks, we use Lemma 4.1 and Lemma 4.5 to find the next half-edges on the face-cycles in $O(n \log s)$ time, and then we check whether these new half-edges belong

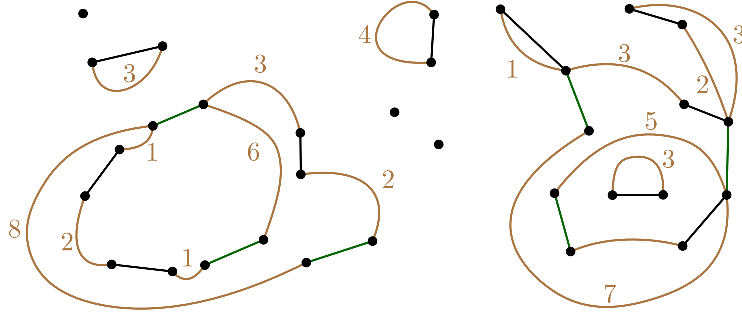

Figure 4.10: The auxiliary graph G is shown. The edge set of G contains the net-edges (in green), the successors of batch-edges (in black), and the compressed edges (beige paths).

to H in $O(s \log s)$ time. Because H contains the net-edges of N, by statement ($N2$) of Observation 4.8, each walk finishes after $O(n/s)$ steps, and thus, the total time for this procedure is $O((n^2/s) \log s)$.

Next, we build an auxiliary *undirected* graph G as follows: the vertices of G are the endpoints of the half-edges in H and the endpoints of the half-edges of $E_{i,s}$. Furthermore, G contains undirected edges for all the half-edges in H and additional *compressed edges*, that represent the outcomes of the walks: if a walk started from the head h of a half-edge in H and ended at the tail t of a half-edge in H, we add an edge from h to t in G, and we label it with the number of steps that were needed for the walk, i.e., the number of half-edges between h and t on that face-cycle. Thus, G contains *H-edges*, and *compressed edges*; see Figure 4.10. Clearly, after all the walks have been terminated, we can construct G in $O(s)$ time, using $O(s)$ cells of workspace.

The auxiliary graph G is actually a representation of the face-cycles in RNG_i. Thus, by adding the batch-edges of $E_{i,s}$ one by one into G, it can represent the next partial relative neighborhood graphs, up to RNG_{i+s}. Hence, we can use G to identify which of the batch-edges of $E_{i,s}$ belong to $\text{EMST}(S)$. This is done by applying Kruskal's algorithm on G as follows: we determine the connected components of G in $O(s)$ time using depth-first search. Then, we insert the batch-edges into G, one after another, in sorted order. As we do this, we keep track of how the connected components of G change, using a union-find data structure [CLRS09]. Whenever a new batch-edge connects two distinct connected components of G, we output it as an edge of $\text{EMST}(S)$. Otherwise, we do nothing; see Figure 4.11. Note that even though G may have a lot more components than RNG_i, the algorithm is still correct because of Observation 4.4.

This execution of Kruskal's algorithm and updating the structure of connected components of G takes $O(s \log s)$ time which is dominated by the running time of $O((n^2/s) \log s)$ to perform the parallel walks. The space requirement for constructing and storing the set H and the graph G as well as the updated versions of G is a total of $O(s)$ cells of workspace. □

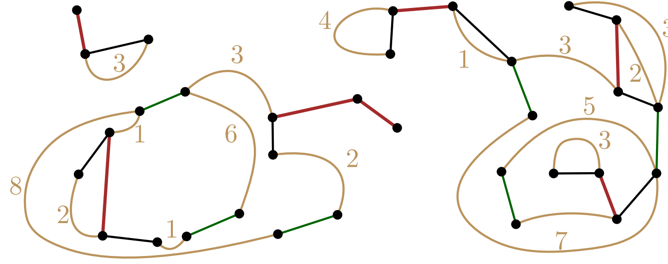

Figure 4.11: The batch-edges of $E_{i,s}$ (in red) have been added to the auxiliary graph G.

4.3.2 Maintaining the s-net

Now that we have described how to use an s-net for RNG_i in order to process the edges $e_i, \dots, e_{i+s}$ of E_R, we need to explain how to maintain the s-net during the algorithm, i.e., construct an s-net for RNG_{i+s} after processing the edges $e_i, \dots, e_{i+s}$. The algorithm in the following lemma computes an s-net for RNG_{i+s}, provided that we have an s-net for RNG_i as well as the graph G described in the proof of Lemma 4.9, for each $i \in \{1, \dots, m\}$.

Lemma 4.10. *Let $i \in \{1, \dots, m\}$, and suppose we have the graph G derived from RNG_i as above, such that all batch-edges have been inserted into G. Then, we can compute an s-net N for RNG_{i+s} in time $O((n^2/s)\log s)$, using $O(s)$ cells of workspace.*

Proof. By construction, all *big* face-cycles of RNG_{i+s}, which are the face-cycles with at least $\lfloor n/s \rfloor + 1$ half-edges, appear as faces in G. Thus, by walking along all faces in G, and taking into account the labels of the compressed edges, we can determine these big face-cycles in $O(s)$ time. The big face-cycles are represented through sequences of H-edges, compressed edges, and batch-edges. For each such sequence, we determine the positions of the half-edges for the new s-net N, by spreading the half-edges equally at minimum distance $\lfloor n/s \rfloor$ and maximum distance $2\lfloor n/s \rfloor$ along the sequence, again taking the labels of the compressed edges into account. Since the compressed edges have length $O(n/s)$, for each of them, we create at most $O(1)$ new net-edges. Now that we have determined the positions of the new net-edges on the face-cycles of RNG_{i+s}, we perform $O(s)$ parallel walks in RNG_{i+s} to actually find them. Using Lemma 4.1 and Lemma 4.5, this takes $O((n^2/s)\log s)$ time. See Figure 4.12. $\square$

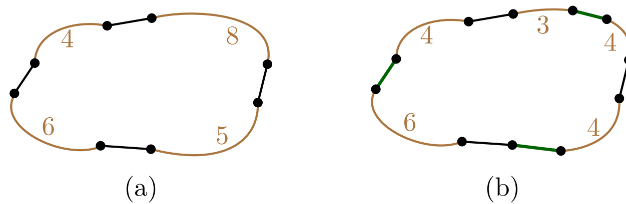

Figure 4.12: (a) A schematic drawing of a face-cycle in G and (b) distributing the new net-edges (in green) on this face-cycle with almost equal distances.

We now have all the ingredients for our main result that provides a smooth trade-off between the cubic-time algorithm in constant workspace and the classical $O(n \log n)$-time algorithm with $O(n)$ cells of workspace. The following theorem presents this algorithm.

Theorem 4.11. *Let S be a planar set of n point-sites in general position stored in a read-only array. Let $s \in \{1, \ldots, n\}$ be a parameter. We can report all the edges of* $\text{EMST}(S)$*, in sorted order of length, in $O\big((n^3/s^2) \log s\big)$ time using $O(s)$ cells of workspace.*

Proof. This follows immediately from our lemmas: applying Lemma 4.3, we produce a batch of s edges of $\text{RNG}(S)$ in sorted order of length. Then, among them, we report the edges of $\text{EMST}(S)$, using Lemma 4.3. Finally, we maintain the s-net structure to be used for the next batch of s edges of $\text{RNG}(S)$, by Lemma 4.10. All these steps are done in $O\big((n^2/s) \log s\big)$ time using $O(s)$ cells of workspace. Since $\text{RNG}(S)$ has $O(n)$ edges, we need to process $O(n/s)$ batches of edges of $\text{RNG}(S)$, leading to an s-workspace algorithm with total running time of $O\big((n^3/s^2) \log s\big)$. □

For our algorithm, it suffices to update the s-net every time that a new batch is considered. It is, however, possible to maintain the s-net and the auxiliary graph G through insertions of single edges. This allows us to handle graphs constructed incrementally and maintain their compact representation using $O(s)$ workspace cells. We believe this is of independent interest and can be used by other algorithms for planar graphs in the limited-workspace model.

Part

Problems on Polygonal Domains

CHAPTER 5

Preliminaries and Background on Polygonal Domains

In this chapter, we introduce basic notations and preliminaries on geometric problems in the limited workspace model that deal with polygonal domains. We also present some tools that will be exploited later. Our focus will be on *visibility* type problems, and we investigate different notions in this family. First, we provide some basic definitions.

Definitions. A polygon is called *simple* if there is no pair of non-consecutive edges sharing a point. The boundary of a simple polygon is a simple closed polygonal chain; see Figure 5.1a. A *polygonal domain* P with $h \geq 0$ holes and n vertices is a connected and closed subset of $\mathbb{R}^2$ with h holes, whose boundary consists of $h+1$ pairwise disjoint simple closed polygonal chains of n line segments in total. One of the simple closed polygonal chains is the outer boundary, and the others form the h holes. The interior induced by a hole boundary and the exterior of the outer boundary are not contained in P. We also refer to it as a polygon with h holes; see Figure 5.1b.

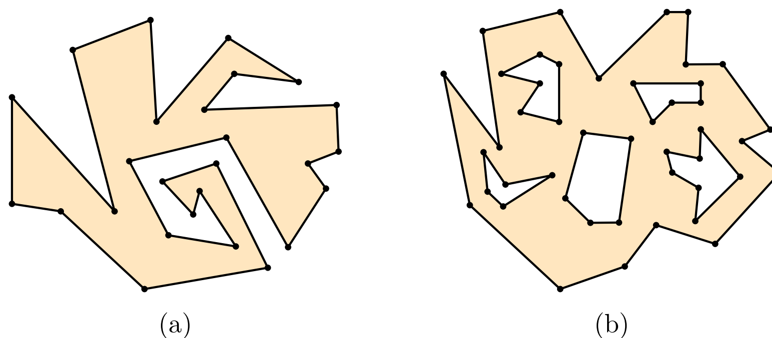

Figure 5.1: (a) A simple polygon. (b) A polygon with $h = 5$ holes.

We assume that the input polygon P with h holes is given as a sequence of n vertices in counterclockwise order (i) along ∂P, if $h = 0$, or in other words, if P is a simple polygon; and (ii) along the outer boundary of P followed by the boundaries of the holes, one by one, in no particular order.

5.1 Visibility Region

Visibility problems have played a major role in computational geometry for a long time; see [Gho07] for an overview. In the following, two of the simplest problems in this family are introduced.

Definition. Let P be a simple polygon with n vertices and let $q, p \in P$ be two points in P. The point p is *visible* from q if and only if the line segment pq has no proper intersection with the boundary ∂P of P; however, it may touch ∂P; see Figure 5.2a. The *visibility region* of the point q in P is the set of all points in P that are visible from q, and it is denoted by $V(P, q)$; see Figure 5.2b.

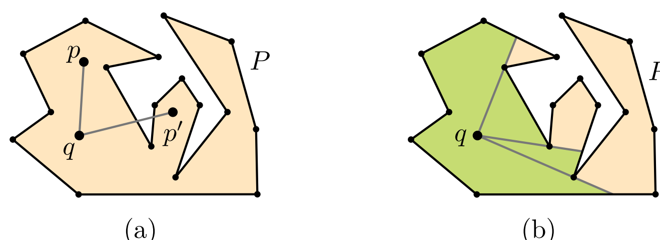

Figure 5.2: A simple polygon P and the point $q \in P$. (a) From q, the point $p \in P$ is visible but $p' \in P$ is not. (b) The visibility region of q in P is shown (in green).

A vertex of P is *reflex* if its inner angle is bigger than π. A reflex vertex v of P is called *reflex with respect to* q if both incident edges to v lie on the same side of the line through q and v. We can observe that the reflex vertices with respect to q are the vertices where important changes may occur in the visibility region of q. More precisely, $\partial V(P, q)$ contains some chains on ∂P and some line segments in the interior of P whose endpoints lie on ∂P and it actually connects the endpoints of the chains. Such a line segment is called a *chord*. Each chord on $\partial V(P, q)$ is characterized as a specific segment on the line through q and a reflex vertex of P with respect to q. Moreover, that reflex vertex is one endpoint of the chord; see Figure 5.3. Since q is fixed, we simply refer to the reflex vertices with respect to q as reflex vertices.

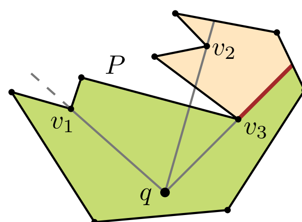

Figure 5.3: Among the three reflex vertices v_1, v_2, v_3 of P, only v_2 and v_3 are reflex with respect to q, and among them, only v_3 is visible from q. Thus, v_3 creates a chord which belongs to $\partial V(P, q)$. The chord is shown in red.

The problem of computing, (i.e., reporting the edges of) the visibility region of a given point in a given simple polygon with n vertices can be solved in $O(n)$ time and $O(n)$ space, using a classic algorithm [JS87].

Another notion in this area is called *weak visibility region*, and it is defined as follows: given a simple polygon P and an edge e of P, the *weak visibility region* of e in P is the set of all the points in P that are visible from at least one point on e; see Figure 5.4. The algorithms for computing the weak visibility region normally find the visible part of each edge of P from e. This is known as *edge-to edge* visibility. In the classic model, Avis *et al.* [AGT86] described an $O(n)$-time algorithm to compute the visible part of one edge from another which uses $\Omega(n)$ cells of workspace. Thus, the best known algorithm for computing the weak visibility region of an edge in a simple polygon using at least $\Omega(n)$ cells of workspace has a running time of $O(n^2)$ [AGT86].

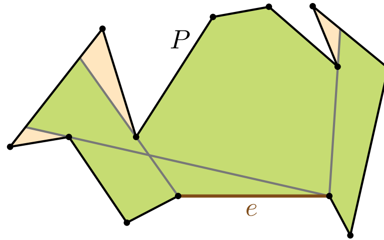

Figure 5.4: For a simple polygon P and an edge $e \in P$, the weak-visibility region of e in P is shown in green.

Constant Workspace Algorithms. Barba *et al.* [BKLS14] presented a constant workspace algorithm that reports the visibility region of a point q in a simple polygon P, in $O(n\bar{r})$ time, where $\bar{r}$ is the number of reflex vertices of P (with respect to q) that are visible from q. Their algorithm scans ∂P in counterclockwise order, and it reports the maximal visible subchains of ∂P. More precisely, the algorithm first finds a visible vertex v_{start} of P in $O(n)$ time. Then it performs a walk on ∂P from v_{start} in counterclockwise direction, until it reaches the next reflex vertex v_{vis} that is visible from q. They also find the first intersection point of ∂P with the ray qv_{vis} from q, which is called the *shadow* of v_{vis}. This step of the algorithm takes $O(n)$ time and $O(1)$ cells of workspace.

To proceed to the next step, they observed that the end vertex of the maximal counterclockwise visible chain starting at v_{start} is either v_{vis} or its shadow. In each case, the next maximal visible chain starts at the other of the two vertices (v_{vis} or its shadow). Thus, by identifying the shadow of v_{vis}, we find a maximal visible chain and a new starting point; see Figure 5.5. This takes $O(n)$ time using only $O(1)$ cells of workspace. The number of iterations is equal to the number of reflex vertices that are visible from q, thus it is $\bar{r}$. This gives an algorithm with $O(n\bar{r})$ running time and $O(1)$ cells of workspace. They also extend their algorithm to a time-space trade-off; more details follow below.

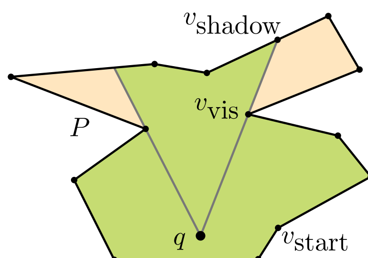

Figure 5.5: The visible chain starting at v_{start} ends at the next counterclockwise visible reflex vertex v_{vis}. The next chain starts with v_{shadow} which is the shadow of v_{vis} and ends at the shadow of the next counterclockwise visible reflex vertex.

For the weak visibility problem, Abrahamsen developed a constant workspace algorithm that computes the weak visibility region of an edge e of a simple polygon P with n vertices [Abr13]. This algorithm first considers the edge-to-edge visibility between e and any other edge of P. More precisely, for each edge e' of P such that $e \neq e'$, the algorithm computes the visible part of e' from e in $O(n)$ time; see Figure 5.6. This leads to an $O(nm)$ time algorithm for finding the weak visibility region of e inside P using a constant number of cells of workspace, where m denotes the size of the resulting weak visibility region. And that m is at most n. This result also gives an $O(n^2)$ time algorithm for computing a minimum-link-path between two points in a simple polygon with n vertices that uses only $O(1)$ cells of workspace. The minimum-link-path is defined as a polygonal path inside the polygon that has the minimum number of segments.

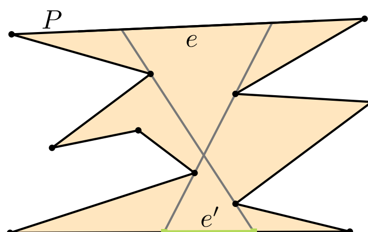

Figure 5.6: For the edges e and e' of the simple polygon P, the visible part of e' from e is shown as a green segment.

Time-Space Trade-Offs. For computing the visibility region of a given point q in a given simple polygon P, Barba *et al.* [BKLS14] provided a time-space trade-off. Their method is recursive, and it uses their constant workspace algorithm as the base. In each step of the recursion, the algorithm splits a chain on ∂P into two subchains, such that each subchain contains roughly half of the visible reflex vertices of the original chain. This results in an s-workspace algorithm, for $s \in \{1, \ldots, O(\log r)\}$, that runs in $O(nr/2^s + n \log^2 r)$ total deterministic time or in $O(nr/2^s + n \log r)$ total randomized expected time. Here, r is the number of reflex vertices in P with respect to q.

Only slightly later, a superset of the authors [BKL+15] gave an improved algorithm for the visibility problem that runs faster for specific combinations of r, s, and n. In fact, Barba *et al.* [BKL+15] discovered a much more general method for obtaining time-space trade-offs for a wide class of geometric algorithms that they classify as *stack-based* algorithms. Intuitively, a deterministic incremental algorithm is stack-based if its main data structure takes the form of a stack. In addition to the algorithm for computing the visibility region [JS87], classic examples from this category include the algorithms for computing the convex hull of a simple polygon by Lee [Lee83] or for triangulating a monotone polygon by Garey *et al.* [GJPT78]. Other applications in graphs of bounded treewidth are also known [BCR+15].

The general trade-off is obtained by using a *compressed stack* that explicitly stores only a subset of the stack that is needed during the computation and that recomputes the remaining parts of the stack as they are needed. Some delicate work goes into balancing the space required for the partial stack and the time needed for reconstructing the other parts. The upshot of applying this technique is as follows: given a stack-based algorithm that on input size n runs in $O(n)$ time and uses a stack with $O(n)$ cells, one can obtain an algorithm that uses s cells of workspace and runs in $O(n^2 \log n/2^s)$ time for $s = o(\log n)$, and in $n^{1+O(1/\log s)}$ time for $s \geq \log n$.[1] An experimental evaluation of the framework was conducted by Baffier *et al.* [BDK18].

5.2 k-Visibility Region

A natural extension of the concept of visibility is captured as k-visibility.

Definition. Let P be a simple polygon with n vertices and q be a point in P. Let $k \in \{0, \dots, n-1\}$. A point p in the plane is *k-visible* from q if and only if the segment pq properly intersects ∂P at most k times. Note that the touching points of pq with ∂P, and particularly p and q, do not count toward the number of intersections; see Figure 5.7.

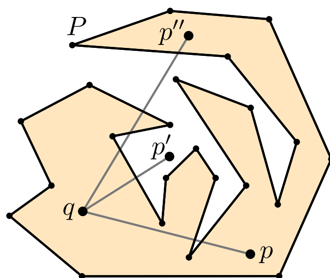

Figure 5.7: A simple polygon P and a point $q \in P$ are shown. The two points p and p' are 2-visible from q. However, the point p'' is not 2-visible from q since the line segment qp' intersects ∂P in four points.

[1] The actual trade-off is more nuanced, but we simplified the bound to make it more digestible for the casual reader. More details can be found in the original paper [BKL+15].

The *k-visibility region* of a point q in P which we denote by $V_k(P, q)$ is defined as the set of all points in P that are k-visible from q. For $k = 0$, this notion corresponds to the classic visibility concept in simple polygons, as in Chapter 5.1. Thus, if we interpret ∂P as the walls of a building, the 0-visible region of q is the set of points that q can see directly, without seeing through the walls. If we consider 2-visibility, we allow the segment to leave (and re-enter) P once, and so on; see Figure 5.8. Since each of the n edges of P can be counted as at most one wall that blocks the visibility of q in any fixed direction, for $k = n - 1$, all the points in P are k-visible from q. Therefore, there is no reason to consider $k > n - 1$.

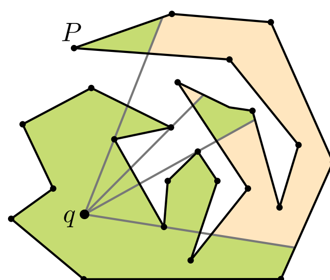

Figure 5.8: The 2-visibility region of a point q in a simple polygon P is shown in green.

While the 0-visibility region of a point in a simple polygon is always connected, the k-visibility region may have several components. The problem of computing $V_k(P, q)$ deals with reporting all the edges of all the components of $V_k(P, q)$. The boundary $\partial V_k(P, q)$ consists of pieces of ∂P, i.e, sub-segments of some of the edges of P, and also chords in P that connect two such pieces. We will explain more about the geometry of these chords in the next chapter.

Without loss of generality, we assume that k is even: if k is odd, we can instead compute $V_{k-1}(P, q)$, which is the same as $V_k(P, q)$. This is because, for odd k, all the k-visible points from q that are not $(k-1)$-visible, lie outside of the polygon P, and thus by definition, do not belong to the k-visibility region of q in P; see Figure 5.9.

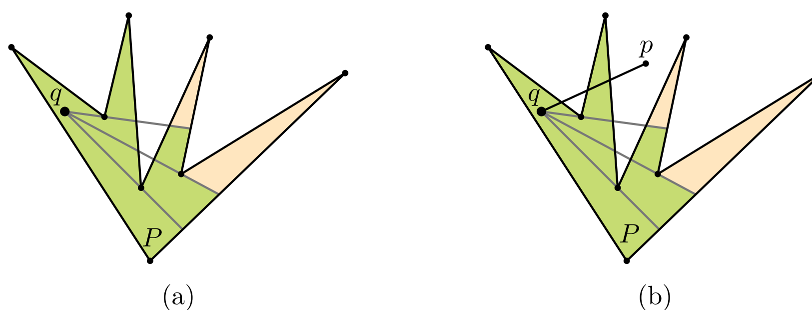

Figure 5.9: A simple polygon P and a point $q \in P$. (a) The 2-visibility region of q in P is shown in green. (b) The 3-visibility region of q in P is shown in green. The point p is 3-visible from q but it is not in the 3-visibility region of q in P.

We fix a coordinate system with origin q. For $\theta \in [0, 2\pi)$, $r(\theta)$ denotes the ray that emanates from q and has a counterclockwise angle of θ with the x-axis. We require that the input is in *weak general position*, i.e., there is no $\theta \in [0, 2\pi)$ such that $r(\theta)$ goes through two distinct vertices of P. An edge of P that intersects $r(\theta)$ is called an *intersecting edge* of $r(\theta)$. Moreover, the *edge list* of $r(\theta)$ is defined as the list of intersecting edges of $r(\theta)$ that are sorted according to their intersection with $r(\theta)$ in increasing distance from q. The jth element of this list is denoted by $e_j(\theta)$. We also say that $e_j(\theta)$ has *rank* j in the edge list of $r(\theta)$, or simply has rank j on $r(\theta)$; see Figure 5.10.

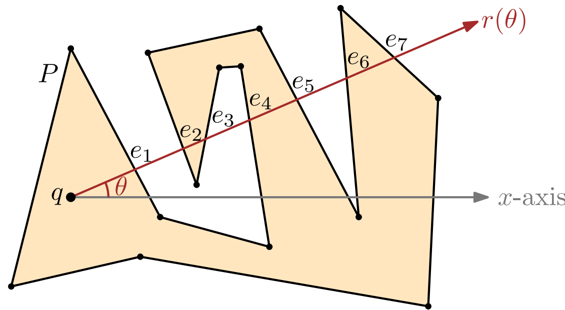

Figure 5.10: The ray $r(\theta)$ from q in the polygon P is shown. The edges $e_1, \ldots, e_7$ intersect $r(\theta)$ such that, for $i \in \{1, \ldots, 7\}$, the edge e_i has rank i in the edge list of $r(\theta)$, and thus, it represents $e_i(\theta)$.

The *angle* of a vertex v of P refers to the angle $\theta \in [0, 2\pi)$ at which the ray $r(\theta)$ encounters v. Suppose $r(\theta)$ stabs a vertex v of P. From the viewpoint q, the vertex v is a *critical* vertex if its incident edges lie on the same side of $r(\theta)$, and v is *non-critical* otherwise. We can check in $O(1)$ time whether a given vertex of P is critical. We use c to denote the number of critical vertices in P from the viewpoint q. Let v be a critical vertex. We call v a *start vertex* if both incident edges of v lie counterclockwise of the ray r_θ; otherwise, we call v an *end vertex*; see Figure 5.11.

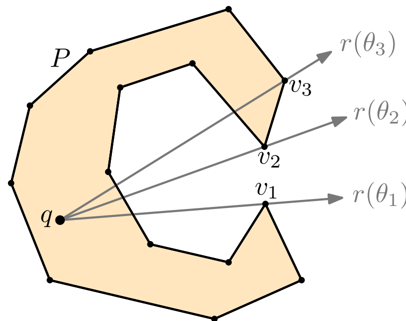

Figure 5.11: From the viewpoint q in the polygon P, the vertices v_1 and v_2 are two of the critical vertices of P, and v_3 is one of the non-critical vertices. Furthermore, v_1 is an end vertex, and v_2 is a start vertex.

A *chain* is defined as a sequence of edges of P (in clockwise or counterclockwise order along ∂P) that starts at a start vertex and ends at an end vertex and contains no other critical vertices. Note that every ray $r(\theta)$, for $\theta \in [0, 2\pi)$, intersects each chain at most once. Thus, the chain list of $r(\theta)$ is defined similarly as the edge list: the list of the chains that intersect $r(\theta)$ in the sorted order of their intersections with $r(\theta)$ form q; see Figure 5.12 for an illustration.

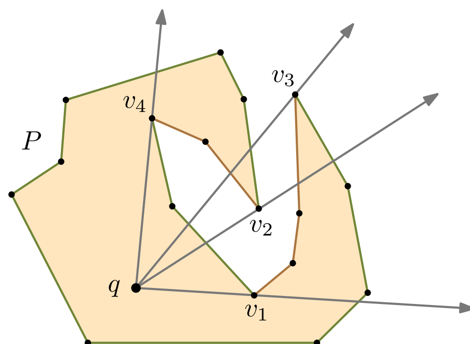

Figure 5.12: From the viewpoint q in the polygon P, the vertices v_1 and v_2 are start vertices, and v_3 and v_4 are end vertices. ∂P is partitioned into 4 disjoint chains, namely, the clockwise chain v_1v_3 (in brown), the counterclockwise chain v_1v_4 (in green), the clockwise chain v_2v_4 (in brown), and the counterclockwise chain v_2v_3 (in green).

Previous Results. After the classic visibility problems, the concept of 1-visibility first appeared in a work by Dean *et al.* [DLS88] as far back as 1988. In the related *superman problem* [MS94], we are given two polygons P and G such that $G \subseteq P$, and a point $p \in P \setminus G$. The goal is to find the minimum number of edges in P that need to be made opaque in order to make G invisible from p.

The general k-visibility, for $k > 1$, is more recent. Since 2009, this variant of visibility has been explored more widely due to its relevance in wireless networks. In particular, it models the coverage areas of wireless devices whose radio signals can penetrate up to k walls [AFMFP+09, FMVU09]. This makes the problem particularly interesting for the limited workspace model, since these wireless devices are typically equipped with only a small amount of memory for computational tasks and may need to determine their coverage region using the few resources at their disposal. The notion of k-visibility has been also considered in the context of art-gallery-style questions [BBB+13, EGS07, FHP09, O'R12] and in the definition of certain geometric graphs [DEG+05, FM08, HVW07].

Bajuelos *et al.* [BCHPM12] presented an algorithm for a slightly different notion of k-visibility. They consider all k-visible points in the plane and not just the points inside the polygon. Their algorithm computes the region of the *plane* that is k-visible from q in the presence of a simple polygon P with n vertices. In this setting, the k-visibility region is connected; see Figure 5.13.

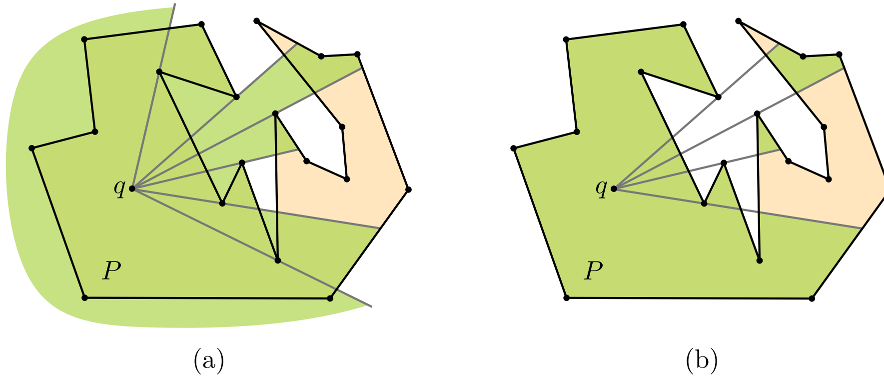

Figure 5.13: A simple polygon P and the point $q \in P$. (a) The region of the points in the plane which are k-visible from q (by the definition in [BCHPM12]) is shown in green. (b) The k-visibility region of q in P (by the definition in this thesis) is shown in green.

The algorithm by Bajuelos *et al.* finds the intersecting edges of P with all the rays from q through the vertices of P. Then, for each ray, it determines the rank of each of its intersecting edges. This takes a total of $O(n^2)$ time and $O(n^2)$ space. Next, the algorithm reports the edge with rank k on each ray, and it also identifies and reports the connecting chords. The algorithm runs in $O(n^2)$ time using $O(n^2)$ cells of workspace.

Time-Space Trade-offs. We provide the first time-space trade-offs for the problem of computing the k-visibility region of a given point q in a given polygon P. Our algorithm uses $O(s)$ cells of workspace, where s may range from 1 to n, and it reports the edges of $\partial V_k(P, q)$ in some arbitrary order in $O((k + c)n/s + n \log s)$ deterministic time, or in a slightly faster $O(cn/s + c \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ randomized expected time. Here, $1 \le c \le n$ is the number of critical vertices of P. Our approach requires a careful analysis of the combinatorics of the k-visible region of p in P, and it makes use of known time-space trade-offs for the k-selection problem [CMR14]; more details follow.

We generalize this result for polygons with holes and for sets of non-crossing line segments. More specifically, we show that in a polygon P with h holes, we can report the k-visibility region of a point $q \in P$ in $O(cn/s + c \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time using $O(s)$ cells of workspace. Furthermore, in an arrangement of n pairwise non-crossing line segments, reporting the k-visibility region of a point q takes $O(n^2/s + n \log s)$ deterministic time [BBB+18]. More details are provided in Chapter 6.

Regarding the slightly different variant of k-visibility which was defined by Bajuelos *et al.* [BCHPM12], we believe that our ideas are also applicable for this notion, and they lead to an improvement of their result. More precisely, the algorithm of Bajuelos *et al.* [BCHPM12] essentially first computes a complete arrangement of quadratic size that encodes the whole visibility information and then extracts the k-visible region from this arrangement. Our algorithms, on the other hand, use a plane sweep so that only the relevant parts of this arrangement are considered. Thus, when $O(n)$ cells of workspace are available, we achieve a running time of $O(n \log n)$.

Since the 0-visibility and the k-visibility region for $k > 0$ have different properties, there seems to be no straightforward way to generalize the approach for computing the 0-visibility region by Barba *et al.* [BKLS14] to our setting. Actually, none of their crucial observations for the 0-visibility region are extensible for the k-visibility region. Furthermore, their idea of processing the vertices in their order along the boundary of the polygon entails performing a selection step for each critical vertex. In contrast, our method processes the vertices in the order of their occurrence in an angular sweep giving us the advantage to perform the selection step only for some critical vertices and to update the results for the other ones.

5.3 Sorting

In our algorithms in the next chapter, we need to identify the contiguous batches of vertices in angular order. To do this efficiently, we will use a procedure that is taken from the work of Chan and Chen [CC07]. The next lemma explains this prodecure; see the second paragraph in the proof of Theorem 2.1 in [CC07].

Lemma 5.1. *Suppose we are given a read-only array A with n pairwise distinct elements from a totally ordered universe, and an element $x \in A$. Let s be a given parameter in $\{1, \dots, n\}$. We can find the set of the first s elements in A that follow x when they are sorted in $O(n)$ time using a workspace of size $2s$ elements of the universe.*

Proof. Let $A_{>x}$ be the subsequence of A that contains exactly the elements in A that are larger than x. The algorithm makes a single and sequential pass over A and selects the elements in $A_{>x}$ by single comparisons. This takes a total of $O(n)$ time. During this pass, every batch of s elements of $A_{>x}$ is being processed in one step. The first step is as follows: the algorithm inserts the first $2s$ elements of $A_{>x}$ into the workspace (without sorting them). Then, it selects the median M of these $2s$ elements in $O(s)$ time and space, using the classic median finding procedure. Knowing the value of the median, the algorithm removes from the workspace the elements that are larger than M, again in $O(s)$ time. Thus, only s elements remain.

In the next step, the algorithm inserts the next batch of s elements from $A_{>x}$ into the workspace. It again finds the median M of the resulting $2s$ elements and removes those elements that are larger than M. The algorithm repeats the latter step until all the elements of $A_{>x}$ have been processed. Clearly, at the end of each step, the s smallest elements of $A_{>x}$ among the ones that have been seen so far reside in the workspace. Consequently, when the pass over all the elements of A is accomplished, the remaining s elements in the workspace are the desired ones.

The number of steps in the algorithm is $O(n/s)$, and each step takes $O(s)$ time; moreover, the overhead time for selecting the elements of $A_{>x}$ is $O(n)$. Thus, the total running time of the algorithm is $O(n)$. By construction, only a workspace of size $2s$ elements of the universe is needed. □

5.4 Selection

The following lemma describes how to efficiently select an element with a given rank from an unsorted list in the limited workspace model. This will be applied in our algorithms in the next chapter to select the edges with rank k in the edge list of the rays.

Lemma 5.2. *Suppose we are given a read-only array A with n pairwise distinct elements from a totally ordered universe, and a number $k \in \{1, \ldots, n\}$. Let s be a given parameter in $\{1, \ldots, n\}$. We can find the kth smallest element in A in $O(\lceil k/s \rceil n)$ time using a workspace of size $2s$ elements of the universe.*

Proof. The algorithm applies Lemma 5.1 to find consequtive batches of s elements of A in their sorted order, until it reaches the $\lceil k/s \rceil^{\text{th}}$ batch, which is actually the batch containing the kth smallest element in A. Then the algorithm selects the kth smallest element of A from that batch.

More precisely, the algorithm proceeds as follows: in the first step, by applying Lemma 5.1, it finds the first batch of s smallest elements in A, and it stores them in the workspace.[2] Then, if $k \leq s$, the algorithm selects the kth smallest element among the elements that reside in the workspace and halts. This takes $O(s)$ time using the classic selection procedure. If $k > s$, the algorithm finds the largest element x in the workspace in $O(s)$ time, and it continues with the next steps.

In each step $i \in \{2, \ldots, \lceil k/s \rceil\}$, using the largest element from the prevalthough step as the input element x in Lemma 5.1, the algorithm finds the set of s smallest elements following x among the elements of A. This is the ith batch of s elements in the sorted sequence of elements of A. The algorithm inserts this set into the workspace. In step $i < \lceil k/s \rceil$, it selects the largest element among the elements in the workspace, and it continues to the next step. In step $i = \lceil k/s \rceil$, the algorithm selects the $(k - (i-1) \times s)^{\text{th}}$ smallest element in the workspace in $O(s)$ time. This element is the desired k^{th} smallest element in A.

Since the number of steps is $\lceil k/s \rceil$, and since each step takes $O(n)$ time, the total running time of the algorithm is $O(\lceil k/s \rceil n)$, and it uses only a workspace of size $2s$ elements of the universe. $\square$

In addition to our simple algorithm in Lemma 5.2, there are several other results on selection in the read-only model; see Table 1 of [CMR14]. In particular, there are $O(n \log \log_s n)$ expected time randomized algorithms for selection using $O(s)$ cells of workspace in the limited workspace model [Cha10, MR96]. Depending on the values of k, s, and n, we will choose one of the latter randomized algorithms or the deterministic algorithm that we gave in Lemma 5.2. In conclusion, in the limited workspace model the running time of selection using $O(s)$ cells of workspace, which we denote by $T_{\text{selection}}(s)$, is $O\big(\min\{\lceil k/s \rceil n, n \log \log_s n\}\big)$ expected time.

[2]Note that to identify the set of s smallest elements in A, the input element x in Lemma 5.1 is not needed.

k-Visibility Region

CHAPTER 6

In this chapter, we investigate the problem of computing the k-visibility region of a given point q inside a given simple polygon P. We present an s-workspace algorithm for this problem which runs in $O\big((k + c)n/s + c \log s\big)$ total deterministic time or in $O\big(cn/s + c \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\}\big)$ total expected time. We also generalize this result for polygons with holes and for sets of non-crossing line segments.

In Chapter 6.1, we clarify the geometry of the k-visibility region via identifying the pieces that form $\partial V_k(P, q)$. In Chapter 6.2, we provide a constant workspace algorithm for computing $V_k(P, q)$. Then, in Chapter 6.3, we extend our constant workspace algorithm to time-space trade-offs for this problem. We provide two algorithms, of which the first one is simple and provides a better understanding of the more involved latter one. In the second algorithm, by optimizing the data structures, we achieve a better time-space trade-off for computing the visibility region of a point in a polygon. Finally, in Chapter 6.4, we show how to modify our algorithm in order to compute the k-visibility region of a point in any polygonal domain.

6.1 Geometry of the k-visibility region

Recall that $r(\theta)$ denotes the ray that is originated from q and makes an angle of θ with the x-axis. Suppose we continuously increase θ from 0 to 2π such that $r(\theta)$ performs an angular sweep. The edge list of $r(\theta)$ only changes when $r(\theta)$ encounters a vertex v of P. This change only involves the two edges incident to v.

At a non-critical vertex v, the edge list of $r(\theta)$ is updated by replacing the incident edge of v, which lies clockwise of $r(\theta)$, with the other incident edge, that lies counter-clockwise of $r(\theta)$. The other edges and their order in the edge list do not change. At a critical vertex v, the edge list of $r(\theta)$ is updated by inserting or removing both incident edges of v, depending on whether v is a start vertex or an end vertex, respectively. The other edges and their order in the edge list are not affected; see Figure 6.1.

Note that, for the chain lists of $r(\theta)$, the situation is almost similar. The only difference is that the chain list of $r(\theta)$ only changes when $r(\theta)$ encounters a critical vertex. In other words, by definition of the chain, if $r(\theta)$ encounters a non-critical vertex v, the chain list of $r(\theta)$ and particularly the chain containing v will not be affected.

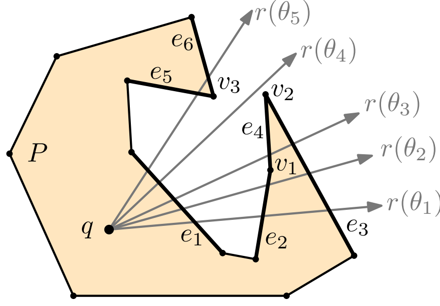

Figure 6.1: An illustration of the changes in the edge list in a counterclockwise rotation of a ray from q. For $r(\theta_1)$ the edge list is e_1, e_2, e_3. The edge list for $r(\theta_2)$ does not differ. The ray $r(\theta_3)$ lies counterclockwise of the non-critical vertex v_1, and its edge list is e_1, e_4, e_3. For $r(\theta_4)$ the edge list is e_1, which is obtained by removing the two incident edges of the end vertex v_2 from the list. For $r(\theta_5)$ the edge list is e_1, e_5, e_6, which is obtained by adding the two incident edges of the start vertex v_3 to the list.

If the ray $r(\theta)$ stabs a vertex v of P, we define the edge (chain) list of $r(\theta)$ to be the edge (chain) list of (i) the ray $r(\theta+\varepsilon)$, if v is a start vertex; (ii) the ray $r(\theta-\varepsilon)$, if v is an end vertex; (iii) the ray $r(\theta-\varepsilon)$, if v is a non-critical vertex; in all cases for a small enough $\varepsilon > 0$. By (i) and (ii) we conclude that, if $r(\theta)$ stabs any critical vertex v, then both incident edges of v are in the edge list of $r(\theta)$. See Figure 6.2 for an illustration.

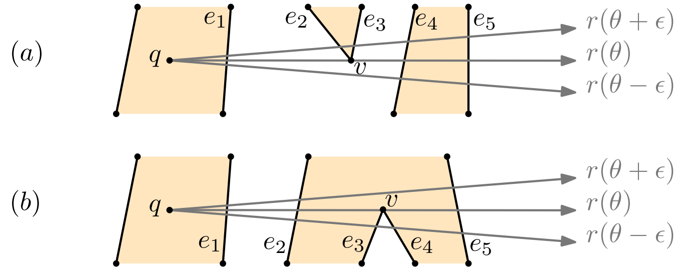

Figure 6.2: A schematic drawing of the edge list of $r(\theta)$ from q through the critical vertex v in a simple polygon. The edge list of $r(\theta)$ in both (a) and (b) is e_1, e_2, e_3, e_4, e_5, which is equal to the edge list of $r(\theta+\epsilon)$ in figure (a) and $r(\theta+\epsilon)$ in figure (b).

Now, we consider the concept of k-visibility for the elements of the chain list. The argument is similar for the edge list. By definition, for any $\theta \in [0, 2\pi)$, only the first $k+1$ elements in the chain list of $r(\theta)$ are k-visible from q in the direction θ. While increasing θ, the chains that are k-visible in direction θ do not change unless $r(\theta)$ encounters a critical vertex v. At that moment the k-visible chains from q are affected only if v is k-visible, which is equivalent to: v does not lie after $e_{k+1}(\theta)$ on $r(\theta)$; or at least one of the incident chains to v is among the first $k+1$ elements in the chain list of $r(\theta)$; or both incident chains to v are among the first $k+2$ elements; see Figure 6.3. The next lemma shows that in this case a segment on $r(\theta)$ may occur on $\partial V_k(P, q)$.

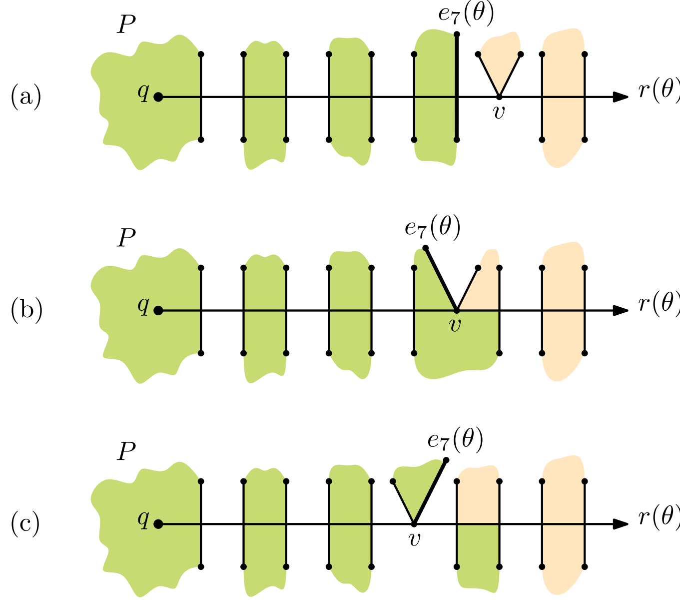

Figure 6.3: A schematic drawing of the edge list of $r(\theta)$ from q through a critical vertex v in a simple polygon P. For $k = 6$, the green regions in P are k-visible from q, and the edge $e_7(\theta)$ represents $e_{k+1}(\theta)$. (a) The critical vertex v lies after $e_7(\theta)$ on $r(\theta)$, thus, it does not affect the list of 6-visible edges in direction θ. (b) One of the two incident edges to v is among the first $k + 1$ elements in the edge list of v. Thus, the lists of k-visible edges differ before and after $r(\theta)$. (c) Both incident edges of v are k-visible in direction θ, thus, it affects the list of k-visible edges on $r(\theta)$.

Lemma 6.1. *Let $\theta \in [0, 2\pi)$ such that $r(\theta)$ stabs a k-visible end or start vertex v. Then, the segment on $r(\theta)$ between $e_{k+2}(\theta)$ and $e_{k+3}(\theta)$ is an edge of $V_k(P, q)$, provided that these two edges exist.*

Proof. Let v be a k-visible end vertex and e_1 and e_2 be the two edges incident to v. We consider the differences between the edge lists of $r(\theta - \epsilon)$ and $r(\theta + \epsilon)$ for a small $\epsilon > 0$.

The edge list of $r(\theta - \epsilon)$ contains e_1 and e_2. However, right after we encounter $r(\theta)$, the edges e_1 and e_2 will not exist any longer in the edge list of $r(\theta + \epsilon)$. Since v is k-visible from q, the edges e_1 and e_2 are among the first $k + 2$ entries in the edge list of $r(\theta)$, which is equal to the edge list of $r(\theta - \epsilon)$. Therefore, after removing e_1 and e_2, the edge with rank $k + 3$ on $r(\theta - \epsilon)$ will have rank $k + 1$ on $r(\theta + \epsilon)$. This means that on the ray $r(\theta + \epsilon)$ the k-visibility region of q extends to the edge $e_{k+3}(\theta - \epsilon) = e_{k+1}(\theta + \epsilon)$; however, on the ray $r(\theta - \epsilon)$ it extends only to $e_{k+1}(\theta - \epsilon)$; see Figure 6.4.

We conclude that the points that lie between $e_{k+1}(\theta)$ and $e_{k+3}(\theta)$ are k-visible from q, if and only if they lie counterclockwise of $r(\theta)$. However, among those points, only the ones between $e_{k+2}(\theta)$ and $e_{k+3}(\theta)$ are in P, since we assume that k is even. Therefore, on $r(\theta)$ the segment between $e_{k+2}(\theta)$ and $e_{k+3}(\theta)$ belongs to $\partial V_k(P, q)$. The situation for a k-visible start vertex v is symmetric. □

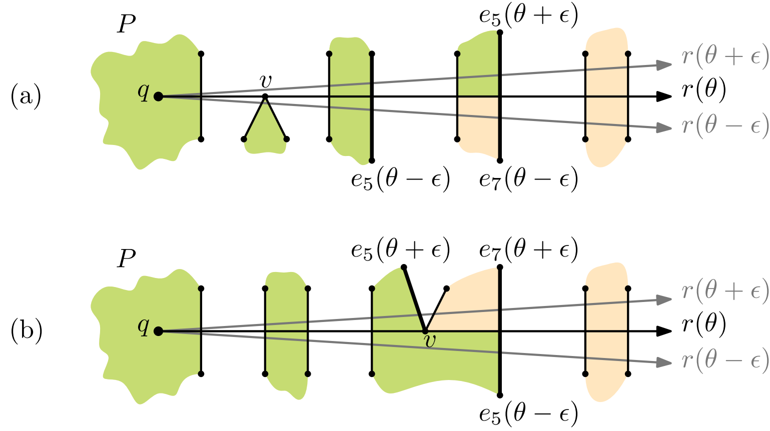

Figure 6.4: A schematic drawing of the ray $r(\theta)$ from q through a critical vertex v in a simple polygon P. For $k = 4$, the green regions in P are k-visible from q. (a) v is an end vertex. The k-visibility region on $r(\theta - \epsilon)$ extends to $e_5(\theta - \epsilon)$ and on $r(\theta + \epsilon)$ extends to $e_5(\theta + \epsilon)$ which is $e_7(\theta - \epsilon)$. (b) v is a start vertex. The k-visibility region on $r(\theta - \epsilon)$ extends to $e_5(\theta - \epsilon)$, which is $e_7(\theta + \epsilon)$, and on $r(\theta + \epsilon)$ extends to $e_5(\theta + \epsilon)$.

Lemma 6.1 leads to the following definition: let $\theta \in [0, 2\pi)$ such that $r(\theta)$ stabs a k-visible end or start vertex v. The segment on $r(\theta)$ between $e_{k+2}(\theta)$ and $e_{k+3}(\theta)$, if these edges exist, is called the *window* of $r(\theta)$; see Figure 6.5.

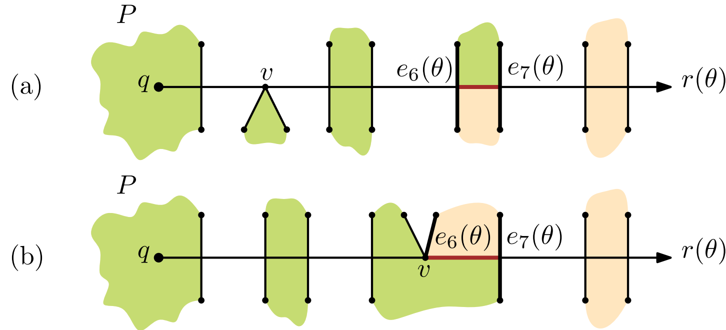

Figure 6.5: A schematic drawing of the ray $r(\theta)$ from q through a critical vertex v in a simple polygon P. For $k = 4$, the green regions in P are k-visible from q. (a) v is an end vertex. (b) v is a start vertex. In both (a) and (b) the segment $r(\theta)$ between $e_6(\theta)$ and $e_7(\theta)$ is the window of $r(\theta)$ and belongs to $\partial V_k(P, q)$.

Observation 6.2. *The k-visibility region $V_k(P, q)$ has $O(n)$ vertices.*

Proof. The boundary $\partial V_k(P, q)$ consists of subchains of ∂P and of windows. Thus, a vertex of $V_k(P, q)$ is either a vertex of P or an endpoint of a window. Since each critical vertex causes at most one window, since each window has two endpoints, and since there are at most n critical vertices, the total number of vertices of $V_k(P, q)$ is $O(n)$. □

6.2 A Constant Workspace Algorithm for $V_k(P, q)$

In this chapter, we present an algorithm that computes $V_k(P, q)$ for the given point q in the simple polygon P, whose vertices are given along its boundary and stored in a read-only array, using only constant cells of workspace.

If the input polygon P has no critical vertex, there is no window, and $V_k(P, q) = P$. This can be checked in $O(n)$ time by a simple scan through the input. Thus, we assume that P has at least one critical vertex v_0. Again, v_0 can be found in $O(n)$ time with a single scan. We choose our coordinate system such that q is the origin and v_0 lies on the positive x-axis. We number the critical vertices of P as $v_0, v_1, \ldots, v_{c-1}$ in the order that the ray $r(\theta)$ encounters them as θ increases. Let θ_i be the angle for v_i. We simplify our notation and write $r(i)$ instead of $r(\theta_i)$, and we denote by $e_j(i)$ the edge which has rank j on $r(i)$. We also use $w(i)$ to refer to the window on $r(i)$.

The main idea of our algorithm is as follows: first we find $e_{k+1}(i)$ on each ray $r(i)$. Then we report all the intersecting edges of $r(i)$ that lie before $e_{k+1}(i)$ on $r(i)$. Moreover, we identify the window $w(i)$ and we report it. The main key to efficiently find $e_{k+1}(i)$ on a ray $r(i)$ is to use the edge $e_{k+1}(i-1)$. This is possible due to the properties of the k-visibility region. The following theorem provides this algorithm.

Theorem 6.3. *Suppose we are given a simple polygon P with n vertices, stored in a read-only array, and a point q in P. Let k be a parameter in $\{0, \ldots, n-1\}$. We can report the k-visibility region of q in P in $O(kn + cn)$ time using $O(1)$ cells of workspace, where c is the number of critical vertices of P with respect to q.*

Proof. In step 0, we start with the ray $r(0)$, and we perform a simple *selection* subroutine to find the intersecting edge on $r(0)$ with rank k as follows: we scan the input $k+1$ times, and in each scan of P, we find the next intersecting edge of $r(0)$ until $e_{k+1}(0)$. The running time of this selection subroutine that uses $O(1)$ cells of workspace is $T_{\text{selection}}(1) = O(kn)$.

By comparing the position of $e_{k+1}(0)$ and v_0 on $r(0)$, we determine if v_0 is k-visible. If so, we report the window $w(0)$ (if it exists), as given by Lemma 6.1. Since $w(0)$ is defined by $e_{k+2}(0)$ and $e_{k+3}(0)$, we can find $w(0)$ in two more scans over the input using the edge $e_{k+1}(0)$

Next in step 1, we find v_1 by a single scan of the input polygon P. Then, we determine $e_{k+1}(1)$ in $O(n)$ time by using $e_{k+1}(0)$ as a starting point: we know that if v_0 is an end vertex, the two incident chains of v_0 disappear in the chain list of $r(1)$. Furthermore, if v_1 is a start vertex, the two incident chains of v_1 appear in the chain list of $r(1)$. All the other chains are not affected, and they intersect $r(0)$ and $r(1)$ in the same order. Using this, we first find the edge e' that has rank $k+1$ on the ray $r(\theta_0 + \varepsilon)$ just after $r(0)$. Depending on the type and position of v_0, the edge e' is either $e_{k+1}(0)$ or $e_{k+3}(0)$, and it can be found in $O(n)$ time. Then, by scanning ∂P starting from e', we find the edge e'' on the chain of e' that intersects the ray $r(\theta_1 - \varepsilon)$ just before $r(1)$, again in $O(n)$ time. Depending on the type and position of v_1, the edge e'' is either $e_{k+1}(1)$ or $e_{k+3}(1)$. Thus, we can find $e_{k+1}(1)$ using e'' in $O(n)$ time; see Figure 6.6.

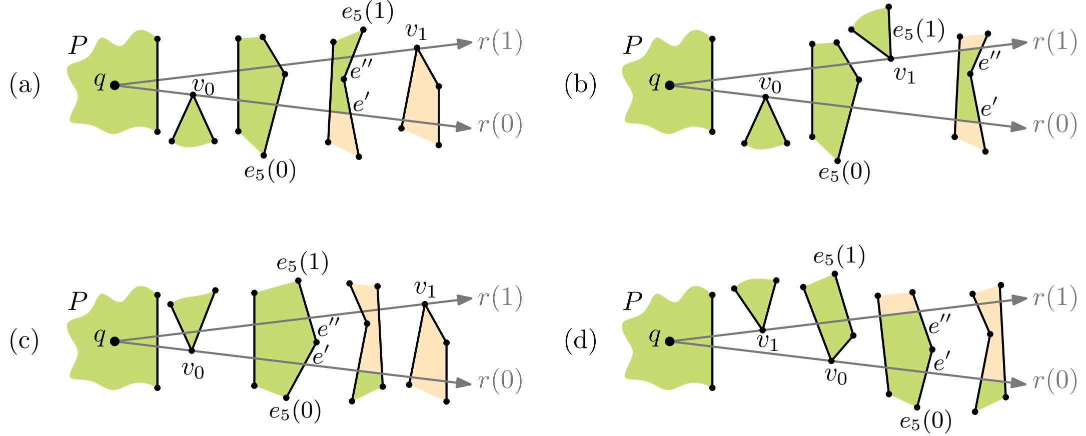

Figure 6.6: For the two consecutive rays $r(0)$ and $r(1)$ through the end or start vertices v_0 and v_1, respectively; four cases of going from $r(0)$ to $r(1)$ are shown. For $k = 4$ the green regions in P are k-visible from q . The edge e' has rank $k+1$ on the ray $e_5(\theta_0+\epsilon)$, and the chain of e' intersects $r(\theta_1-\epsilon)$ at the edge e''. (a) Using $e_5(0)$ we find $e' = e_7(0)$. Then, by walking on the chain containing e', we find e'' which is the desired edge $e_5(1)$. (b) Using $e_5(0)$ we find $e' = e_7(0)$. Then, by walking on the chain containing e', we find e'' which is $e_7(1)$. Then using $e_7(1)$ we find $e_5(1)$. (c) The edge $e' = e_5(0)$. By walking on the chain containing e', we find e'' which is the desired edge $e_5(1)$. (d) The edge $e' = e_5(0)$. By walking on the chain containing e', we find e'' which is $e_7(1)$. Then using $e_7(1)$ we find $e_5(1)$.

After finding $e_{k+1}(1)$, we compare its position with v_1. If v_1 is k-visible, we report the window of $r(1)$ in $O(n)$ time, as described above. Finally, by scanning ∂P, we report the subchains of $\partial V_k(P, q)$ which lie in the counterclockwise $cone(0, 1)$ between $r(0)$ and $r(1)$. More precisely, we walk along ∂P in counterclockwise direction. Whenever we enter $cone(0, 1)$ by intersecting $r(0)$ or $r(1)$, we check whether the intersection occurs at or before $e_{k+1}(0)$ or $e_{k+1}(1)$, respectively. If so, we report the subchain of ∂P until we leave $cone(0, 1)$ again.

In each step $i \in \{2, \dots, c-1\}$, we repeat the same procedure as in step 1: we use $e_{k+1}(i-1)$ to identify $e_{k+1}(i)$ in $O(n)$ time. Now having $e_{k+1}(i)$, by a single scan of ∂P, we find the window $w(i)$ (if it exist) and we report it. Furthermore, we report all the k-visible subchains of ∂P which lie in $cone(i-1, i)$, by comparing the position of the subchains with $e_{k+1}(i-1)$ and $e_{k+1}(i)$, in $O(n)$ time.[1] See Algorithm 6.1.

Regarding the running time, the selection subroutine in step 0 takes $O(kn)$ time. After that, each step $i \in \{1, \dots, c-1\}$ takes $O(n)$ time. Thus, the total running time of the algorithm is $O(kn + cn)$. By construction, the algorithm uses only $O(1)$ cells of workspace. Thus, the claim follows. □

[1]Here and in the following algorithms, if there are less than $k+1$ intersecting edges on $r(i-1)$, we store the last intersecting edge together with its rank. We use this edge instead of $e_{k+1}(i-1)$ to find $e_{k+1}(i)$ or to find the last intersecting edge of $r(i)$ with its rank.

Algorithm 6.1: Computing $V_k(P, q)$ using $O(1)$ cells of workspace

input: Simple polygon P, point $q \in P$, $k \in \mathbb{N}$
output: The boundary of the k-visibility region of q in P, $\partial V_k(P, q)$

```
1  if P has no critical vertex then
2  |  return ∂P
3  v_0 ← a critical vertex of P
4  Find e_{k+1}(0) using selection subroutine
5  if v_0 lies on or before e_{k+1}(0) on r(0) then
6  |  Find and report the window w(0) (if it exists)
7  i ← 1
8  repeat
9  |  v_i ← the next counterclockwise critical vertex after v_{i−1}
10 |  Find e_{k+1}(i) using e_{k+1}(i − 1)
11 |  if v_i lies on or before e_{k+1}(i) on r(i) then
12 |  |  Find and report the window w(i) (if it exists)
13 |  Report the subchains of ∂V_k(P, q) which lie in cone(i − 1, i)
14 |  i ← i + 1
15 until v_i = v_1
```

6.3 Time-Space Trade-Offs for $V_k(P, q)$

In this chapter, we assume that we have $O(s)$ cells of workspace at our disposal. We show how to exploit this additional workspace to improve the running time for computing the k-visibility region of q in a simple polygon P. We describe two algorithms. The first algorithm is simpler, and it is meant to illustrate the main idea behind the trade-off. Our main contribution is in the second algorithm, which is a bit more complicated but achieves a better running time.

In the first algorithm, we process all the vertices of P in angular order in contiguous batches of size s. In each iteration, we find the next batch of s vertices, and using the edge list of the last processed vertex, we construct a data structure that serves us identifying the windows of the batch. Using the windows, we report the subchains of $\partial V_k(P, q)$ that lie between the first and the last ray of the batch.[2]

In the second algorithm, we improve the running time by skipping the non-critical vertices. Specifically, in each iteration, we find the next batch of s contiguous critical vertices in angular order. Then, as before, we construct a data structure for finding the windows and the k-visible subchains of ∂P. However, in this algorithm, we need a more involved approach in order to maintain the data structure.

[2]We emphasize that $\partial V_k(P, q)$ is not necessarily reported in order, but we ensure that the union of the reported line segments constitutes the boundary of the k-visibility region.

6.3.1 Processing All the Vertices of the Polygon

Suppose we are given a vertex v_0 of P, as well as the edge $e_{k+1}(0)$ with rank $k+1$ on the ray $r(0)$ from q through v_0. Let $v_1, \ldots, v_s$ be the sorted list of s vertices of P that follows v_0 in a counterclockwise angular sweep around q. Let $r(1), \ldots, r(s)$ be the rays from q through the vertices $v_1, \ldots, v_s$, respectively. Let $cone(0, s)$ be the counterclockwise cone between $r(0)$ and $r(s)$. The next lemma shows how to exploit the edge $e_{k+1}(0)$ in order to find the subchains of $\partial V_k(P, q)$ that lie in $cone(0, s)$.

First, we fix our terminology. For an edge e of P, the $0s$*-segment* of e is the subsegment of e that lies in $cone(0, s)$. Since the k-visibility along ∂P changes only at window endpoints, the k-visibility along a $0s$-segment does not change unless an endpoint of a window lies on it. In other words, if a $0s$-segment does not contain an endpoint of a window, then either the whole $0s$-segment is k-visible from q or the whole $0s$-segment is not k-visible. We will use this to report the k-visible subchains of ∂P that lie in $cone(0, s)$, more details follow.

Lemma 6.4. *Suppose we are given a vertex v_0 of the simple polygon P, as well as the edge $e_{k+1}(0)$ with rank $k+1$ on the ray $r(0)$ from q. We can report the subchains of $\partial V_k(P, q)$ that lie in* cone$(0, s)$ *in $O(n + s \log s)$ time using $O(s)$ cells of workspace.*

Proof. We consider the angles of the vertices of P as the elements of the input array and the angle of v_0 as the element x in Lemma 5.1. We apply Lemma 5.1 to find, in $O(n)$ time, the batch of s vertices with the s smallest angles after v_0. We store these vertices in the workspace in their sorted order, this takes an additional $O(s \log s)$ time for sorting. Let $V_{[1,s]} = v_1, \ldots, v_s$ denote the sequence of these vertices in the sorted order. Furthermore, let $r(1), \ldots, r(s)$ denote the rays from q through the vertices $v_1, \ldots, v_s$, respectively.

Next, we again apply Lemma 5.1, four times, in order to find the at most $4s + 1$ intersecting edges with ranks in $\{k + 1 - 2s, \ldots, k + 1 + 2s\}$ on $r(0)$. To do this, while scanning edges of P, we consider all the edges that intersect $r(0)$ as the elements of the input array in Lemma 5.1. The distance of q to the intersection points of the edges with $r(0)$ is the value of these elements. Lemma 5.1 can be applied because we have $e_{k+1}(0)$ at hand, which plays the role of x in the lemma. We insert the edges with ranks in $\{k+1-2s, \ldots, k+1+2s\}$ on $r(0)$ into a balanced binary search tree T, sorted according to their ranks. This takes $O(n + s \log s)$ time.

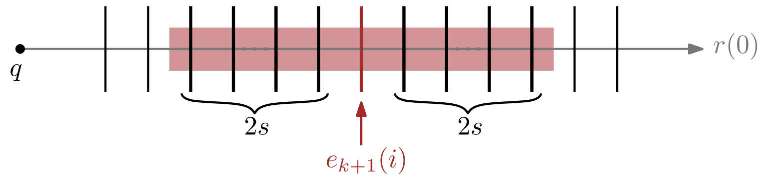

Figure 6.7: For the ray $r(0)$ from q, the edges of P that intersects $r(0)$ are shown. Among them the edges with rank in $\{k + 1 - 2s, \ldots, k + 1 + 2s\}$ are marked. These edges are the elements of T in sorted order by their rank on $r(0)$.

The chains that contain the edges in T are candidates for having rank $k+1$ on the next s rays $r(1), \ldots, r(s)$. This is because, as we have seen in Chapter 6.2 (see Figure 6.6), if $e_{k+1}(i)$ belongs to the edge list of $r(i-1)$, in this edge list, there is at most one edge between $e_{k+1}(i-1)$ and the edge of $e_{k+1}(i)$. Therefore, by induction, if the chain containing $e_{k+1}(i)$ appears in the chain list of $r(0)$, then in this chain list, there are at most $2i-1$ chains between $e_{k+1}(0)$ and the chain of $e_{k+1}(i)$. Thus, T covers all the intersecting chains of $r(0)$ that can possibly be the rank $k+1$ chain on one of the next s rays.

Now, we process a vertex of P per step. In step 0, we check if v_0 is a k-visible critical vertex, i.e., whether it is a critical vertex with respect to q and does not occur after the edge $e_{k+1}(0)$ on $r(0)$. If so, we report the window $w(0)$ which is defined by $e_{k+2}(0)$ and $e_{k+3}(0)$ (if they exist).

In step 1, we go to the next vertex v_1, and we update T depending on the types of v_0 and v_1: if v_0 is a non-critical vertex, we may need to exchange one incident edge of v_0 with another in T; if v_0 is an end vertex, we may need to remove its incident edges from T (if one of the incident edges has rank $k+1$, we need to remember its position in T); and if v_1 is a start vertex, we may need to insert its incident edges into T. In all other cases, no action is necessary. The insertion and/or deletion is performed only for the edges whose ranks on $r(1)$ are between the smallest and the largest rank in T (with respect to $r(1)$). Since only constantly many edges have to be inserted or deleted from T, the update of T takes $O(\log s)$ time. Afterwards, we can find $e_{k+1}(1)$ using the position of $e_{k+1}(0)$ or its neighbors in T, as explained in Theorem 6.3. If v_1 is a k-visible critical vertex, we find the window $w(1)$ (if it exists). This is done in $O(1)$ time by taking the next two entries in T after $e_{k+1}(1)$. See Figure 6.8 for an illustration.

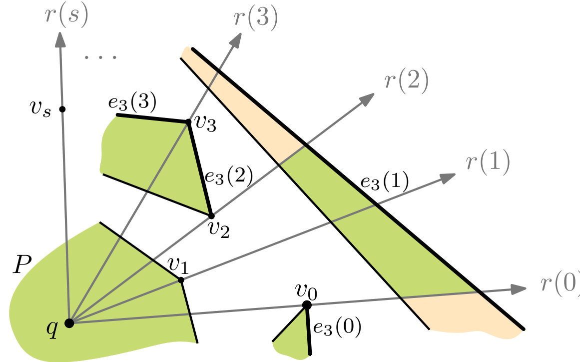

Figure 6.8: A schematic drawing of the 2-visibility region of q in a simple polygon P. A vertex v_0 of P and the first batch $v_1, \ldots, v_s$ of s vertices following v_0 sorted by their angles are shown. Assume T contains $4s+1$ intersecting edges of $r(0)$. The edge $e_3(1)$ is the second neighbor to the right of $e_3(0)$ in T because v_0 is an end vertex and v_1 is non-critical. The edge $e_2(3)$, which is inserted in T before processing v_2, is the second neighbor to the left of $e_1(3)$ in T. The edge $e_2(3)$ is exchanged with $e_3(3)$ after processing v_3 because v_3 is a non-critical vertex.

In each step $i = 2, \dots, s$, we repeat the same procedure as in step 1: first, based on the types of v_{i-1} and v_i, we update T. Then, using T and the edge $e_{k+1}(i-1)$ from the previous step, we determine the edge $e_{k+1}(i)$. After that we determine if v_i is k-visible or not, and if so, using T we find and report the window $w(i)$ (if it exists). Whenever we find a window, we insert its endpoints into a balanced binary search tree W. This takes $O(\log s)$ time per window. The endpoints of windows in W are sorted according to their counterclockwise order along ∂P. Furthermore, we store in the workspace the sequence $E = e_0(k+1), \dots, e_s(k+1)$ of edges with rank $k+1$ on $r(0), \dots, r(s)$, respectively.

For reporting the k-visible subchains of ∂P that lie in $cone(0, s)$, the counterclockwise cone between $r(0)$ and $r(s)$, we use W and E: we can walk along ∂P and, simultaneously, along the window endpoints in W. For each edge e of P, we check if the endpoints of the $0s$-segment of e are k-visible or not. Using E, this can be done in $O(1)$ time. Moreover, with the help of the parallel traversal of W, we find the window endpoints that lie on e. This takes $O(|w_e|)$ time, where $|w_e|$ is the number of window endpoints on e. With this information, we can report the k-visible subsegments of the $0s$-segment of e. Since there are $O(n)$ window endpoints by Observation 6.2, and since we check each window endpoint once, it follows that we need $O(n)$ time to report the k-visible subchains of ∂P that lie in $cone(0, s)$; see Figure 6.9

The first part of the algorithm takes $O(n + s \log s)$ time to find the sorted sequence of the vertices $V_{[1,s]}$ and the sorted list of the candidate intersecting edges T. After that, the algorithm processes the vertices in $V_{[1,s]}$ in s steps, where each step takes $O(\log s)$ time, making it a total of $O(s \log s)$ time. The time for constructing W is also $O(s \log s)$ since there are at most $s + 1$ windows on the rays $r(0), \dots, r(s)$. Finally, reporting the k-visible subchains of ∂P in $cone(0, s)$ using W and E takes $O(n)$ time. Overall, the running time of the algorithm is $O(n + s \log s)$. The space requirement for storing $V_{[1,s]}$, T, W and E is $O(s)$ cells of workspace. □

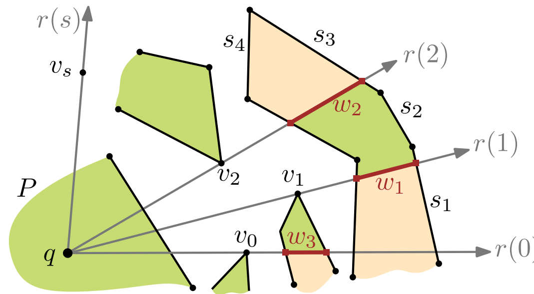

Figure 6.9: A schematic drawing of the 2-visibility region in $cone(0, s)$. Assume W contains the endpoints of the windows w_1, w_2, w_3 sorted on ∂P. On a counterclockwise walk on the $0s$-segment s_1, the sub-segment of s_1 before encountering the endpoint of w_1 is non-visible and after that is k-visible. The k-visibility along s_2 does not change, since there is no window endpoint on it. For s_3, we start with a k-visible sub-segment until we reach an endpoint of w_2 and then the rest of s_3 is non-visible. Both endpoints of s_4 are non-visible and there is no window endpoint on it. Thus, s_4 is completely non-visible.

The algorithm in Lemma 6.4 demonstrates that having a relevant sub-sequence of the edge list of a ray accelerates finding the rank $k + 1$ edge on the following rays. We extend this idea for all the rays from q through the vertices of P. To do this, we process the rays in batches of size s using Lemma 6.4. The following theorem explains the details of our algorithm.

Theorem 6.5. *Suppose we are given a simple polygon P with n vertices, stored in a read-only array, and a point $q \in P$. Let $k \in \{0, \ldots, n-1\}$ and $s \in \{1, \ldots, n\}$ be two parameters. We can report the k-visibility region of q in P in $O(n^2/s + n \log s)$ time using $O(s)$ cells of workspace.*

Proof. We take a vertex v_0 of P. We choose our coordinate system such that q is the origin and such that v_0 lies on the positive x-axis. Let $v_1, \ldots, v_{n-1}$ be the sequence of vertices of P sorted by their angle, and let $r(1), \ldots, r(n-1)$ be the rays from q through the vertices $v_1, \ldots, v_{n-1}$, respectively.

We use Lemma 5.2 to select the edge with rank $k + 1$ on the ray $r(0)$. To do this, we consider all the edges of P as the input elements; however, we filter the edges that intersect $r(0)$. The distance of their intersection point to q is the value of each of these elements. Thus, we can select the intersecting edge of $r(0)$ which has the $(k+1)$-smallest distance to q in $O(\lceil k/s \rceil)$ time using $O(s)$ cells of workspace. This edge has rank $k + 1$ on $r(0)$, and thus, it is $e_{k+1}(0)$. Recall that if there are less than $k + 1$ intersecting edges on $r(0)$, we store the last intersecting edge together with its rank, and this edge will be used instead of $e_{k+1}(0)$.[3]

Now, using v_0 and $e_{k+1}(0)$, we apply Lemma 6.4 to find and process the batch of the s vertices $v_1, \ldots, v_s$ following v_0, as well as to report the subchains of $\partial V_k(P, q)$ that lie in $cone(0, s)$. At the end of this procedure, we also have the edge with rank $k+1$ on the ray $r(s)$ at hand.[4] Thus, in the next iteration, we similarly use v_s and $e_{k+1}(s)$ to apply Lemma 6.4 for processing the next batch of s vertices after v_s. This also reports the subchains of $V_k(P, q)$ that lie in $cone(s, 2s)$, which is the counterclockwise cone between $r(s)$ and $r(2s)$.

We repeat this procedure for $\lfloor n/s \rfloor$ iterations, and in each iteration, we consider the next batch of s vertices of P until all the vertices are processed. If n is not divisible by s, the last batch wraps around, taking the indices modulo n, but we report only the part of $\partial V_k(P, q)$ before the ray $r(n) = r(0)$; see Algorithm 6.2.

Overall, it takes $O(n + s \log s)$ time to process a batch of s vertices of P. Due to number of vertices of P, we have $O(n/s)$ batches. Moreover, for the ray $r(0)$, we run the selection subroutine using $O(s)$ cells of workspace whose running time is dominated by the other terms. Thus, the total running time of the algorithm is $O(n^2/s + n \log s)$. The space requirement is immediate. □

[3] The same comment applies to the other rays.

[4] However, the search tree T cannot be used to process the edges of the next batch. This is because T does not necessarily contain any adjacent neighbor of $e_s(k + 1)$ in the edge list of $r(s)$.

Algorithm 6.2: Computing $V_k(P,q)$ using $O(s)$ cells of workspace

input: Simple polygon P, point $q \in P$, $0 \leq k < n$, $1 \leq s \leq n$
output: The boundary of k-visibility region of q in P, $\partial V_k(P,q)$

```
1  v_0 ← a vertex of P
2  E ← ⟨e_{k+1}(0)⟩ [by Lemma 5.2]
3  T, W ← an empty balanced binary search tree
4  i ← 0
5  repeat
6      v_{i+1}, ..., v_{i+s} ← vertices following v_i in angular order [by Lemma 5.1]
7      T ← at most 4s + 1 edges with rank in {k+1−2s, ..., k+1+2s} on r(i)
8      for j = i to i + s − 1 do
9          if v_j lies on or before e_{k+1}(j) on r(j) then
10             Report the window of r(j) (if it exists) [using T]
11             Insert the endpoints of the window into W [sorted by position on ∂P]
12         Update T according to the types of v_j and v_{j+1}
13         Append e_{k+1}(j+1) to E [find it using e_{k+1}(j) and T]
14     Report subchains of ∂V_k(P,q) in cone(i, min{i+s, n}) [using W and E]
15     i ← i + s
16 until i ≥ n
```

6.3.2 Processing Only the Critical Vertices of the Polygon

In this algorithm, as before, we process the vertices of P in batches; however, here we focus only on the critical vertices. This is reasonable since there is no window on the rays through non-critical vertices; therefore, we do not need to find the edge of rank $k+1$ on those rays.

Furthermore, the new algorithm maintains the chain list of the current ray through a critical vertex, instead of its edge list, i.e., for each such a chain, the data structure contains some edge on that chain as its reference, and not necessarily the edge that intersects the ray. The reason for working with the chain list is that updating the whole edge list is time consuming. More precisely, there are possibly non-critical vertices that lie between two consecutive rays, and such non-critical vertices change the edge list of those rays but do not affect on the chain list. Therefore, these changes do not need to be reflected in the new data structure, unless we need the specific edge that intersects a ray, and in this case, we find that edge by walking along its corresponding chain.

Let $v_0, \ldots, v_{c-1}$ denote the sequence of critical vertices of P that are sorted by their angle, where c is the number of critical vertices of P with respect to $q \in P$. Let $r(0), \ldots, r(c-1)$ be the rays from q through the critical vertices $v_0, \ldots, v_{c-1}$, respectively. The following lemma explains how to process a batch of s consecutive critical vertices $v_1, \ldots, v_s$, provided that the edge with rank $k+1$ on $r(0)$ is given. The algorithm in this lemma also reports the subchains of $\partial V_k(P,q)$ that lie in $cone(0,s)$, which is the counterclockwise cone between the rays $r(0)$ and $r(s)$.

Lemma 6.6. *Suppose we are given a critical vertex v_0 of a simple polygon P, as well as the edge $e_{k+1}(0)$ with rank $k+1$ on the ray $r(0)$ from q. We can report the subchains of $\partial V_k(P, q)$ that lie in cone$(0, s)$ in $O(n + s \log s)$ time using $O(s)$ cells of workspace.*

Proof. Consider the angle of the critical vertices of P as the input elements in Lemma 5.1. Using this lemma and a traditional sorting algorithm, we compute $V_{[1,s]} = v_1, \ldots, v_s$, the list of s critical vertices of P after v_0, sorted according to their angle. This takes $O(n + s \log s)$ time and $O(s)$ cells of workspace.

Next, we use Lemma 5.1 with the intersecting edges of $r(0)$ as the elements of the input array and the edge $e_{k+1}(0)$ as the element x. We find the at most $4s+1$ intersecting edges of $r(0)$ with rank in $\{k+1-2s, \ldots, k+1+2s\}$, and we insert them into a balanced binary search tree T, ordered according to their rank on $r(0)$. This takes $O(n + s \log s)$ time and $O(s)$ cells of workspace.

Then, as in the algorithm of Lemma 6.4, we process one critical vertex in each step. In step 0, we use $e_{k+1}(0)$ and T to find and report $w(0)$ (if it exists). In step 1, we update T according to the types of v_0 and v_1, so that it contains the chain list of $r(1)$. This is done as follows: if v_0 is an end vertex, and if its incident edges are in T, we remove those edges from T; and if v_1 is a start vertex, we insert the two incident edges (chains) of v_1 into T, if their ranks on $r(1)$ are in the interval of the ranks of the chains in T. For finding the rank of the chains of v_1 on $r(1)$, using a binary search, we compare the positions of those chains and the elements of T on $r(1)$. Whenever a comparison needs to be performed with respect to an edge e in T, we check whether e intersects r_1. If not, we walk along the chain of e until we find such an edge; see Figure 6.11. Thus, it takes $O(\log s + n'_1)$ time to update T, where n'_1 denotes the number of non-critical vertices that are traversed to find the correct edges for comparisons during the update operations.

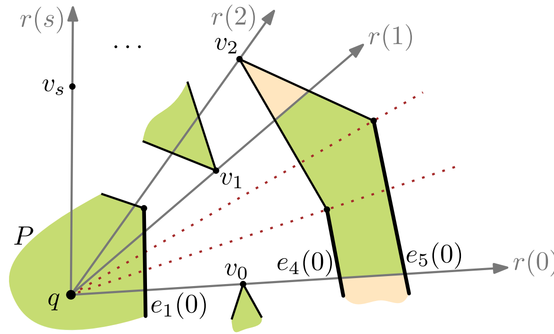

Figure 6.10: A schematic drawing of the 2-visibility region of q in a simple polygon P. A critical vertex v_0 and the first batch $v_1, \ldots, v_s$ of s critical vertices following v_0 sorted by their angles are shown. Assume T contains $4s+1$ intersection edges of v_0. The edges $e_4(0)$ and $e_5(0)$, in contrast to $e_1(0)$, do not intersect $r(1)$. While inserting the chains of the start vertex v_1 into T, if a comparison involving $e_4(0)$ or $e_5(0)$ needs to be performed, we first walk along the chain of those edges to find the edge that intersects $r(1)$.

Now, T contains at most $4s+1$ intersecting chains of $r(1)$. To find $e_{k+1}(1)$, we walk along either the chain of $e_{k+1}(0)$ or its neighbors in T, as we have explained in Theorem 6.3. By having $e_{k+1}(1)$ and T, we find and report the window $w(1)$ (if it exists). This is done again by following the chains that define $w(1)$ if the corresponding edge of those chains in T does not intersect $r(1)$ yet. Finding $e_{k+1}(1)$ and $w(1)$ takes $O(n''_1)$ time, where n''_1 is the number of non-critical vertices that are traversed.

In step $i = \{2, \ldots, c-1\}$, we repeat the same procedure as in step 1. We update T regarding the edges that are incident to the critical vertices v_{i-1} and v_i. The only difference is that, if v_{i-1} is an end vertex, checking whether its chains are in T, and identifying their corresponding elements in T are not as straightforward as in step 1. The problem occurs due to the fact that we do not know which edge of each chain has been stored as its reference in T. To resolve this problem, for any chain in T, we additionally store another edge of the chain which is called the *guide edge* and which is defined as follows: for a chain C that intersects $r(0)$, the edge on C that intersects $r(0)$ is a *type* 1 guide edge; and for a chain C that has been inserted into T in one of the steps $2, \ldots, i-1$, the first edge of C is a *type* 2 guide edge.

We store the type 1 guide edges in a list G_1 sorted according to their rank on $r(0)$, i.e., we copy the sorted elements of T in step 0 into G_1. The type 2 guide edges are stored in another list G_2 sorted according to the step in which they have been inserted into T, i.e., the angle of the start vertex of the chain. Therefore, in each step that new edges are inserted into T, those edges will also be added to G_2 in $O(1)$ time. Furthermore, the elements in G_1 and G_2 have cross-pointers to their corresponding chains in T.

To find the corresponding elements of the chains of an end vertex v_{i-1} in T, we walk backward on those chains, until we either find an edge that intersects $r(0)$ (a type 1 guide edge), or we stop at the first edge of that chain (a type 2 guide edge). See Figure 6.11 for an illustration. Then, by a binary search in G_1 or G_2 in $O(\log s)$ time, and using

Figure 6.11: A schematic drawing of $k = 2$-visibility region of q in a simple polygon P. A critical vertex v_0 and the first batch $v_1, \ldots, v_s$ of s critical vertices following v_0 sorted by their angles are shown. Assume T contains $4s+1$ intersection edges of v_0. To remove the incident chains of the end vertex v_2, we need to find the corresponding elements to its chains in T. By a backward walk along its chains, we find a type 2 guide edge that is incident to v_1 and a type 1 guide edge that is $e_4(0)$.

the cross-pointer to the elements in T, we identify the corresponding elements of chains of v_{i-1} in T. Therefore, we remove them from T and also G_1 or G_2, in $O(\log s)$ time. Now, T contains the chain list of v_i and can be used to find $e_{k+1}(i)$ with the help of $e_{k+1}(i-1)$. Finally, we report the window $w(i)$ (if it exist), as in step 1.

While processing the batch, we insert all $e_{k+1}(i)$, $0 \leq i \leq s$, into E. Also, whenever we find and report a window, we insert its endpoints, sorted according to their counter-clockwise order along ∂P, into a balanced binary search tree W, in $O(\log s)$ time. After processing all the vertices of the batch, we use W and E to report the part of $\partial V_k(P, q)$ between $r(0)$ and $r(s)$, as in Lemma 6.4. The only difference is that now we keep track of the visibility of the whole chains between $r(0)$ and $r(s)$ instead of individual edges. As before, this takes $O(n)$ time.

Overall, processing the changes in T takes $O(n' + s \log s)$ total time, where n' is the number of non-critical vertices that lie in $cone(0, s)$. This is dominated by the time $O(n + s \log s)$ that is needed to find $V_{[0,s]}$ and T. The space required is $O(s)$ cells of workspace. Thus, the claim follows. □

In contrast to the algorithm in Lemma 6.4, that processes a cone with s vertices of P, the algorithm in Lemma 6.6 processes a cone that contains s critical vertices, and possibly an unlimited number of non-critical vertices. By cleverly dealing with the effects of the non-critical vertices, it obtains the same running time. Therefore, if we plug in this new lemma in our main algorithm, we will achieve a better time-space trade-off. The following theorem shows how this is done.

Theorem 6.7. *Suppose we are given a simple polygon P with n vertices, stored in a read-only array, and a point $q \in P$. Let $k \in \{0, \ldots, n-1\}$ and $s \in \{1, \ldots, n\}$ be two parameters. We can report the k-visibility region of q in P in total expected time $O(cn/s + c \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ using $O(s)$ cells of workspace. Here, c is the number of critical vertices of P with respect to q.*

Proof. As in Chapter 6.2, if P has no critical vertex, then $V_k(P, q) = P$. This can be checked in $O(n)$ time by a simple scan through the input. Thus, we let v_0 be some critical vertex, and we choose our coordinate system such that q is the origin and such that v_0 lies on the positive x-axis.

Similar to Theorem 6.5, we find $e_{k+1}(0)$ using the selection subroutine (with $O(s)$ cells of workspace) that we have introduced in Chapter 5.4. Now, we apply Lemma 6.6 to process the first batch of s critical vertices in angular order. In each of the subsequent iterations, we again apply Lemma 6.6 to process the next batch of s critical vertices, until all the critical vertices are processed; see Algorithm 6.3.

Since in each iteration s distinct critical vertices of P are processed, the number of iterations is $O(c/s)$. By Lemma 6.6, each iteration takes $O(n + s \log s)$ time. Thus, we get a total of $O(cn/s + c \log s)$ time, in addition to $T_{\text{selection}}(s)$ for selecting $e_{k+1}(0)$ at the beginning of the algorithm. Depending on the value of n, s and k, we take a selection subroutine which gives us the best running time; see Chapter 5.4. Overall, the running time of the algorithm is $O(cn/s + c \log s + \lceil k/s \rceil n)$ total deterministic time or $O(cn/s + c \log s + n \log \log_s n)$ total expected time. Thus, the claim follows. □

Algorithm 6.3: Computing $\partial V_k(P, q)$ using $O(s)$ cells of workspace

input: Simple polygon P, point $q \in P$, $0 \le k < n$, $1 \le s \le n$

output: The boundary of k-visibility region of q in P, $\partial V_k(P, q)$

```
1  v_0 ← a critical vertex of P
2  E ← ⟨e_{k+1}(0)⟩ [by Lemma 5.2]
3  T, W ← an empty balanced binary search tree
4  G_1, G_2 ← ⟨⟩
5  i ← 0
6  repeat
7      v_{i+1}, ..., v_{i+s} ← critical vertices following v_i in angular order [by Lemma 5.1]
8      T ← at most 4s + 1 edges with rank in {k + 1 − 2s, ..., k + 1 + 2s} on r(i)
9      G_1 ← the sorted elements of T
10     for j = i to i + s − 1 do
11         if v_j lies on or before e_{k+1}(j) on r(j) then
12             Report the window of r(j) (if it exists) [using T]
13             Insert the endpoints of the window into W [sorted by position on ∂P]
14         if v_j is an end vertex then
15             Find the guide edges of chains of v_j [by walking along the chains]
16             if the guide edges exist in G_1 or G_2 then
17                 Use their cross-pointers to find the corresponding elements in T
18                 Remove those corresponding elements from T and G_1 or G_2
19         if v_{j+1} is a start vertex then
20             repeat
21                 Take the next edge e of T [in a binary search fashion]
22                 if e does not intersect r(j + 1) then
23                     Walk along the chain of e to find e' that intersects r(j + 1)
24                     Exchange e with e' in T
25                 Compare the position of e with v_{j+1} on r(j + 1)
26             until rank of the chains of v_{j+1} on r(j + 1) are determined
27             if the rank of the chains of v_{j+1} are valid for T then
28                 Insert the chains of v_{j+1} into T according to their rank on r(j + 1)
29                 Append the chains of v_{j+1} to G_2
30         Find e_{k+1}(j + 1) by walking along e_{k+1}(j) or its neighbors in T
31         Append e_{k+1}(j + 1) to E
32     Report subchains of ∂V_k(P, q) in cone(i, min{i + s, n}) [using W and E]
33     i ← i + s
34 until i ≥ n
```

6.4 Variants and Extensions

Our results can be extended in several ways, for example: computing the k-visibility region of a point q inside a polygon P, where P may have $h \geq 0$ holes; and computing the k-visibility region of a point q in a planar arrangement of n non-crossing segments inside a bounding box.[5] In other words, computing the k-visible sub-segments of a planar set S of n non-crossing segments, from a point q.

Concerning the first extension, all the arguments in the algorithms for simple polygons also hold for the polygons with holes The only noteworthy issue is the use of ∂P to report the k-visible subchains of ∂P in each cone. In the case of polygons with holes, after walking on the outer part of ∂P, we walk on the boundaries of the holes one by one and we apply the same procedures for them. If there is no window on the boundary of a hole, then it is either completely k-visible or completely non-k-visible. For such a hole, we check if it is k-visible and, if so, we report it completely; see Figure 6.12. This leads to the following corollary.

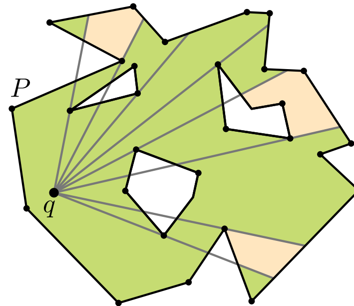

Figure 6.12: For a polygon P with 3 holes and the point $q \in P$ the 2-visibility region of q in P is shown in green.

Corollary 6.8. *Suppose we are given a polygon P with $h \geq 0$ holes and n vertices, stored in a read-only array, and a point $q \in P$. Let $k \in \{0, \ldots, n-1\}$ and $s \in \{1, \ldots, n\}$ be two parameters. We can report the k-visibility region of q in P in total expected time $O(cn/s + c \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ using $O(s)$ cells of workspace. Here, c is the number of critical vertices of P with respect to q.*

Concerning the second problem, for a planar arrangement of n non-crossing segments inside a bounding box, the output consists of the k-visible parts of the segments. All the segments endpoints are critical vertices and should be processed. In the parts of the algorithm where a walk on the boundary is needed, a sequential scan of the input leads to similar results. Similarly, there may be some segments with no window endpoints. For these, we only need to check visibility of an endpoint to decide whether they are completely k-visible or completely non-k-visible; see Figure 6.13. This leads to the following corollary.

[5]The bounding box is only for bounding the k-visibility region.

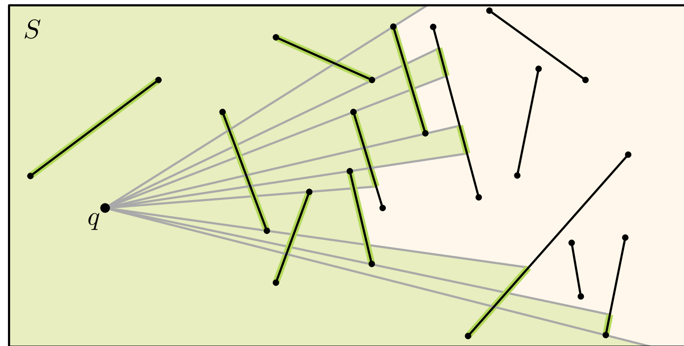

Figure 6.13: For a set of line segments S in a bounding box, the 2-visible subsets of segments in S are marked in green.

Corollary 6.9. *Suppose we are given a set S of n non-crossing planar segments in a bounding box B that is stored in a read-only array, as well as a point $q \in B$. Let $k \in \{0, \dots, n-1\}$ and $s \in \{1, \dots, n\}$ be two parameters. We can report the k-visible subsets of segments in S from q in $O(n^2/s + n \log s)$ total deterministic time using $O(s)$ cells of workspace.*

Conclusions

CHAPTER 7

In this thesis, we have considered algorithms that use a workspace of small size proportional to the size of the read-only input. This kind of algorithms does not only provide an interesting trade-off between running time and memory needed, but it is also very useful in portable devices where hardware constraints are present. We have particularly focused on three geometric problems in the limited workspace model, an area that has become more popular during the last decade.

The algorithms that we have presented are typically deterministic, but there are also results that use randomization. Randomized algorithms in the limited workspace model may use an unlimited stream of random bits. However, these bits cannot be accessed arbitrarily, i.e., if the algorithm wishes to revisit previous random bits, it needs to store them in its workspace.[1]

In the preceding chapters, we have seen several approaches to deal with the space constraints as well as many different techniques to exploit the additional workspace and obtain a time-space trade-off. However, we believe that these results are only a few steps in this direction, and that other novel techniques are needed to achieve efficient time-space trade-offs for many other interesting geometric problems.

Voronoi diagrams. We have obtained a non-trivial time-space trade-off for computing the family of higher-order Voronoi diagrams of order 1 to K, for $K \in O(\sqrt{s})$, as well as, the nearest site Voronoi diagram and the farthest site Voronoi diagram. For the nearest site Voronoi diagram and the farthest site Voronoi diagram, our running times come close to the sorting lower bound. More precisely, Beame shows that the time-space product for sorting is $\Omega(n^2)$ [Bea91]. Although improvement by a logarithmic factor may be possible, the gap between upper and lower bounds is very small.

There is a much larger gap for general higher-order Voronoi diagrams. In fact, we are not aware of any lower bounds (beyond the sorting lower bound). In particular, it would be interesting to have a bound in terms of the order of the diagram. It seems likely that at least $\Omega(n^2K^2/s)$ steps are needed to find the family of all Voronoi diagrams of order up to K for a given n-point set using s cells of workspace.

[1]Refer to Goldreich's book [Gol08] for further discussion of randomness in the presence of space restrictions.

Thus, a natural question for further research is to better understand the nature of the trade-off. Even for constant workspace, it does not seem clear how to significantly improve the naive running time of $O(n^4K)$ that can be obtained by computing the whole arrangement and considering each $k \in \{1, \ldots, K\}$ individually. Another open problem concerns the problem of computing a diagram of a given order without computing the diagrams of lower order. Our results also raise the question of how can we compute Voronoi diagrams of order larger than $\sqrt{s}$ when s cells of workspace are available?

Euclidean minimum spanning trees. As the main tool in our algorithm, we have carefully selected a *dense* set of s edges of the graph, called an s-net, for which we have remembered their face incidences. Due to the density property of this s-net, we have quickly found the face of the graph that any given edge lies on. The s-net has been designed to speed up the implementation of Kruskal's EMST algorithm on planar graphs using limited workspace. Nevertheless, this structure is of independent interest as it provides a compact way to represent planar graphs that can be exploited by other algorithms. We hope that this would inspire the further reseach, and that the s-net structure will be applied to wider range of problems.

Althought the optimality of the $O(n^3)$-time algorithm using $O(1)$ cells of workspace by Asano *et al.* [AMRW11] has not been proven, it seems unlikely to have an $o(n^3)$ time algorithm that computes EMST using $O(1)$ cells of workspace. Since our time-space trade-off provides a smooth transition between the $O(n^3)$-time algorithm with constant cells of workspace and the optimal $O(n \log n)$-time algorithm with $O(n)$ cells of workspace, it is plausible that our algorithm is near optimal. However, this is still an open problem.

Visibility regions. We have proposed time-space trade-offs for a class of k-visibility problems in the limited workspace model. In our algorithms, we have used some properties of the k-visibility region in order to restrict the search for all the k-visible edges in an edge list to one specific ranked edge and also to identify that specific ranked edge on each ray using the result for the previous ray.

As we have mentioned in Chapter 5, our approach is also applicable to the slightly different definition of k-visibility region used by Bajuelos *et al.* [BCHPM12] and it provides time-space trade-offs in this setting, too. Moreover, our techniques can be used to improve their result, achieving $O(n \log n)$ running time if $O(n)$ cells of workspace are available.

We leave it as an open question whether the presented algorithms are optimal. Also, it would be interesting to see whether there exists an output sensitive algorithm whose running time depends on the number of windows in the k-visibility region, instead of the critical vertices in the input polygon.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, UK, 2009.

[ABB+13] Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Memory-constrained algorithms for simple polygons. *Comput. Geom. Theory Appl.*, 46(8):959–969, 2013.

[ABOS17] Hee-Kap Ahn, Nicola Baraldo, Eunjin Oh, and Francesco Silvestri. A time-space trade-off for triangulations of points in the plane. In *Proc. 23rd Internat. Comput. and Combinat. Conf. (COCOON)*, pages 3–12, 2017.

[Abr13] Mikkel Abrahamsen. An optimal algorithm computing edge-to-edge visibility in a simple polygon. In *Proc. 25th Canad. Conf. Comput. Geom. (CCCG)*, 2013.

[Abr15] Mikkel Abrahamsen. An optimal algorithm for the separating common tangents of two polygons. In *Proc. 31st Int. Sympos. Comput. Geom. (SoCG)*, pages 198–208, 2015.

[AdBMS98] Pankaj K. Agarwal, Mark de Berg, Jiří Matoušek, and Otfried Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM J. Comput.*, 27(3):654–667, 1998.

[AEK13] Tetsuo Asano, Amr Elmasry, and Jyrki Katajainen. Priority queues and sorting for read-only data. In *Proc. 10th Int. Conf. Theory and Applications of Models of Computation (TAMC)*, pages 32–41, 2013.

[AFMFP+09] Oswin Aichholzer, Ruy Fabila Monroy, David Flores Peñaloza, Thomas Hackl, Clemens Huemer, Jorge Urrutia Galicia, and Birgit Vogtenhuber. Modem illumination of monotone polygons. In *Proc. 25th European Workshop Comput. Geom. (EWCG)*, pages 167–170, 2009.

[AGSS89] Alok Aggarwal, Leonidas J. Guibas, James B. Saxe, and Peter W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete Comput. Geom.*, 4:591–604, 1989.

[AGT86] David Avis, Teren Gum, and Godfried Toussaint. Visibility between two edges of a simple polygon. *The Visual Computer*, 2(6):342–357, 1986.

[AHPV04] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. Approximating extent measures of points. *J. ACM*, 51(4):606–635, 2004.

[AK13] Tetsuo Asano and David G. Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *Proc. 13th Algorithms and Data Structures Symposium (WADS)*, pages 61–72, 2013.

[AKL13] Franz Aurenhammer, Rolf Klein, and Der-Tsai Lee. *Voronoi diagrams and Delaunay triangulations*. World Scientific Publishing, 2013.

[AKP+16] Boris Aronov, Matias Korman, Simon Pratt, André van Renssen, and Marcel Roeloffzen. Time-space trade-offs for triangulating a simple polygon. In *Proc. 15th Scand. Symp. Work. Alg. Theory (SWAT)*, pages 30:1–30:12, 2016.

[AMRW11] Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *J. of Computational Geometry*, 2(1):46–68, 2011.

[AMW11] Tetsuo Asano, Wolfgang Mulzer, and Yajun Wang. Constant-work-space algorithms for shortest paths in trees and simple polygons. *J. Graph. Alg. Appl.*, 15(5):569–586, 2011.

[Asa08] Tetsuo Asano. Constant-working-space algorithms: How fast can we solve problems without using any extra array? In *Proc. 19th Annu. Internat. Sympos. Algorithms Comput. (ISAAC)*, page 1, 2008.

[Aur90] Franz Aurenhammer. A new duality result concerning Voronoi diagrams. *Discrete Comput. Geom.*, 5:243–254, 1990.

[AW16] Mikkel Abrahamsen and Bartosz Walczak. Outer common tangents and nesting of convex hulls in linear time and constant workspace. In *Proc. 24th ESA*, pages 4:1–4:15, 2016.

[BBB+13] Brad Ballinger, Nadia Benbernou, Prosenjit Bose, Mirela Damian, Erik D. Demaine, Vida Dujmovic, Robin Y. Flatland, Ferran Hurtado, John Iacono, Anna Lubiw, Pat Morin, Vera Sacristán Adinolfi, Diane L. Souvaine, and Ryuhei Uehara. Coverage with k-transmitters in the presence of obstacles. *Journal of Combinatorial Optimization*, 25(2):208–233, 2013.

[BBB+18] Yeganeh Bahoo, Bahareh Banyassady, Prosenjit Bose, Stephane Durocher, and Wolfgang Mulzer. A time-space trade-off for computing the k-visibility region of a point in a polygon. *Theoretical Computer Science*, 2018.

[BBM18] Bahareh Banyassady, Luis Barba, and Wolfgang Mulzer. Time-space tradeoffs for computing Euclidean minimum spanning trees. In *Proc. 13th Latin American Symp. Theoretical Inf. (LATIN)*, pages 108–119, 2018.

[BCC04] Hervé Brönnimann, Timothy M. Chan, and Eric Y. Chen. Towards in-place geometric algorithms and data structures. In *Proc. 20th Annu. Sympos. Comput. Geom. (SoCG)*, pages 239–246, 2004.

[BCHPM12] António Leslie Bajuelos, Santiago Canales, Gregorio Hernández-Peñalver, and Ana Mafalda Martins. A hybrid metaheuristic strategy for covering with wireless devices. *J. UCS*, 18(14):1906–1932, 2012.

[BCR+15] Niranka Banerjee, Sankardeep Chakraborty, Venkatesh Raman, Sasanka Roy, and Saket Saurabh. Time-space tradeoffs for dynamic programming algorithms in trees and bounded treewidth graphs. In *Proc. 21st Internat. Comput. and Combinat. Conf. (COCOON)*, pages 349–360, 2015.

[BDK18] Jean-François Baffier, Yago Diez, and Matias Korman. Experimental study of compressed stack algorithms in limited memory environments. In *Proc. 17th Int. Symp. Experimental Algorithms (SEA)*, page to appear, 2018.

[Bea91] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991.

[BCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, third edition, 2008.

[BIK+02] Hervé Brönnimann, John Iacono, Jyrki Katajainen, Pat Morin, Jason Morrison, and Godfried T. Toussaint. In-place planar convex hull algorithms. In *Proc. 5th Latin American Symp. Theoretical Inf. (LATIN)*, pages 494–507, 2002.

[BJ00] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull with optimal query time. In *Proc. 7th Scand. Symp. Work. Alg. Theory (SWAT)*, pages 57–70, 2000.

[BJ02] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proc. 43rd FOCS*, pages 617–626, 2002.

[BKL+15] Luis Barba, Matias Korman, Stefan Langerman, Kunihiko Sadakane, and Rodrigo I. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015.

[BKLS14] Luis Barba, Matias Korman, Stefan Langerman, and Rodrigo I. Silveira. Computing a visibility polygon using few variables. *Comput. Geom. Theory Appl.*, 47(9):918–926, 2014.

[BKM18a] Bahareh Banyassady, Matias Korman, and Wolfgang Mulzer. Computational geometry column 67. *ACM SIGACT News*, 49(2):77–94, 2018.

[BKM+18b] Bahareh Banyassady, Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Improved time-space trade-offs for computing voronoi diagrams. *J. of Computational Geometry*, 7(2):19–45, 2018.

[CC07] Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.

[CC08] Timothy M. Chan and Eric Y. Chen. In-place 2-d nearest neighbor search. In *Proc. 19th SODA*, pages 904–911, 2008.

[CC10] Timothy M. Chan and Eric Y. Chen. Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection. *Comput. Geom. Theory Appl.*, 43(8):636–646, 2010.

[CE87] Bernard Chazelle and Herbert Edelsbrunner. An improved algorithm for constructing kth-order voronoi diagrams. *IEEE Transactions on Computers*, 100(11):1349–1354, 1987.

[Cha00] Timothy M Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM J. Comput.*, 30(2):561–575, 2000.

[Cha06] Timothy M. Chan. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Comput. Geom. Theory Appl.*, 35(1-2):20–35, 2006.

[Cha10] Timothy M. Chan. Comparison-based time-space lower bounds for selection. *ACM Transactions on Algorithms*, 6(2):26, 2010.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, third edition, 2009.

[CMR14] Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Selection and sorting in the “restore” model. In *Proc. 25th SODA*, pages 995–1004, 2014.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry. II. *Discrete Comput. Geom.*, 4(5):387–421, 1989.

[CT16] Timothy M. Chan and Konstantinos Tsakalidis. Optimal deterministic algorithms for 2-d and 3-d shallow cuttings. *Discrete Comput. Geom.*, 56(4):866–881, 2016.

[DE14] Omar Darwish and Amr Elmasry. Optimal time-space tradeoff for the 2d convex-hull problem. In *Proc. 22nd ESA*, pages 284–295, 2014.

[DEG+05] Alice M. Dean, William Evans, Ellen Gethner, Joshua D. Laison, Mohammad Ali Safari, and William T Trotter. Bar k-visibility graphs: Bounds on the number of edges, chromatic number, and thickness. In *Proc. 13th Int. Symp. Graph Drawing (GD)*, pages 73–82, 2005.

[DLS88] James A Dean, Andrzej Lingas, and Jörg-Rüdiger Sack. Recognizing polygons, or how to spy. *The Visual Computer*, 3(6):344–355, 1988.

[EGS07] David Eppstein, Michael T Goodrich, and Nodari Sitchinava. Guard placement for efficient point-in-polygon proofs. In *Proc. 23rd Annu. Sympos. Comput. Geom. (SoCG)*, pages 27–36, 2007.

[Epp00] David Eppstein. Spanning trees and spanners. In Jörg-Rudiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 9, pages 425–461. Elsevier, 2000.

[FHP09] Radoslav Fulek, Andreas F Holmsen, and János Pach. Intersecting convex sets by rays. *Discrete Comput. Geom.*, 42(3):343–358, 2009.

[FM08] Stefan Felsner and Mareike Massow. Parameters of bar k-visibility graphs. *J. Graph. Alg. Appl.*, 12(1):5–27, 2008.

[FMVU09] Ruy Fabila-Monroy, Andres Ruis Vargas, and Jorge Urrutia. On modem illumination problems. In *Proc. 13th Encuentros de Geometría Computacional (EGC)*, 2009.

[Fre87] Greg N Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *Journal of Computer and System Sciences*, 34(1):19–26, 1987.

[Gho07] Subir Kumar Ghosh. *Visibility Algorithms in the Plane.* Cambridge University Press, New York, NY, USA, 2007.

[GJPT78] M. R. Garey, David S. Johnson, Franco P. Preparata, and Robert Endre Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.*, 7(4):175–179, 1978.

[Gol08] Oded Goldreich. *Computational Complexity. A conceptual perspective.* Cambridge University Press, Cambridge, UK, 2008.

[He13] Meng He. Succinct and implicit data structures for computational geometry. In *Space-Efficient Data Structures, Streams, and Algorithms–Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 216–235, 2013.

[HP14] Sariel Har-Peled. Quasi-polynomial time approximation scheme for sparse subsets of polygons. In *Proc. 30th Annu. Sympos. Comput. Geom. (SoCG)*, pages 120–129, 2014.

[HP16] Sariel Har-Peled. Shortest path in a polygon using sublinear space. *J. of Computational Geometry*, 7(2):19–45, 2016.

[HVW07] Stephen G Hartke, Jennifer Vandenbussche, and Paul Wenger. Further results on bar k-visibility graphs. *SIAM J. on Discrete Mathematics*, 21(2):523–531, 2007.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[Ind04] Piotr Indyk. Streaming algorithms for geometric problems. In *Proc. 24th Annu. Conf. Found. Software Technology and Theoret. Comput. Sci. (FSTTCS)*, pages 32–34, 2004.

[Jac88] Guy Joseph Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, 1988.

[Jar73] Ray A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2(1):18–21, 1973.

[JS87] B. Joe and R. B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT*, 27(4):458–473, 1987.

[JT92] Jerzy W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80:1502–1517, 1992.

[KMvR+15] Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and Voronoi diagrams. In *Proc. 14th Algorithms and Data Structures Symposium (WADS)*, pages 482–494, 2015.

[KMvR+17] Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and Voronoi diagrams. *Comput. Geom. Theory Appl.*, page available online, 2017. `doi:10.1016/j.comgeo.2017.01.001`.

[Knu97] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Redwood City, CA, USA, 3rd edition, 1997.

[Kor16] Matias Korman. Memory-constrained algorithms. In *Encyclopedia of Algorithms*, pages 1260–1264. Springer-Verlag, 2nd edition, 2016.

[Lee82] Der-Tsai Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Computers*, 31(6):478–487, 1982.

[Lee83] Der-Tsai Lee. On finding the convex hull of a simple polygon. *International Journal of Parallel Programming*, 12(2):87–98, 1983.

[MM17] Joseph S. B. Mitchell and Wolfgang Mulzer. Proximity algorithms. In Jacob E. Goodman, Joseph O'Rourke, and Csaba D. Tóth, editors, *Handbook of Discrete and Computational Geometry*, pages 849–874. CRC Press, third edition, 2017.

[MP80] J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoret. Comput. Sci.*, 12(3):315–323, 1980.

[MR96] J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165(2):311–323, 1996.

[MS94] Naji Mouawad and Thomas C. Shermer. The superman problem. *The Visual Computer*, 10(8):459–473, 1994.

[Mut05] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016.

[OA17] Eunjin Oh and Hee-Kap Ahn. A New Balanced Subdivision of a Simple Polygon for Time-Space Trade-off Algorithms. In *Proc. 28th Annu. Internat. Sympos. Algorithms Comput. (ISAAC)*, pages 61:1–61:12, 2017.

[O'R12] Joseph O'Rourke. Computational geometry column 52. *ACM SIGACT News*, 43(1):82–85, 2012.

[Poh69] Ira Pohl. A minimum storage algorithm for computing the median. Technical Report RC2701, IBM, 1969.

[Ram99] Edgar A. Ramos. On range reporting, ray shooting and k-level construction. In *Proc. 15th Annu. Sympos. Comput. Geom. (SoCG)*, pages 390–399, 1999.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17, 2008.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. System Sci.*, 4(2):177–192, 1970.

[Sze87] Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–99, 1987.

[Tou80] Godfried T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich alle Hilfsmittel und Hilfen angegeben habe und versichere, auf dieser Grundlage die Arbeit selbständig verfasst zu haben. Die Arbeit habe ich nicht in einem frühren Promotionsverfahren eingereicht.

Berlin, den 19. November 2018

Zusammenfassung

Der Speicherplatzbedarf ist seit den Anfängen des Algorithmenentwurfs von Interesse. Die erhöhte Verfügbarkeit von Geräten mit begrenztem Speicherplatz oder begrenzter Stromversorgung – wie Smartphones, Drohnen oder kleine Sensoren – sowie die Verbreitung neuer Speichermedien, bei denen der Schreibzugriff vergleichsweise langsam ist und negative Auswirkungen auf die Lebensdauer haben kann – wie beispielsweise Flash-Laufwerken – haben zu erneuter Aufmerksamkeit für dieses Thema geführt. In der Folge hat der Entwurf von Algorithmen für das *Limited Workspace Model* (Modell mit begrenztem Arbeitsspeicher) in den letzten zehn Jahren einen signifikanten Anstieg an Popularität in der algorithmischen Geometrie erfahren.

In diesem Setting haben wir in der Regel eine große Menge an Daten, die verarbeitet werden müssen. Obwohl wir auf die Daten beliebig oft und in beliebiger Weise zugreifen können, ist der Schreibzugriff auf den Hauptspeicher begrenzt und/oder langsam. Zwischenergebnisse werden daher nur in einem kleineren, übergeordneten Speicher (z. B. CPU-Register) abgelegt. Da die Anwendungsbereiche der oben genannten Geräte – Sensoren, Smartphones und Drohnen – oft mit einer großen Menge an geografischen (d. h., geometrischen) Daten umgehen, ist dieses Szenario aus Sicht der algorithmischen Geometrie besonders interessant.

Motiviert durch diese Überlegungen haben wir geometrische Probleme im Limited Workspace Model untersucht. In diesem Modell befindet sich die Eingabe der Größe n in einem schreibgeschützten Speicher, ein Algorithmus kann einen Arbeitsspeicher der Größe $s = \{1, \ldots, n\}$ verwenden, um die Zwischendaten während der Ausführung zu lesen und zu schreiben. Die Ausgabe sendet er an einen lesegeschützten Stream. Ziel ist es, Algorithmen zu entwickeln, deren Laufzeit mit zunehmender Verfügbarkeit an Arbeitsspeicher abnimmt, was einen *Time-Space Trade-Off* (Laufzeit-Speicher-Abwägung) darstellt.

In dieser Arbeit betrachten wir drei grundlegende geometrische Probleme, nämlich die Berechnung verschiedener Arten von Voronoi-Diagrammen einer Punktmenge in der Ebene, die Berechnung des euklidischen minimalen Spannbaums einer ebenen Punktmenge und die Bestimmung der k-Sichtbarkeitsregion (k-visibility region) eines Punkts innerhalb eines polygonalen Gebiets. Mit mehreren innovativen Techniken entwickeln wir entweder die ersten Time-Space Trade-Offs für diese Probleme oder verbessern die bisherigen Ergebnisse.