JAIST Repository
https://dspace.jaist.ac.jp/

Title	Memory Constrained Algorithms for Geometric Problems
Author(s)	小長谷, 松雄
Citation	
Issue Date	2016-06
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/13719
Rights	
Description	Supervisor:上原　隆平, 情報科学研究科, 博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Memory Constrained Algorithms for Geometric Problems

by

Matsuo KONAGAYA

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Ryuhei Uehara

School of Information Science
Japan Advanced Institute of Science and Technology

June , 2016

Abstract

Due to recent advancement of technologies of CPU and memory grow in recent years, the possibility of lack of memory space decreases while executing a program. However, the constraint of using limited memory spaces can be required in process on small devices such as digital cameras and cellular phones, because of their volume restriction. In the theoretical sense, considering required space, not only running time, for problems is to be meaningful to catch its complexity.

To design memory constrained algorithms, we have to define our computation model as follows. The memory space in which every stored item is allowed to be read, overwritten is called as the work-space. We use a standard random access machine, so that invoking an item in a memory space takes in constant time. We assume that input data is stored in a read-only array. Thus, no reordering and overwriting to the array are possible. In this paper, we proposed memory constrained algorithms for geometric problems as follows.

First, we consider computing a farthest-point Voronoi diagram using work-space of size $O(\log n)$ bits. Given sets of n points in a plane, a farthest-point Voronoi diagram for the set partition of the plane to regions, such that each region, there exists a farthest point in the set from any point in the region. The situation of using only work-space of size $O(\log n)$ bits is the most strict constraint in our computation model. To invoke every input item stored in a space, sufficient large space is necessary for the work-space to distinguish every index. We call algorithms which is designed under such memory constraint as constant work-space algorithms. It is known as log-space algorithms in computational complexity theory. The algorithm for the Voronoi diagram can have quite simple implementation and also runs in reasonable running time. Moreover, we also consider the problem of finding the smallest enclosing circle for given points, which is applications of farthest-point Voronoi diagram. The problem is one of the fundamental geometric problems as same as Voronoi diagram. We present that our algorithm for finding the smallest enclosing circle can be designed using in constant work-space.

Second, we turn to the Depth-Frist Search (DFS), which is a basic algorithm in many areas, using in $O(n)$ bits work-space. The work-space is lager than constant size, the algorithm could be faster by using the space efficiently. Typically, the sublinear work-space for an input size is supposed when memory constrained algorithms are invented. Instead of using work-space larger than constant size, we devise algorithms to be able to accomplish efficient space usage. So, we investigate fundamental algorithms to obtain somewhat techniques for designing space efficient algorithms. In this paper, we provide algorithms performing DFS on a directed or undirected graph with n vertices and m edges using only $O(n)$ bits. If the work-space of size $O(n \log n)$ bits is available, DFS can be implemented easily by using stack data structure. The advantage of the stack can find the next vertex to be visited in constant time. To find such a vertex without $O(n \log n)$ bits stack, all vertices are maintained by four colors and our DFS algorithms trace given graph in many times as the tracing proceeds.

Third, we also provide an adjustable work-space algorithm for the segment intersection problems. Roughly speaking, the model of an adjustable work-space algorithm can perform with work-space of an arbitrary size between $O(1)$ and $O(n)$ words. Given n line segments in a plane, we invent adjustable work-space algorithms for detecting a segment intersection and for reporting all pairs of intersecting line segments. Specially, algorithm of detecting a segment intersection can be practical, which means that no complicated data structure is needed in the algorithm. We also obtain practical algorithms of reporting segment intersecting pairs for input of c slopes line segments in which the input has at most c different slopes. In general, however,

the number of line slopes can be n. In this case, we have to use a sophisticated data structure.

Finally, we give polynomial-time algorithms for subgraph isomorphism problems for small graph classes of perfect graphs. This work comes from on the way that we try to design memory constrained algorithms for geometric graphs. So far, the memory constrained algorithms for the graphs have not be achieved yet. Although this work is out of the framework of memory constrained algorithms, we include into this paper as a future topics.

Key words: Computational geometry, Algorithms, Memory constrained algorithms, Space-Time tradeoffs

Contents

Chapter 1

Introduction

There are increasing demands for highly functional consumer electronics such as printers, scanners, and digital cameras. To achieve this functionality they need sophisticated embedded software. One fundamental difference from software used in conventional computers is that there is little allowance of work-space which can be used by the software. Programs have been developed under the assumption that sufficient memory space is available. The situation, however, asks us to design algorithms which work in small memory space.

1.1 Memory constrained algorithms

Work-space of an algorithm is a memory space used to store input data and results of temporary calculation for supporting its algorithm. To design memory constrained algorithms we assume that such work-space is given as a auxiliary space different from input and output space. Furthermore, we assume that input space is read-only space, and output space is write-only space, which means that those of space are not available as work-space.

Typically, the amount of work-space for an algorithm is limited into sublinear space for input data of length n, e.g. the work-space of size $O(\sqrt{n})$ words. For example, there is a quite reasonable algorithm to use work-space of $O(\sqrt{n})$ words for an binary image of size $O(\sqrt{n}) \times O(\sqrt{n})$ [12]. Especially, one extreme sublinear constraints is to use only constant number of words. We just compute and maintain a constant number of words for input data consists of n items, and thus it takes work-space of $O(1)$ words of size $O(\log n)$ bits. Algorithms under this constraint have been extensively studied in complexity theory [5]. We call such algorithm as *log-space algorithms* or *constant work-space algorithms* in this paper.

We also consider more flexible usage of work-space for an algorithm than above constraint. Work-space used in an algorithm for a problem involves its performance. Precisely, since there is tradeoffs between time and work-space for the algorithm, thus it runs in slower if less memory space is available. In theoretical sense, studies of time-space tradeoffs of an algorithm for a problem could give better comprehensions for its complexity rather than investigating only either time or space requirement. Many results for time-space tradeoffs have been appeared so far since early 1980s [59]. In particular, Beame and Borodin showed that lower bound for sorting problem of time-space product, which is the complexity of (running time of sorting) $\times$ (required space amount), under the machine model of branching program [17, 3]. Therefore, we also consider another memory constrained algorithms which can perform with a work-space of an arbitrary size. More precisely, work-space of the algorithm for input data of size n is required in $O(s)$ words, where s is parameter with $1 \leq s \leq n$. We call such algorithms as *adjustable work-space algorithms*. In recent years, there are many adjustable work-space algorithms for geometric problems [52, 7, 41]. Chan and Chen provide algorithms for computing convex hull for given n points in a plane [22]. It is known as one of early adjustable work-space algorithms applied for geometric problems. Their algorithm outputs the convex vertices for input points in

the increasing order in $O((n/s)(n + s \log s))$ time with $O(s)$ words.

In this paper, we present memory constrained algorithms for geometric problems. We classify the algorithms with type of the work-space following Section 1.2 and Section 1.2.2, respectively.

1.2 Framework of memory constrained algorithms

Instead of not using a large memory space, constant work-space algorithms could work on every machine. The algorithm can apply to process for massive data set such as so-called streaming algorithms [60]. On the other hand, however, the constraint to the constant work space seems too severe for practical applications. So, it is necessary to assume that much more work-space is available in algorithms rather than constant work-space.

1.2.1 A constant work-space algorithm

In this paper, we present limitted work-space algorithms for two fundamental problems. One of the them is that we present a constant work-space algorithm for drawing a farthest-point Voronoi diagram in Chapter 3. Farthest-point Voronoi diagram is obtained for a point set of n points in a plane. The diagram partitions into unbounded regions surrounded by line segments and rays. Each region has a point p in the given set such that p is the farthest point from any point in its region.

It is usually described using a doubly-connected edge list, which can be computed in $O(n \log n)$ time for n points. It supports the following operations

- To enumerate all Voronoi vertices,
- To enumerate all directed Voronoi edges,
- To determine whether a specified point is on the convex hull, and
- To follow the boundary of the Voronoi region for a point on the convex hull if we specify the point.

Once the doubly-connected edge list is constructed for a given set of points S of n points in $O(n \log n)$ time using $O(n)$ work space, we can enumerate all vertices in $O(1)$ time per vertex. In the constant work-space algorithm, with no preprocessing time we can enumerate all Voronoi vertices in $O(n)$ time per each vertex. It is just the same for Voronoi edges. Following the boundary of a Voronoi region is also done in $O(n)$ time per step.

1.2.2 $O(n)$ bits work-space algorithms

The other constraint is somewhat relaxed from constant work-space algorithms. In Chapter 4, we investigate how one can perform Depth-First Search (DFS) of a given graph using limited amount of memory. DFS, being one of the most fundamental and important ways to search or explore a graph, is used as a subroutine in many prominent graph algorithms [70]. A better understanding of DFS in terms of space complexity and memory-efficient algorithms is desirable, but it appears that such aspects of DFS have not been considered much. This is perhaps because the problem is trivial if one uses $O(n \log n)$ bits of memory on one hand, where n is the number of vertices of a graph, and, on the other hand, the problem is P-complete [65] and thus polylogarithmic-space algorithms are unlikely to exist.

1.3 Adjustable work-space algorithms

We propose adjustable work-space algorithms for segment intersection problems. The work-space is sublinear, however, work-space of the size $O(s)$ words is available, where s is parameter with $1 \leq s \leq n$.

The problem has been well studied. Given n segments in the plane, we can report all K intersections in $O(n \log n + K)$ time if we can use work-space of size $O(n)$ words. Since $\Omega(n \log n + K)$ time is required in the worst case, the algorithm given by Balaban [15] achieving $O(n \log n + K)$ time and $O(n)$ words is optimal.

In Section 5.1 we begin with a simple adjustable work-space algorithm for segment intersection detection, which runs in $O((n^2/s) \log s)$ time using work-space of $O(s)$ words for a set of n line segments stored in a read-only array. Section 5.2 extends the result to the problem of reporting intersections using $O(s)$ work space. We present three different adjustable work-space algorithms all of which run in $O((n^2/s) \log s + K)$ time for a set of n *isothetic* segments (e.g. when given segments are horizontal or vertical). We need some special treatment if input segments may overlap each other, that is, if their intersection (in the mathematical sense) is a line segment, not a line. We show this problem can be resolved using techniques called filtering search given by Chazelle [25] . We also present an adjustable work space algorithm for a general set of segments with arbitrary slopes. The algorithm runs in roughly $O(n^2/\sqrt{s} + K)$ time.

We aim to design adjustable work-space algorithms so that it can be simple implementation into a program. Thus, algorithms should not be adopted complicated data structures such as multi-layer data structure [29]. Although our algorithm for segment intersection detection applies for plane sweep algorithms provided by Bentley and Ottman [19], it is still quite simple. We also aim to obtain deterministic algorithms. The product of time and space required in an algorithm for a problem should have an optimal lower bound. Considering memory constrained algorithms like adjustable work-space algorithm is helpful to understand for the study.

1.4 Paper organization

We present memory constrained algorithms for geometric problems in this paper. In Chapter 2, we define computation model on which our algorithms execute.

In Chapter 3, we propose memory constrained algorithms for fundamental geometric problem, but it is not adjustable work-space. The problem is to find a farthest-point Voronoi diagram for given n points in a plane. In the chapter, we present constant work-space (i.e. $O(\log n)$ bits) algorithm for the problem. The algorithm consists of some functions which also perform using only constant work-space. Those functions are described in Section 3.2 with their implementations. Especially, we can obtain an simple algorithm for an application of farthest-point Voronoi diagram, which is the problem of finding smallest enclosing circle. So, we give algorithms for drawing farthest-point Voronoi diagram and its application in Section 3.3.

We present another memory constrained algorithm in Chapter 4 for Depth-First Search, which traces all vertices and edges in depth-first order for a given graph with n vertices and m edges. We show that Depth-First search can implement using the work-space of size $O(n)$ bits in Section 4.5. Furthermore, in Section 4.6 we provide also a constant work-space algorithm which performs Depth-First Search for spacial graphs of tree and $O(1)$-size feedback vertex set.

In Chapter 5, we present adjustable work-space algorithm for segment intersection problems. Precisely, given n line segments in a plane, the objectives of the problem are to find

a segment intersection or to report all pairs of intersecting line segments. In Section 5.1 we present an adjustable work-space algorithm for the former problem which is called as *Segment Intersection Detections*, in Section 5.2 algorithms are given for the latter problem of *Segment Intersection Reporting*, respectively. Both of the algorithms can work on the work-space of size $O(s)$ words, where s is parameter with $1 \leq s \leq n$, which means an arbitrary size is available as its work-space.

In Chapter 6, as an additional work of this paper, we also give the results of the polynomial-time algorithms for Subgraph Isomorphism problems. And we prove the NP-hardness of the Subgraph Isomorphism problem in small graph classes for perfect graph. It was supposed to consider designing memory constrained algorithms associated with some geometric graphs classes. It remains one of the future work.

Chapter 2

Preliminaries

2.1 Computation model

To design memory constrained algorithms, we define the computation model used in our algorithms (Fig 2.1). First of all, we assume that our algorithms run on the standard RAM model.

The *work-space* is storages in which every stored element is allowed to be read, overwritten and removed. We measure the amount of work-space with the number of *words* associated with the size of input data. Work-space of a word is assumed to be large enough to store an input data or pointer. In this paper, we use the term of words for represent the amount of work-space. Assuming that one word equals to $O(\log n)$ bits, we also use the term of *bits* to denote the size. We also assume that the work-space is given different form input and output space.

In addition, we assume that input data of size n is stored in a read-only array. So, no reordering elements and overwriting in the array are possible while the algorithm is running. Output data is not stored in a memory space. Each element of output data is immediately reported when it is just generated. The facts imply the space of input and output storing are not available as the work-space. As a motivation of using read-only property, given input data, we may want to perform several different algorithms. If we reorder input points for an algorithm, then we have to reorder them for another one. For example, a good ordering for closet-point Voronoi diagram may be different from one for farthest-point Voronoi diagram. In fact, a problem of finding the minimum-width annulus for a set of points in the plane can be solved using both of the Voronoi diagrams. The topic is described in Section 3.3.

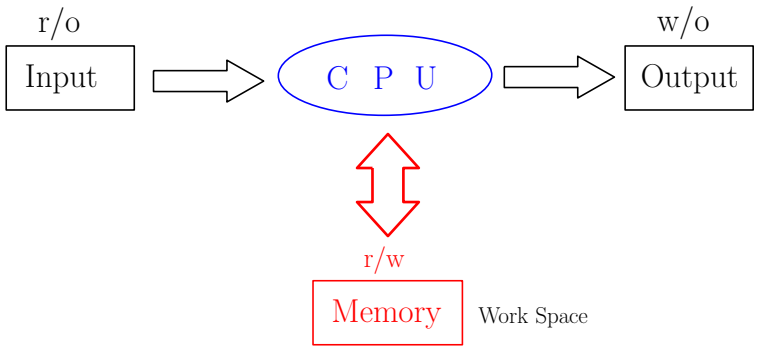

Figure 2.1: Diagram of computation model

Chapter 3

Constant work-space algorithm for farthest-point Voronoi Diagram

In this chapter, we presents a constant work-space algorithm for drawing a farthest-point Voronoi diagram. Voronoi diagram for a set of n points in the plane, which is a collection of algorithms for supporting various operations on the diagram using only a constant number of words of $O(\log n)$ bits in addition to a read-only array to store the given point set. We show the operations to be supported can be executed in $O(n)$ time without using only constant work-space (without using any extra array). This is an extension of our previous results [6, 11, 14, 13].

3.1 Notations and functions for farthest-point Voronoi diagram

We define some notations for farthest-point Voronoi diagram and functions for supporting the constant work-space algorithm. We consider algorithm for a farthest-point Voronoi diagram $FV(S)$ for a set S of n points in the plane. For simplicity we assume that given points are in general positions, that is no four points of S are cocircular and thus every vertex of $FV(S)$ is incident to exactly three Voronoi edges. This restriction will be removed later. A diagram is defined by Voronoi regions and Voronoi edges. A Voronoi region $FVR(p_i)$ for a point $p_i \in S$ is the region such that the point p_i is farthest among the point set S from any point in the region. Each Voronoi region is known to be an infinite polygonal region, whose boundary consists of two infinite edges and (possibly no) finite edges with two endpoints. In this paper we orient Voronoi edges on the boundary of a Voronoi region $FVR(p_i)$ so that the Voronoi region for the point p_i lies to their left. Each Voronoi edge lies between two Voronoi regions. So, by $E(p_i, p_j)$ we denote a Voronoi edge between two Voronoi regions $FVR(p_i)$ and $FVR(p_j)$ with $FVR(p_i)$ to its left (and $FVR(p_j)$ to its right). Thus, the oppositely directed edge, called the twin edge, is denoted by $E(p_j, p_i)$. By our assumption exactly three Voronoi edges meet at each endpoint of Voronoi edges, which defines a Voronoi vertex. Thus, we can assume that each Voronoi vertex is characterized by three points from an input points such as $V(p_i, p_j, p_k)$.

It is well known that only those points of an input point set on its convex hull have their Voronoi regions [29, 64].

An example of a farthest-point Voronoi diagram is shown in Figure 3.1. In the figure, the leftmost point among the input points is denoted by p_1 and other input points on the convex hull are denoted by $p_2, \ldots, p_5$ in the counter-clockwise order. The Voronoi region $FVR(p_1)$ for the point p_1 is shadowed in the figure.

A farthest-point Voronoi diagram is defined by Voronoi vertices, Voronoi edges which are either directed rays or directed line segments, and Voronoi regions which are infinite regions. It is common to use a doubly-connected edge list (DCEL in short) to represent a farthest-point

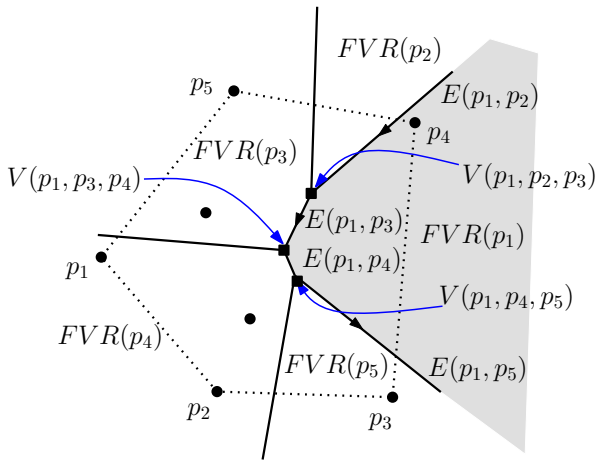

Figure 3.1: Farthest-point Voronoi diagram. Vertices on the convex hull are $\{p_1,\ldots,p_5\}$. $FVR(p_i)$ and $E(p_i,p_j)$ are the Voronoi region for point p_i and Voronoi edge for two points p_i and p_j, respectively.

Voronoi diagram. The DCEL consists of three collections of records [29].

Vertex record: A vertex record of a vertex v stores the coordinates of v and a pointer *IncidentEdge(v)* to a directed edge outgoing of v.

Face record: A face record of a face f stores a pointer *FirstVoronoiEdge(f)* to the first Voronoi edge on the boundary of the face f, which is a ray from the infinity.

Edge record: An edge record of a Voronoi edge e stores a pointer *NextVoronoiEdge(e)* to the next Voronoi edge on the same boundary and a pointer *IncidentFace(e)* to the face to its left.

We support these functions, *IncidentEdge(v)*, *FirstVoronoiEdge(f)*, *NextVoronoiEdge(e)*, and *IncidentFace(e)* by providing the following functions.

FirstExtremePoint(S) returns the leftmost extreme point (more exactly, the index of the point) in a set S of points.

CounterClockwiseNextExtremePoint(p_i) returns the index of the extreme point next to p_i in a counter-clockwise order on the convex hull.

FrontEndpointOfVoronoiEdge($E(p_i,p_j)$) returns the index k of the point p_k of S that determines the front (terminating) endpoint $V(p_i,p_j,p_k)$ of a directed Voronoi edge $E(p_i,p_j)$.

BackEndpointOfVoronoiEdge($E(p_i,p_j)$) returns the index k of the point p_k of S that determines the back (starting) endpoint $V(p_i,p_j,p_k)$ of a directed Voronoi edge $E(p_i,p_j)$.

NextVoronoiEdge($E(p_i,p_j),V(p_i,p_j,p_k)$) returns the next Voronoi edge $E(p_i,p_k)$ of $E(p_i,p_j)$ on the Voronoi region $FVR(p_i)$ which starts at the Voronoi vertex $V(p_i,p_j,p_k)$, more exactly the two indices i and k.

ExtremePoint(p_i) returns TRUE if and only if the point p_i is on the convex hull.

3.2 Functions for supporting constant work-space algorithm

We propose implementations of above functions for supporting our algorithm. Our constant work-space algorithm first computes the centroid c of the input point set in advance by computing the average x and y coordinates of all given points. It is well known that the centroid always lies in the interior of the convex hull for the point set.

The operations listed above can be implemented in linear time using only $O(\log n)$ bits as follows:

FirstExtremePoint(S): The leftmost point in a point set S must be on the convex hull of S since the left half plane defined by the vertical line through the leftmost point is empty (i.e., no point of S is contained there). It is easy to find the leftmost point in S in $O(n)$ time using $O(\log n)$ bits.

CounterClockwiseNextExtremePoint(p_i): Let $p_i \in S$ be an extreme point on the convex hull of S. We define a ray emanating from the point p_i in the opposite direction to the centroid c (refer to Figure 3.2). Then, we rotate the ray in the counter-clockwise order until it encounters a point of S, which is the point required. This is an intuitive description of an algorithm. Formally, we find the next extreme point p_j as follows. The point p_j must satisfy the following two properties:
(1) (c, p_i, p_j) is clockwisely oriented since p_j must be to the left of the directed line $\overrightarrow{cp_i}$, and
(2) for any other point $p_k \in S \setminus \{p_i, p_j\}$ with the property (1) the three points p_k, p_i, p_j are ordered clockwisely since p_j lies to the left of $\overrightarrow{p_k p_i}$ to minimize the angle with the ray from p_i (see Figure 3.2). Thus, it can be computed in $O(n)$ time using $O(\log n)$ bits.

FrontEndpointOfVoronoiEdge($E(p_i, p_j)$): Each Voronoi edge $E(p_i, p_j)$ is associated with one or two enclosing circles, whose centers are the endpoints of the edge. Due to our orientation, the front endpoint of a Voronoi edge $E(p_i, p_j)$ is determined by a point of S lying to the left of $\overrightarrow{p_i p_j}$. For each such point p_k (such that (p_i, p_j, p_k) is counter-clockwisely ordered) we compute the center of the circle through p_i, p_j, and p_k. This is a kind of mapping of a point of S into one on the perpendicular bisector of p_i and p_j. The center point giving the front endpoint must give an enclosing circle as stated above. Thus, the center point must be farthest from the center point of p_i and p_j. Thus, it can be computed in $O(n)$ time using $O(\log n)$ bits. See Figure 3.3. It shows how to find such a point. Given a Voronoi edge $E(p_1, p_3)$, extreme points of S lying to the left of the directed line $\overrightarrow{p_1 p_3}$ are p_4 and p_5. Since p_4 corresponds to a larger circle, the front endpoint of the edge is determined by p_4 together with p_1 and p_3 in this example. It should be noted that the last Voronoi edge on the boundary of a Voronoi region when we traverse it counterclockwisely has its front endpoint at infinity and thus its front endpoint is undefined.

BackEndpointOfVoronoiEdge($E(p_i, p_j)$): This is just symmetric to the case of the front endpoint. Thus, the first Voronoi edge on the boundary of a Voronoi region has no back endpoint.

NextVoronoiEdge($E(p_i, p_j), V(p_i, p_j, p_k)$): Once Voronoi edge $E(p_i, p_j)$ and its front endpoint $V(p_i, p_j, p_k)$ are known (more exactly, three indices i, j, and k are known), the next Voronoi edge is $E(p_i, p_k)$. Thus, it is done in $O(1)$ time.

ExtremePoint(p_i) We can easily compute the line L_i that is perpendicular to the ray from the centroid c toward p_i. If one of the half plane contains no point of S except p_i on the boundary, then the point p_i is on the convex hull by the definition of the convex hull. Otherwise, it is an interior point. See Figure 3.4. This is done in $O(n)$ time.

In addition, given a Voronoi edge $E(p_i, p_j)$ and its front endpoint $V(p_i, p_j, p_k)$, we know the three Voronoi edges $E(p_j, p_i), E(p_i, p_k)$ and $E(p_k, p_j)$ are outgoing edges from the Voronoi vertex $V(p_i, p_j, p_k)$ ordered in a clockwise way around the vertex. Thus, the algorithm above behaves like a doubly-connected edge list.

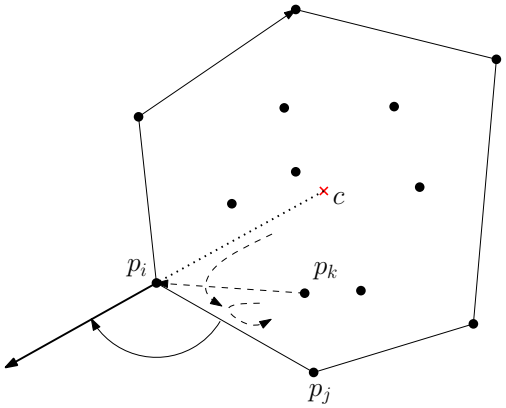

Figure 3.2: Finding the counter-clockwise next extreme point using a ray from p_i and the centroid c.

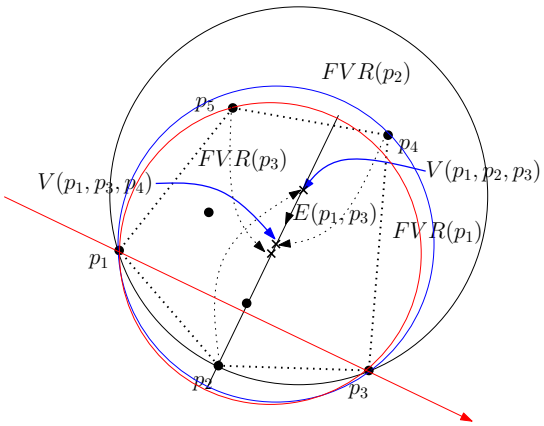

Figure 3.3: Finding the front endpoint of a Voronoi edge which is determined by a point of S lying to the left of the directed line $\overline{p_1p_3}$. Points lying to the directed line $\overline{p_1p_3}$ are p_4 and p_5. The center point of the circle defined by (p_1, p_3, p_4) is farther than that defined by (p_1, p_3, p_5), and thus the front endpoint of the directed Voronoi edge $E(p_1, p_3)$ is the Voronoi vertex $V(p_1, p_3, p_4)$. On the other hand, only one point p_2 lies to the directed line $\overline{p_3p_1}$, and thus that of $E(p_3, p_1)$ is $V(p_3, p_1, p_2) = V(p_1, p_2, p_3)$.

3.3 Applications of the algorithm

Using the constant work-space algorithm for farthest-Point Voronoi diagram, we can of course draw the diagram for any given set of n points in $O(n^2)$ time using only $O(\log n)$ bits given as

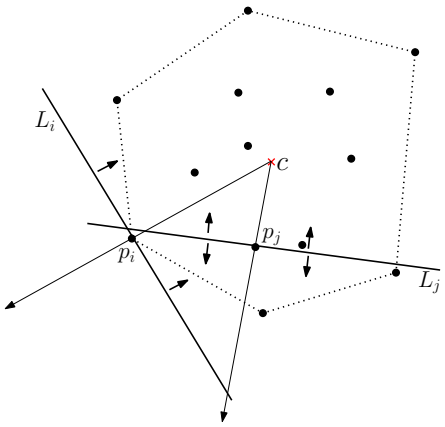

Figure 3.4: Deciding whether a given point is on the convex hull. The point p_i is on the convex hull shown by dotted lines since one of the half plane defined by the line L_i is empty. The point p_j is not so since none of the half planes is empty.

Algorithm 1 below.

A constant-work-space algorithm for drawing the farthest-point Voronoi diagram
Input: A set $S = \{p_1, \ldots, p_n\}$ of n points.
Output: Voronoi edges and Voronoi vertices of the farthest-point Voronoi diagram of the set S.
Algorithm{
  p_i = FirstExtremePoint(S).
  $i_0 = i$.
  repeat{
    p_j = CounterClockwiseNextExtremePoint(p_i).
    p_k = FrontEndpointOfVoronoiEdge(p_i, p_j).
    Report the first Voronoi edge $E(p_i, p_j)$ emanating from the Voronoi vertex $V(p_i, p_j, p_k)$.
    repeat{
      $p_j = p_k$.
      p_k = FrontEndpointOfVoronoiEdge(p_i, p_j).
      if(p_k is undefined) **then** exit the loop.
      Report the Voronoi edge (segment) $E(p_i, p_j)$ (pair of indices i and j in practice) and the Voronoi vertex $V(p_i, p_j, p_k)$ together with its coordinates and three indices.
    }(forever)
  } **until**($i = i_0$)
  Report the last Voronoi edge (ray) $E(p_i, p_j)$ emanating from the last Voronoi vertex.
  p_i = CounterClockwiseNextExtremePoint(p_i).
}

We can also compute the smallest enclosing circle of the points set by enumerating all the Voronoi vertices and Voronoi edges in $O(n^2)$ time. The smallest enclosing circle for a

point set S is defined either by three points associated with a Voronoi vertex or by a diametral pair of extreme points. In the former case the point must appear as a Voronoi diagram of the farthest-point Voronoi diagram. In the latter case, the diametral pair of points mus appear as one associated with a Voronoi edge. Thus, if we enumerate all Voronoi vertices and Voronoi edges, we can find the center of the smallest enclosing circle. Since there are $O(n)$ Voronoi vertices and edges, the algorithm runs in $O(n^2)$ time.

Another application is to the smallest annulus of a point set. Given a set S of n points in the plane, two co-centric circles are called an annulus of S if all the points of S lie between the two circles. See Figure 3.5. The width of an annulus is the difference of the two radii.

There are two cases to determine the center of the smallest-width annulus. In one case one of the circles is determined by three points and the other by a single point. In the other case both of them are determined by two points. The center in the latter case is given as an intersection of two Voronoi edges, one from the closest-point Voronoi diagram and the other from the farthest-point Voronoi diagram of S [33]. An algorithm for enumerating all the edges of the closest-point Voronoi diagram in $O(n^2)$ time using $O(\log n)$ bits is available [13]. Thus, a straightforward algorithm is to enumerate all edges of the farthest-point Voronoi diagram for each edge in the closest-point Voronoi diagram and to check intersection of those edges from different Voronoi diagrams. This algorithm runs in $O(n^4)$ time and $O(\log n)$ bits.

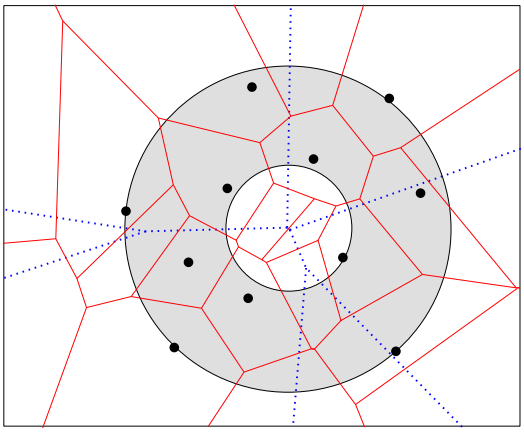

Figure 3.5: The minimum-width annulus for a set of points. The closest-point and farthest point Voronoi diagrams are drawn in solid and dotted lines, respectively, in the figure.

3.4 How to cope with degeneracies

We have assumed that given points are in general positions, that is, (1) no three points are on a line or (2) no four points are on a circle. In this section we will show how to cope with degeneracies on given points.

3.4.1 Degeneracy caused by collinear points

Figure 3.6 shows an example of a degeneracy caused by collinear points in which four points lie on the convex hull of a given point set.

Suppose three points from an input point set S lie on a line and one of the half plane defined by the line is empty, that is, it contains no point of S. If three points p_a, p_b, and p_c are arranged in this order on the line, the middle point p_b never contributes to the farthest-point Voronoi

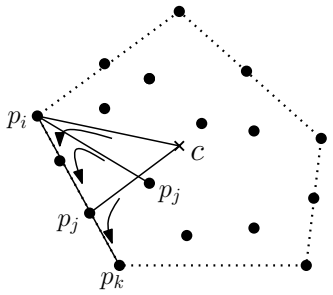

Figure 3.6: Degeneracy caused by collinear points.

diagram for S, in other words, p_b has no its own Voronoi region, for any circle touching p_b never includes both of p_a and p_c, and the point p_a (resp., p_c) lying outside the circle is farther from the center of the circle than the other point p_c (resp., p_a). This means that we can neglect those intermediate points on the convex hull edges, which are not convex hull vertices. All these observations lead to the following algorithm for *CounterClockwiseNextExtremePoint(p_i)*:

CounterClockwiseNextExtremePoint(p_i)
for each point $p_k \in S \setminus \{p_i\}$ **do**
if (c, p_i, p_k) is counter-clockwise **then**
break;
for each point p_j, $j = k+1, \ldots, n$ **do**
if (c, p_i, p_j) is counter-clockwise **then**
if (p_k, p_i, p_j) is counter-clockwise
then $p_k = p_j$;
else if (p_k, p_i, p_j) is collinear **and** (c, p_k, p_j)
is counter-clockwise **then** $p_k = p_j$;
return p_k.

3.4.2 Degeneracy caused by cocircular points

Figure 3.7 shows another type of degeneracy, which is caused by cocircular points. In the figure five points $p_1, \ldots, p_5$ on the convex hull lie on a circle.

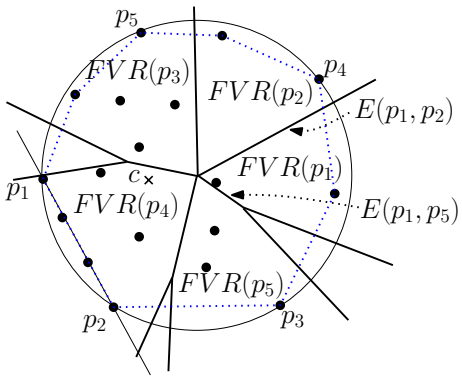

Figure 3.7: Degeneracy caused by cocircular points.

Suppose we are about to examine a convex hull edge (p_1, p_2). We first find a Voronoi edge $E(p_1, p_2)$, which is a ray from the infinity, as shown in the figure. To compute its front endpoint

we examine all the points lying to the left of the directed line from p_1 to p_2 to find one whose corresponding circle center is farthest from the middle point of p_1 and p_2. In this case the three points p_3, p_4 and p_5 all give the same circle center since they are cocircular. Note that all those points must be extreme points. What we want is the point closest to p_2 in the clockwise order on the convex hull. Thus, if we find two candidate extreme points p_a and p_b to define the front endpoint of a Voronoi edge $E(p_i, p_j)$ and the four points p_a, p_b, p_i and p_j are cocircular, then we check the orientation of (p_i, p_a, p_b). We choose p_a if it is counter-clockwise, and choose p_b otherwise. This extra condition leads to a correct ordering of those cocircular points. In the example of Figure 3.7, the front endpoint of $E(p_1, p_2)$ is defined by p_3, and thus the next Voronoi edge should be $E(p_1, p_3)$. Its front endpoint is defined by p_4 and thus the following edge should be $E(p_1, p_4)$. In the same manner the Voronoi edge $E(p_1, p_4)$ is followed by $E(p_1, p_5)$. So, we have an edge sequence $E(p_1, p_2), E(p_1, p_3), E(p_1, p_4), E(p_1, p_5)$. Here note that the Voronoi edges $E(p_1, p_3)$ and $E(p_1, p_4)$ are degenerated edges, that is, their two endpoints coincide.

3.5 Concluding Remarks in Chapter 3

We have presented a constant work-space algorithm for a farthest-point Voronoi diagram, which is a collection of algorithms to execute all of operations associated with the diagram as efficiently as possible using only constant work-space. A number of problems are left open. One of them is to establish some trade-off between running time and amount of work-space. Given work-space of $O(s)$ words, how fast can we compute a farthest-point Voronoi diagram? It is not known whether we can establish time complexity such as $O(n^2/s)$ or $O(n^2/s \log n)$. To answer this question we need to devise a algorithm using $O(s)$ space with $s \in o(n)$. One typical question is how fast can we draw a farthest-point Voronoi diagram for a set of n point in the plane using the work-space of size $O(\sqrt{n})$ words.

Chapter 4

Depth-First Search Using $O(n)$ Bits

We provide algorithms performing Depth-First Search (DFS) on a directed or undirected graph with n vertices and m edges using only $O(n)$ bits. One algorithm uses $O(n)$ bits and runs in $O(m \log n)$ time. Another algorithm uses $n+o(n)$ bits and runs in polynomial time. Furthermore, we show that DFS on a directed acyclic graph can be done in work-space $n/2^{\Omega(\sqrt{\log n})}$ bits and in polynomial time, and we also give a simple linear-time $O(\log n)$ bits algorithm for the depth-first traversal of an undirected tree. Finally, we also show that for a graph having an $O(1)$-size feedback set, DFS can be done in $O(\log n)$ bits work-space. Our algorithms are based on the analysis of properties of DFS and applications of the *s-t* connectivity algorithms due to Reingold and Barnes et al., both of which run in sublinear space.

4.1 The DFS problem

Before we outline previous work on DFS, we explain some technical details about the DFS problem. We can cast a DFS problem in various ways. The output can be (1) the DFS tree, or the output can be (2) the DFS numbering of each vertex, that is, the ordering of vertices with respect to the time of the first visit, or (3) the input can be a graph together with two vertices u and v, and the output can be the yes/no answer as to whether vertex u is visited before vertex v in DFS. For our purposes, which of the three variants above we consider does not matter since they can all be reduced to each other using $O(\log n)$ bits. Furthermore, all the algorithms we present can directly handle any of the three variants in a straightforward way. For definiteness, we think of DFS problem as the DFS tree construction problem.

We assume that an input graph is given by an adjacency list. Suppose that DFS is visiting a vertex v for the first time, reaching v from vertex u. DFS will now visit the first unvisited neighbor of v, where the "first" is usually with respect to either one of the following two orders: (1) the appearance order in v's adjacency list; or, (2) in the case of undirected graphs: under the assumption that n vertices are numbered $1, \ldots, n$ and with respect to the cyclic ordering of $1, \ldots, n$, the unvisited vertex x among v's neighbors that appears first after u in the cyclic ordering.

DFS with respect to either one of the two scenarios above is sometimes called *lexicographically smallest DFS* or *lexicographic* DFS or *lex-DFS* [30], [31] (and sometimes simply called DFS). Usually, a lex-DFS algorithm can handle both scenarios (1) and (2) in the same manner, and one does not need to distinguish the two scenarios. All algorithms in this paper perform lex-DFS.

In contrast to lex-DFS, we can also consider an algorithm that outputs *some* DFS tree of a given graph. Such an algorithm treats an adjacency list as a *set*, ignoring the order of appearance of vertices in it, and outputs a spanning tree T such that there exists *some* adjacency ordering R such that T is the DFS tree with respect to R. We say that such a DFS algorithm performs *general-DFS*.

4.2 Related work

This paper is concerned with the more classical Random Access Machine (RAM) model, where input data is in read-only random access memory, and computation proceeds using additional working space, which, for example, consists of $O(\log n)$ or $o(n)$ or $O(n)$ bits. The output will be stored in a write-only output tape. For this model, recent works have given some new interesting memory-limited algorithms: Elberfeld et al. [34] and Elberfeld and Kawarabayashi [35] have given $O(\log n)$ bits algorithms for solving a family of fundamental graph problems (more precisely those problems expressibly in monadic second-order language on graphs of bounded tree-width) and for the canonization of graphs of bounded genus. Very recently, Asano et al. [10] and Imai et al. [47] have shown that the reachability problem on directed graphs can be solved using only $\tilde{O}(\sqrt{n})$ words for planar graphs.

Reif [65] has shown that lex-DFS is P-complete. Anderson and Mayr [4] have shown that computing the lexicographically first maximal path, that is, computing the leftmost root-to-leaf path of the lex-DFS tree, is already P-complete.

Aggarwal and Anderson [2] have shown that general-DFS is computable in RNC, that is, computable by a randomized parallel algorithm with polynomially many processors and in poly-logarithmic parallel time in the PRAM model, or, equivalently, by randomized polynomial-size poly-logarithmic depth circuits. There is no known deterministic NC algorithm for general-DFS.

In a seminal work, Reingold [66] has given a deterministic $O(\log n)$ bits work-space algorithm for the Undirected *s-t* Connectivity Problem:

Theorem 1 (Reingold [66]) *Given an undirected graph and two vertices s and t, determining whether s and t are connected can be done in deterministic $O(\log n)$ bits.*

Using Reingold's algorithm, one can compute a minimum spanning tree of a given graph in $O(\log n)$ bits.

The *s-t* connectivity problem for *directed* graphs is NL-complete. This problem can be solved using $O(\log^2 n)$ bits and $n^{O(\log n)}$ time by Savitch's algorithm (see [62]). Concerning polynomial-time algorithms solving this problem, the best known upper bound for space is the following slightly sublinear one due to Barnes et al. [16]:

Theorem 2 (Barnes et al. [16]) *Directed s-t connectivity can be solved deterministically in work-space $n/2^{\Omega(\sqrt{\log n})}$ bits and in polynomial time.*

This is also the best space upper bound for polynomial-time algorithms solving the following problems: computing the distance between a vertex s and a vertex t in an undirected or directed unweighted graph, computing the single-source shortest-path tree in a weighted undirected or directed graph [46], and a computing the breadth-first search tree.

4.3 Preliminaries

Throughout the paper, we assume that the set of vertices of a given graph is the set $\{1, \ldots, n\}$.

We think of DFS in the following way: Initially, all the vertices are *white*. When vertex v is visited from vertex u, the color of v changes from white to *gray* and the *search head* moves *from u to v*. When there is no more white neighbor of v, the search at v is finished, the color of v changes from gray to *black*, and the search head *returns from v to u*.

Suppose that in a given undirected or directed graph, m vertices are reachable from the DFS starting vertex s. At time $t = 0$, all vertices are white. At time $t = 1$, the starting vertex s becomes gray. At each time $t \geq 1$, exactly one vertex changes its color, either from white to gray, or, from gray to black. At time $t = 2m$, the color of s becomes black and the search is completed. For a vertex v, the *discovery time* of v is the time when v changes its color from white to gray and the *finishing time* is the time when v changes its color from gray to black.

Note that the gray vertices always form a simple path from the starting vertex s to the vertex where the search head is currently located. We can also think of this path as residing in the *depth-first-search tree*.

We let Reachable(x, u, G) denote a subroutine that decides, given a graph G and two vertices x and u, whether vertex u is reachable from vertex x in G. If G is a directed graph, reachability is interpreted in terms of a directed path, and if G is undirected, it simply means connectivity. To implement Reachable(x, u, G) we apply Reingold's algorithm and Barnes et al.'s algorithm for the cases of undirected and directed graphs, respectively.

4.4 Characterizations for the Gray and Black Vertices

In this section, for the sake of convenience, we collect our lemmas characterizing the gray vertices, the gray path, and the black vertices in several settings. These lemmas naturally yield our algorithms in the next section and they are crucial to explain their correctness. For most lemmas, proofs are immediate and omitted.

Lemma 3 (All-White Path) *Vertex v is visited during DFS while vertex u is gray, (i.e., v is a descendant of u in the DFS tree) if and only if the following holds: At the time u is discovered, v is white and v can be reached from u by an all-white path.*

Let s be the starting vertex of a DFS. In the following we assume that the state of the DFS at time t is such that the search head is at a gray vertex u. Let $p = \langle i_0 = s, i_1, \ldots, i_{k-1}, i_k = u \rangle$ be the gray path at time t, where i_{j+1} is visited from i_j (for $0 \leq j < k$).

The following lemma characterizes the gray path in terms of black vertices.

Lemma 4 (Gray Path from Black Vertices) *The gray path $p = \langle i_0 = s, i_1, \ldots, i_{k-1}, i_k = u \rangle$ satisfies the following. For $j \in \{0, \ldots, k-1\}$, vertex i_{j+1} is the first vertex x in the adjacency list of vertex i_j such that (1) x is not black at time t, and that (2) x is not in $\{i_0, \ldots, i_j\}$.*

Proof. Vertex i_{j+1} becomes gray only after all the vertices in the adjacency list of i_j preceding i_{j+1} have become non-white. □

Let $C = \{i_0, \ldots, i_k\}$ be the set of gray vertices comprising the gray path P. The following characterization explains how to reconstruct the *path* P from the *set* C.

Lemma 5 (Gray Path from Gray Set) *Let $P' = \langle i_0, \ldots, i_j \rangle$ be the initial segment of P of length j. Then, the following characterizes the immediate successor $x = i_{j+1}$ of i_j in P.*
(1) x is in C.
(2) x is a neighbor of i_j.
(3) x is not in $\{i_0, \ldots, i_j\}$.
(4) x is the first vertex in the adjacency list of i_j satisfying (1), (2), and (3).

For our algorithms we need to be able to reconstruct the gray path $p = \langle i_0 = s, i_1, \ldots, i_{k-1}, i_k = u \rangle$ from the two endpoints s and u alone. The following lemma characterizes the vertices $i_1, \ldots, i_{k-1}$ in such a way that one can reconstruct them given s and u. The proof immediately follows from Lemma 3 (All-White Path).

Lemma 6 (Gray Path) *For $j \in \{1, \ldots, k-1\}$, vertex i_j is the first vertex x in the adjacency list of i_{j-1} from which vertex u can be reached without going through any of the vertices in $\{i_0, \ldots, i_{j-1}\}$.*

Corollary 7 (Gray Path Reconstruction) *Using an n-bit vector one can reconstruct $i_1, \ldots, i_{k-1}$ one by one as follows. For $j = 1, \ldots, k-1$, for each vertex x adjacent to i_{j-1}, use Reachable$(x, u, G - \{i_0, \ldots, i_{j-1}\})$ and the previous lemma to determine whether x is i_j.*

When considering DFS on a DAG the characterization and reconstruction of the gray path simplifies as follows.

Lemma 8 (Gray Path in a DAG) *For $j \in \{1, \ldots, k-1\}$, vertex i_j is the first vertex x in the adjacency list of i_{j-1} such that vertex u is reachable from x.*

Proof. Let P be a directed simple path from x to u. Then no vertex y in $\{i_0, \ldots, i_{j-1}\}$ can appear on the path P since x is reachable from y and the graph is acyclic. □

Corollary 9 (Gray Path Reconstruction in a DAG) *For a DAG, one can reconstruct $i_1, \ldots, i_{k-1}$ similarly as in the Gray Path Reconstruction Corollary above but without keeping track of $i_0, \ldots, i_{j-1}$ by replacing the call to the routine Reachable$(x, u, G - \{i_0, \ldots, i_{j-1}\})$ with Reachable(x, u, G).*

Let s be a starting vertex of a DFS. Assume that at time $t-1$, the gray path is of the form $\langle i_0 = s, \ldots, i_k = u, i_{k+1} = v \rangle$, and that at time t, vertex v gets finished, and thus the gray path is now of the form $\langle i_0 = s, \ldots, i_k = u \rangle$. Let the adjacency list of vertex u be $\langle l_1, \ldots, l_{q-1}, l_q = v, l_{q+1}, \ldots, l_r \rangle$. DFS has backtracked from v to u and now we want to find the first unvisited, white vertex x among $l_{q+1}, \ldots, l_r$ in order to visit x next. If we find out that such an x does not exist, we backtrack further from u.

Suppose $y \in \{l_{q+1}, \ldots, l_r\}$. We can determine whether y has been visited or not, that is, whether y is black, gray or white using the following lemma.

Lemma 10 (Black Vertex in a Directed Graph) *Vertex y is black at time t if and only if there exist $j \in \{0, \ldots, k\}$ and vertex α such that the following hold:*

1. *The directed edge (i_j, α) exists.*
2. *In the adjacency list of i_j, vertex α precedes vertex i_{j+1}.*
3. *Vertex y is reachable from vertex α without going through any of the vertices $\{i_0, \ldots, i_j\}$.*

Proof. Vertex y is black if and only if the path from s to y in the DFS tree is lexicographically smaller than the path from s to u. We can easily finish the proof using the Lemma 3 (All-White Path). □

For undirected graphs, we can simplify the lemma above as follows.

Lemma 11 (White-Black Not Adjacent in Undirected DFS) *During a DFS in an undirected graph, a white vertex is never adjacent to a black vertex.*

Proof. Initially, the property holds, and the property is maintained during DFS since a vertex becomes black only when all of its neighbors are non-white. □

Lemma 12 (Black Vertex in an Undirected Graph) *Vertex y is black at time t if and only if the following holds: There exists a vertex $w \in \{l_1, \ldots, l_{q-1}\}$ such that y is reachable from w without going through $\{i_0, \ldots, i_k\}$.*

Proof. When vertex u is first visited from i_{k-1}, none neighbor of us is black by Lemma 11 (White-Black Not Adjacent in Undirected DFS), and hence j and α, as required in Lemma 10 (Black Vertex in Directed Graph), cannot exist if $j < k$. □

4.5 $O(n)$ bits DFS Algorithms

Our algorithms maintain the color of each vertex. For example, our 4-color algorithm uses 2 bits per vertex to hold the current color and thus uses $2n$ bits in total for color information. Any additional space used is $o(n)$ bits, and thus the space used to maintain the colors dominates the space complexity of our algorithms.

A basic problem that we face when restricted to $O(n)$ bits work-space is that we cannot store, for example, the ordered list of the vertices that are currently gray since that would require work-space of size $\Theta(n \log n)$ bits. A basic solution is to retrieve information by restarting the search from the starting vertex.

Algorithm 1: a 4-Color Algorithm. Our first algorithm, Algorithm 1, uses 4 colors for each vertex. It uses white, gray, and black according to the definitions of these colors explained in Section 2. To backtrack, it retraces the current gray path using Lemma 5 (Gray Path from Gray Set) by using one new color, blue, to keep track of the gray vertices in the initial segment reconstructed so far. We describe the algorithm in greater detail. Initially the starting vertex is colored gray and all other vertices are colored white. Suppose during the DFS we are at a vertex u (which must then be gray). If u has a white neighbor then we proceed with the search going to the first white neighbor of u, which is then colored gray. If u has no white neighbor, we color u black and backtrack. When backtracking, in case u is the starting vertex the search ends. Otherwise we need to determine the parent of u which is done by retracing the gray path as follows. We color the starting vertex blue. Then we repeatedly find the smallest gray vertex v that is a neighbor of the last vertex that was colored blue until this vertex is u. When u is colored blue, the parent of u is the last vertex that was colored blue just before that. We then recolor all blue vertices to be gray (by once again retracing the path), recolor u to be black, and have successfully backtracked.

Theorem 13 *Given an undirected or directed graph G consisting of n vertices and m edges, DFS on G can be done in $O(mn)$ time and in work-space $2n + O(\log n)$ bits.*

Proof. As explained in Section 4.3, the total number of color changes is $2n$. Between any two color changes, each edge is inspected at most $O(1)$ times. □

Algorithm 2: a Faster 4-Color Algorithm. We can speed up Algorithm 1 while still using only $O(n)$ bits as follows. Algorithm 2 uses a double-ended queue (DEQ) holding $O(n/\log n)$ items, where each item in the queue is the $O(\log n)$-bit name of some vertex. The queue holds the most recently visited $O(n/\log n)$ gray vertices.

Backtracking to vertex u can be done in $O(1)$ time if the queue holds u, that is, if the queue is nonempty. When the queue becomes empty, we reconstruct the gray path in $O(m)$ time, but such reconstructions happen at most $O(\log n)$ times. Thus we have the following.

Theorem 14 *DFS on undirected and directed graphs can be done in $O(m \log n)$ time and in $O(n)$ bits.*

Remark 1: When using space s, where $n \leq s \leq n \log n$, the algorithm above can be adjusted to run in time $O(m\, n \log n/s)$. Thus this algorithm is *memory-adjustable* in the sense of [8], [9], and [49].

Algorithm 3: a 3-Color Algorithm. By repeatedly restarting, we can reduce the number of colors by one. In each iteration, Algorithm 3 identifies *one* new black vertex. Starting from vertex s, we proceed using colors white, gray and black until we find the first gray vertex v that is now changing its color from gray to black. At such a point, we globally update the color of v as black, change the color of all the other gray vertices *from gray back to white*, and start a new iteration, again from s. Correctness of Algorithm 3 follows from Lemma 4 (Gray Path from Black Vertices).

Remark 2: In the description of Algorithm 3 above, vertex x being white does not always imply that x has never been visited: Even when x has been visited, the color of x becomes white again in a new iteration on the black-or-white graph.

With Algorithm 3 have obtain following theorem.

Theorem 15 *For every $\varepsilon > 0$, DFS on undirected and directed graphs can be done in $O(mn)$ time and in $(\log_2(3) + \varepsilon + o(1))n$ bits.*

Three Situations of a DFS. To describe our next two algorithms we think of the following three situations in which a DFS algorithm can be:

1. **First visit:** This situation arises if a vertex v has just been visited for the first time. The successor of v will be the first white vertex in the adjacency list of v if such a vertex exists. Otherwise we backtrack.

2. **Backtrack:** When vertex v becomes black, we backtrack to the parent vertex u.

3. **Pivot:** After backtracking from v to u, if the adjacency list of u is $\langle l_1, \ldots, l_q = v, l_{q+1}, \ldots, l_r \rangle$, we wish to find the first white vertex x among $l_{q+1}, \ldots, l_r$, and visit x next if such an x exists; otherwise we backtrack further.

We now describe algorithms in terms of how they proceed in each of the three situations.

Algorithm 4: a 2-Color Algorithm. Our two color algorithm maintains the gray path but does not distinguish between black and white vertices. The three situations of the DFS are handled in the following way.

1. First visit: When v has just been visited for the first time, the color of v had been white and has just become gray. This situation is essentially a special case of the situation Pivot except that there is no vertex v from which we have just returned. To reduce this situation to the pivot situation, we simply pretend that there is an auxiliary black vertex v not part of the actual input graph incident only to the edge (u, v) from which we have just backtracked to u. For this, the vertex v is treated as the first neighbor of u.

2. Backtrack: In order to backtrack we memorize the current vertex u and then retrace the gray path from the starting vertex s to u. To do so, we first reinitialize so that all vertices are white. Using Corollary 7 (Gray Path Reconstruction), we iteratively reconstruct the gray path from the starting vertex s to u, and thereby find u's parent. In applying Corollary 7 we use the Reachable routine and one color, gray.

3. Pivot: Using Corollary 7 (Gray Path Reconstruction) together with Lemmas 10 and 12 (Black Vertex in Directed/Undirected Graph), we find the white vertex x to be visited next by using the Reachable routine and using one color, gray. If no such vertex x exists, we perform a backtracking step.

The space used by Algorithm 4 is n bits plus the space used by the routine Reachable.

Algorithm 5: an algorithm without colors for DAGs. For the case of DAGs, we do not need to use any color. The algorithm copies the behavior of Algorithm 4, which in the case the input graph is a DAG simplifies as follows.

1. First visit: As in the case of Algorithm 4, this situation reduces to the pivot situation.

2. Backtrack: Similar to the Backtrack situation of Algorithm 4, we can just retrace the gray path from the starting vertex s to the current vertex u. Applying Corollary 9 instead of Corollary 7 we always use Reachable(x, u, G) instead of Reachable$(x, u, G-\{i_0, \ldots, i_{j-1}\})$, thus avoid using any colors.

3. Pivot: Again Similar to the respective situation in Algorithm 4, we can follow the gray path by reconstructing one gray edge at a time, thereby forgetting the previous gray edges, and by then invoking Reachable(x, u, G).

With Algorithms 4 and 5 and Theorems 1 and 2, we can conclude as follows.

Theorem 16 *(1) DFS on a directed graph can be done in $n+n/2^{\Omega(\sqrt{\log n})}$ bits and in polynomial time.*
(2) DFS on an undirected graph can be done in $n + O(\log n)$ bits and in polynomial time.
(3) DFS on a DAG can be done either in work-space $n/2^{\Omega(\sqrt{\log n})}$ bits and in polynomial time or in work-space $O(\log^2 n)$ bits and in time $n^{O(\log n)}$.

4.6 Tree-Walking

In this section, we consider undirected trees and forests. We give a simple $O(\log n)$ bits algorithm for the depth-first traversal of a tree. The algorithm can be extended to an $O(\log n)$ bits algorithm for deciding whether two given vertices are connected in a given forest and to an $O(\log n)$ bits algorithm for deciding if a given undirected graph contains a cycle.

Throughout the section, we assume that each vertex u holds a cyclic list of its neighbors, and $Next_u(v)$ denotes the vertex that immediately follows vertex v in the cyclic list of vertex u. We can follow a tree in the depth-first order starting from any edge (u, v) as follows:

procedure EdgeFollow(u, v)
  $c = 0$; $(u_0, v_0) = (u, v)$;
  repeat
    Output the edge (u, v);
    $w = Next_v(u)$;
    $u = v$; $v = w$; $c = c + 1$;
  until $u = u_0$ and $v = v_0$;
  return c;

Lemma 17 *Let G be a tree of n vertices and (u, v) be any edge. Then, the procedure EdgeFollow(u, v) returns $2n - 2$ after visiting every edge of G exactly twice.*

Proof. We prove the lemma by induction on n. Let $Adj(u) = (v = v_0, v_1, \ldots, v_a)$ and $Adj(v) = (u = u_0, u_1, \ldots, u_b)$ be the cyclic adjacency list of u and v, respectively. We also assume that removal of (u, v) from G results in two trees T_u and T_v, where T_u (resp. T_v) is the tree containing the vertex u (resp. v). By the induction hypothesis, if we apply the procedure EdgeFollow(u, v) to the tree $(u, v) + T_v$, then it returns $2|T_v|$ after traversing all the edges in $(u, v) + T_v$. Similarly, applying the procedure EdgeFollow(v, u) to the tree $(v, u)+T_u$, it traverses all the edges of T_u and returns $2|T_u|$. Since $Next_u(v) = v_1$ and $Next_v(u_b) = u$, we can combine the two edge sequences produced by EdgeFollow(u, v) and EdgeFollow(v, u) to obtain the complete sequence of the depth-first search on T. The lemma follows since $|T_u \cup T_v| = n$ and the edge (u, v) is contained twice in each of the sequences. Note that the exceptional cases when T_u or T_v is empty are dealt with appropriately. □

Theorem 18 *Using Edge-Follow, deciding s-t connectivity in a forest and detecting a cycle in an undirected graph can both be done in $O(\log n)$ bits and in $O(n)$ time.*

4.7 DFS in $O(\log n)$-Space for undirected Graphs with $O(1)$-Size Feedback Vertex Set

Now we consider the case where the input is an undirected graph $G = (V, E)$ having feedback vertex set F of constant size. A *feedback vertex set* $F \subseteq V$ of G is a set of vertices such that $G - F$ contains no cycle. If a graph has a feedback vertex set of constant size, we can easily find a constant-size feedback vertex set in polynomial time and logarithmic space $O(|F| \log n)$. Thus, from now on, assume that we are given an undirected graph $G = (V, E)$ and a feedback vertex set $F \subseteq V$ of constant size. Furthermore, without loss of generality, assume that the set F includes a DFS-starting vertex s. (We may always add s to F if $s \notin F$.) Note that $G - F$ is a forest.

Our overall strategy will be similar to Algorithm 4. By exploiting the fact that $G - F$ is a forest, we can proceed similarly as in Algorithm 4 without remembering all the gray vertices and without using any color for reconstructing the gray path.

In particular, for the current gray path P, we only keep in memory the following parts of P: the vertices x in the feedback set F that appear in P, together with appearance order information,

and for each such x, the non-feedback vertices that immediately precede and follow x in P if such vertices exist.

For simplicity of explanation, assume that precisely four vertices in the set F, namely vertices i, j, k and l, appear in the current gray path P in this order. Then, we will keep in memory the following parts of path P as our data (We do *not* keep in memory the parts corresponding to "…".):

$$i, i_1, \ldots, j_0, j, j_1, \ldots, k_0, k, k_1, \ldots, l_0, l, l_1, \ldots, l_c,$$

where all of the above are in the order of appearance in P, i is the starting vertex s, l_c is the vertex where the current head lies, and, for example, j_0, j, and j_1 are consecutive vertices in the path P. It may happen that, for example, $i_1 = j_0$, or that neither i_1 nor j_0 exist (in this case two feedback vertices i and j appear consecutively in P). Since the size of the feedback vertex set F is constant, the data above has size only $O(\log n)$.

How can we reconstruct the whole gray path P from the data above? Consider, for example, the vertices appearing between j_1 and k_0 in the path P. These vertices, together with the appearance ordering, can be characterized as vertices appearing in the unique simple path connecting vertices j_1 and k_0 in the forest $G - F$.

Note that we can follow the vertices appearing in path P consecutively, one at a time, using only the tree-walking algorithm and without using Reingold's *s-t* connectivity algorithm. We can thus in space $O(\log n)$ bits determine at any point in time whether a vertex is gray and within the same space complexity compute the predecessor of a vertex on the gray path.

In analogy to Algorithm 4 it remains to explain how to perform the pivot situation in a DFS. In order to proceed as in Algorithm 4, we have to be able to check whether vertices y and z are connected in $G - C$, where C is the set of the vertices that are currently gray and vertices y and z are two neighbors of the vertex u to which we have just backtracked.

We claim that connectivity checking for the restricted case of constant-size feedback vertex set can be done using only the tree-walking algorithm (again without using Reingold's *s-t* connectivity algorithm) as follows. Since paths without internal vertices from the feedback vertex set and without gray vertices can be found using the tree-walking algorithm, it suffices to show that for each pair of vertices in the feedback vertex set we can determine whether they are connected in $G - C$.

For each pair of feedback vertices f and g, we can determine, using tree-walking, whether f and g are connected by a path going only though non-feedback vertices that are not gray. So, by first checking whether f and g are connected in $G - (C \cup F)$, where C is the set of vertices that are currently gray, and taking the transitive closure, we can obtain the complete table as to which pairs of feedback vertices are connected in $G - C$. Note that since F has constant size, taking the transitive closure can be done in space $O(\log n)$ bits. We conclude the following.

Theorem 19 *For undirected graphs whose minimum feedback vertex set are of constant size, DFS can be done in $O(\log n)$ bits using only the tree-walking algorithm (i.e., without appealing to Reingold's s-t connectivity algorithm).*

Chapter 5

Adjustable work-space algorithms for segment intersection problems

This Chapter presents an efficient algorithm for reporting all intersections among n given segments in the plane using work-space of arbitrarily given size. More exactly, given a parameter s which is between $\Omega(1)$ and $O(n)$ specifying the size of work-space, the algorithm reports all the segment intersections in roughly $O(n^2/\sqrt{s} + K)$ time using $O(s)$ words of $O(\log n)$ bits, where K is the total number of intersecting pairs. The time complexity can be improved to $O((n^2/s)\log s + K)$ when input segments have only some number of different slopes.

5.1 Segment Intersection Detection

Segment intersection detection is a problem of determining whether there is any pair of mutually intersecting segments in an input set of segments in the plane. A simple and efficient algorithm [67] is known for the problem. The algorithm sweeps the plane while visiting each endpoint of input segments in the sorted order and detects an intersection if any. It runs in $\Theta(n \log n)$ time for any set of n segments in the plane using $O(n)$ words.

We design an efficient adjustable work-space algorithm using $O(s)$ words for segment intersection detection a given set of segments stored in a read-only array. A variable s is between $\Omega(1)$ and $O(n)$. This algorithm becomes basis of our algorithm for reporting segment intersections. See sections 5.2 and 5.2.2.

The algorithm first partitions an input set S into $m = n/s$ disjoint subsets $S_1, S_2, \ldots, S_m$. Whenever s does not divide n we add extra dummy segments . So we assume that each subset has exactly s segments. In practice we just compute the size m of the partition. Since the input segments are stored in a read-only array, this partition is done in the index order, that is, S_1 contains the first s segments in the array, S_2 consists of the next s segments, and so on. Then, for each pair (S_i, S_j) with $i < j$ we perform a plane sweep to detect any intersection among given segments in the set $S_i \cup S_j$. It is done in $O(s \log s)$ time using $O(s)$ words work-space by a standard plane sweep algorithm. Since we have $O((n/s)^2)$ different pairs, the algorithm runs in $O((n^2/s)\log s)$ time. The algorithm is referred to as Algorithm 1, where a function BentleyOttmanPlaneSweep() is a function which implement a standard plane sweep algorithm by Bentley and Ottman [19].

Algorithm 1: Segment intersection detection

```
Partition the set S into m = n/s disjoint subsets S_1, S_2, ..., S_m using Index Partition.
for each pair of subsets (S_i, S_j) ,i, j = 1 ... m do
    apply BentleyOttmanPlaneSweep(S_i ∪ S_j).
    if any intersection is found then
        stop after reporting the intersection.
    end
end
stop after reporting "No intersection."
```

Theorem 20 *Given n segments in the plane in a read-only array and a parameter value s between $\Omega(1)$ and $O(n)$, Algorithm 1 correctly determines whether there is any intersection among input segments in $O((n^2/s)\log s)$ time using $O(s)$ words.*

More generally, we propose two different ways of partitioning a given set S of n segments. Since these methods are simple, it may apply the other problems using limited work space.

Index Partition: A given set S of n elements is partitioned into $m = n/s$ disjoint subsets $S_1, \ldots, S_m$ by the indices, that is, S_1 consists of the first s elements in the array storing S, S_2 of the next s elements, and so on. If s does not divide n, then we add extra dummy segments.

Property Partition: We are given a set S of n elements and c properties for the elements. Then, S is partitioned into disjoint subsets $S_1, S_2, \ldots S_r$, so that $|S_i| \leq s$ and all elements of S_i are the same have the same property for each $i = 1, \ldots r$. The number r of subsets is bounded by $n/s + c$, i.e. $r \leq n/s + c$.

The index partition is simple since only index calculation is needed. To have a property partition we scan the input array while checking properties of the elements. Thus, it takes $O(cn)$ time to enumerate all the subsets. In this paper we take slopes of segments as properties.

5.2 Segment Intersection Reporting

In this section we consider the segment intersection reporting problem. We report all intersecting pairs among a given set of segments in the plane. It is not so easy to design an algorithm which it runs in an output sensitive manner. More exactly, if we denote by K the total number of intersecting pairs, the computation time should be $T(n, s) + O(K)$, where $T(n, s)$ only depends on the number n of segments and the size s of work-space. The number of segment intersections could be $\Theta(n^2)$ in the worst case. If we have $\Omega(n^2)$ intersections, then a brute-force algorithm of examining all pairs of segments suffices. Of course, the brute-force algorithm is not optimal unless $K = \Omega(n^2)$.

The segment intersection reporting problem has been well studied. The first algorithm by Bentley and Ottman [19] based on plane sweep runs in $O((n + K)\log n)$ time using $O(n + K)$ words of work-space. It is not so hard to reduce the space to $O(n)$ while keeping the time complexity. The first output-sensitive algorithm for the problem was given by Mairson and Stolfi [53] under the name of red-blue intersection. In the problem we are given two sets of segments colored red or blue. Assuming there is no intersection among segments of the same color, they gave an optimal algorithm which reports all the intersections in $O(n \log n + K)$ time

using $O(n)$ words. The result was strengthened by Chazelle and Edelsbrunner [23] and further by Balaban [15]. Although the algorithm by Chazelle and Edelsbrunner needs $O(n + K)$ words, Balaban's algorithm uses only $O(n)$ words and thus it is theoretically optimal.

Very little has been studied so far when work-space is limited to $o(n)$ words. A space-efficient algorithm is presented by Chen and Chan [26] , which runs in $O((n + K)\log^2 n)$ time using only $O(\log n)$ words extra work-space assuming that input segments are stored in a regular read/write array and thus the array can be used as a work-space. Especially, a technique referred to as an implicit data structure proposed by Munro [58] can be used. In our paper, however, such a technique cannot be used since input data is stored in a read-only array. Throughout this paper we assume that no two segments overlap.

5.2.1 Isothetic Segments

We begin with a simple situation where input segments are isothetic segments. For the time being we assume that no two of them overlap. More exactly, we assume that for any two segments ℓ_i and ℓ_j of the same direction (horizontal or vertical) no endpoint of ℓ_i lies on ℓ_j. Under this assumption it is rather easy to design an algorithm for reporting all segment intersections using only $O(s)$ words in addition to a read-only array storing n input segments.

We can design an algorithm reporting segment intersections by slightly modifying Algorithm 1. It is shown in Algorithm 2.

Algorithm 2: Segment Intersection Reporting for a set of isothetic segments

```
   /* Preprocessing stage:                                              */
 1 Partition the set S into m = n/s disjoint subsets S_1, S_2, ..., S_m using Index Partition.
   /* 1st stage:  Reporting all intersections within each subset        */
 2 for each subset S_i, i = 1...m do
 3 |   report all segments intersecting by BentleyOttmanPlaneSweep(S_i).
 4 end
   /* 2nd stage:  Reporting all intersections between two subsets       */
 5 for each pair (S_i, S_j) of subsets , i, j = 1...m do
 6 |   U ← all horizontal segments in S_i and all vertical segments in S_j.
 7 |   report all segments intersecting by BentleyOttmanPlaneSweep(U).
 8 |   U' ← all vertical segments in S_i and all horizontal segments in S_j.
 9 |   report all segments intersecting by BentleyOttmanPlaneSweep(U').
10 end
```

Theorem 21 *Given n isothetic segments in the plane stored in a read-only array and a parameter value s between $\Omega(1)$ and $O(n)$, Algorithm 2 correctly reports all K intersections between input segments in $O((n^2/s)\log s + K)$ time using $O(s)$ words.*

Proof. Due to the assumption that no two segments overlap each other, no two horizontal (resp., vertical) segments intersect. Thus, after reporting all intersections within each subset, all the remaining intersections are made by two isothetic segments from different subsets. Thus, all the intersections are correctly reported. Since every intersection is reported exactly once, the algorithm runs in $O((n^2/s)\log s + K)$ time. □

5.2.2 Algorithm Using Property Partition

In the algorithms above we have partitioned a given set of n isothetic segments into n/s subsets by index partition. Then, each subset was further decomposed into a set of horizontal segments and one of vertical segments. In the second stage we take a pair (S_i, S_j) and perform a standard plane sweep for the two sets $U_1 = H(S_i) \cup V(S_j)$ and $U_2 = V(S_i) \cup H(S_j)$. $H(S_i)$ (resp. $V(S_i)$) denotes a set of all horizontal (resp. vertical) segments in S_i. Since the partition into subsets is done only by indices, it may happens that $U_1 = \emptyset$ and $U_2 = S_i \cup S_j$ or $U_1 = S_i \cup S_j$ and $U_2 = \emptyset$. It means that we may have a set of segment of so different sizes for plane sweep in the second stage.

Here is a simple way of keeping the set size. When an input set S of n isothetic segments is given, we use the property partition described before. The property we use is the slope of segment, horizontal or vertical. Using the property we partition a given set S into m_h subsets $H_1, H_2, \ldots, H_{m_h}$ and m_v subsets $V_1, V_2, \ldots, V_{m_v}$. Each H_i contains only horizontal segments and each V_j contains only vertical segments. Every $H_i, 1 \le i \le m_h$ consists of exactly s horizontal segments in the index order. Just the same for $V_1, \ldots, V_v$. Due to the definition $m_h + m_v = n/s$.

At the second stage we take two subsets H_i and V_j. In the previous algorithms the subsets were obtained just by computing indices. In this case, however, we maintain two pointers (indices), one for H_i and the other for V_j. The pointer for H_i keeps the last horizontal segment of H_i. If the last segment for H_{i-1} is ℓ_p, then the pointer starts from $p + 1$ and then we examine segments $\ell_{p+1}, \ell_{p+2}, \ldots$ by incrementing the pointer until we get s horizontal segments.

Theorem 22 *Given n isothetic segments in the plane stored in a read-only array and a parameter value s between $\Omega(1)$ and $O(n)$, Algorithm 3 correctly reports all K intersections between input segments in $O((n^2/s)\log s + K)$ time using $O(s)$ words.*

Proof. We partition a set S of n segments into m_h subsets $H_1, \ldots, H_{m_h}$ and m_v subsets of $V_1, \ldots, V_{m_v}$ as above. Now it is obvious that the algorithm reports all K intersections in $O(m_h m_v s \log s + K)$ time. Since $m_h m_v \le (n/(2s))^2$, its worst running time is still $O((n^2/s)\log s + K)$. □

Algorithm 3: Segment Intersection Reporting for a set of isothetic segments using monocolor partition.

```
/* H_i has s horizontal segments.                                   */
/* V_j has s vertical segments.                                     */
1 Partition the set S into m_h + m_v = n/s disjoint subsets H_1, ..., H_{m_h}, V_1, ..., V_{m_v} using
  Property Partition.
2 for each subset H_i, i = 1 ... m_h do
3 |   for each subset V_j, j = 1 ... m_v do
4 |   |   report all segments intersecting by BentleyOttmanPlaneSweep(H_i ∪ V_j).
5 |   end
6 end
```

5.2.3 Algorithm Using Filtering Search

There is another way of achieving the running time $O((n^2/s)\log s + K)$. We use the partition by index $(S_1, S_2, \ldots, S_m)$, $m = n/s$. For each subset S_i, we take all horizontal segments in S_i and put them into a data structure so that for any query vertical segment ℓ_q all k intersections of ℓ_q

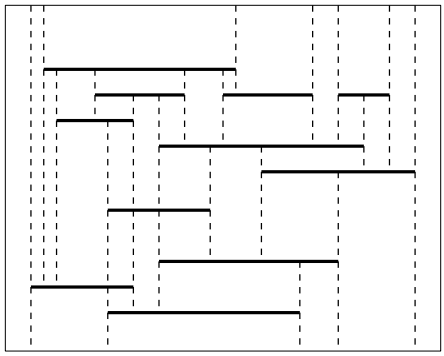

Figure 5.1: Trapezoidal decomposition associated with a set of horizontal segments.

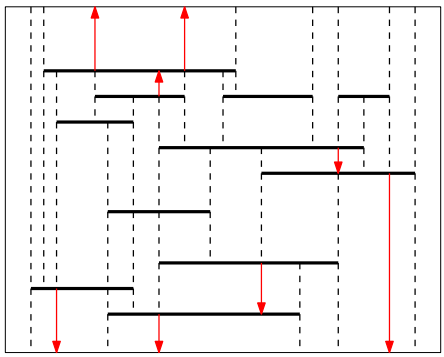

Figure 5.2: Trapezoidal decomposition for filtering search in which none of top and bottom sides of rectangles is incident to more than two vertical edges.

with those horizontal segments in S_i can be reported in $O(k + \log s)$ time. The data structure we use is a **trapezoidal decomposition and filtering search [25]**.

Given a set $H(S_i)$ of at most s horizontal segments, we first compute a sufficiently large rectangle enclosing all the given segments and then extend rays from each endpoint of those segments until they hit an input segment or the boundary (see Figure 5.1). The resulting planar subdivision into rectangles is called the trapezoidal decomposition. If we incorporate a data structure for point location and a graph representing vertical adjacency of those rectangles, then we can report intersections on an arbitrarily given query vertical segment ℓ_q by first locating one of its endpoints and then following the adjacency graph. Unfortunately, this algorithm is not good enough to achieve our target running time. Suppose we have located the lower endpoint of ℓ_q. Then, we want to find a rectangle just above the current rectangle. If a rectangle is vertically adjacent to many rectangles, then it takes time to find the rectangle intersecting the query segment. In the data structure defined by Chazelle [25] we add chords (vertical sides) so that none of top and bottom sides of rectangles is incident to more than two vertical sides. The trapezoidal decomposition shown in Figure 5.1 is modified using arrowed chords in Figure 5.2.

This problem has been extensively studied. From a theoretical point of view, Chazelle [23] presented an algorithm with $O(s \log s)$ preprocessing time, $O(s)$ words, and $O(k + \log s)$ search time. It is theoretically optimal with respect to the worst case.

In this paper we use the Chazelle's data structure outlined above. In addition we use his filtering search technique to achieve the target search time with the linear size of the data structure.

Algorithm 4: Segment Intersection Reporting for a set of isothetic segments using trapezoidal decomposition with filtering search.

```
   /* Preprocessing stage:                                                    */
 1 Partition the set S into m = n/s disjoint subsets S_1, S_2, ..., S_m using Index Partition.
 2 for each subset S_i, i = 1 ... m do
 3 |   Let H(S_i) be a set of all horizontal segments in S_i.
 4 |   Build a data structure TD_i by trapezoidal decomposition and filtering search for
   |   H(S_i).
 5 |   for each segment ℓ_q in S do
 6 |   |   if ℓ_q is vertical then
 7 |   |   |   Report all intersections of ℓ_q with H(S_i) using the data structure TD_i.
 8 |   |   end
 9 |   end
10 end
```

Theorem 23 *Given n segments in the plane stored in a read-only array and a parameter value s between* $\Omega(1)$ *and* $O(n)$*, Algorithm 4 correctly reports all K intersections between input segments in* $O((n^2/s)\log s + K)$ *time using* $O(s)$ *words.*

Proof. Each subset S_i contains at most s horizontal segments. We can build the trapezoidal decomposition with filtering search in $O(s \log s)$ time using $O(s)$ words. Then, for each vertical segment ℓ_q we can report all k intersections of ℓ_q with those in the data structure in $O(\log s)$ time, thus in total $O(K_i + n \log s)$ time, where K_i is the number of intersection reported for S_i. Hence, the total running time is given by

$$\sum_i O(K_i + n \log s) = O(K + (n^2/s)\log s).$$

□

5.2.4 Segment Overlaps

Now, we assume that there are overlaps among segments of the same slope. Algorithm 2 still works, but it is not output sensitive anymore. Suppose the first subset S_1 has two horizontal segments ℓ_p and ℓ_q which overlap each other. Then, if we apply Algorithm 2, it reports the intersecting pair (ℓ_p, ℓ_q) in the first stage. Then, in the second stage it examines pairs $(S_1, S_2), (S_1, S_3), \ldots, (S_1, S_m)$ to perform the plane sweep. For each pair the intersection (ℓ_p, ℓ_q) is detected. Thus, we have to know how to avoid such an intersection due to overlap in the second stage.

Now we have three problems:

Isothetic Segment Intersection Reporting: Given a set of isothetic segments, report all intersections of those of different directions.

Horizontal Segment Overlap Reporting: Given a set S of horizontal segments and a query horizontal segment ℓ_q, report all segments in S which overlap ℓ_q.

Vertical Segment Overlap Reporting: Given a set S of vertical segments and a query vertical segment ℓ_q, report all segments in S which overlap ℓ_q.

Since the last two problems are just symmetric, we just consider Horizontal Segment Overlap Reporting Problem. Once we report all overlaps, we just apply our previous algorithms to report all intersections of isothetic segments.

Horizontal Segment Overlap Reporting

We are given a set S of n isothetic segments. We have seen two different ways of decomposing S into subsets of size $O(s)$, one by indices and the other by slopes. In either way we have $O(s)$ horizontal segments and we want to report all overlaps with them. For that purpose we first build a binary search tree using their y-coordinates as keys. Then, two or more horizontal segments may have the same y-coordinate. We create just one leaf node for those segments sharing the same y-coordinate. We apply the Chazelle's filtering search technique to those segments [25]. Since all those horizontal segments have the same y-coordinate, they can be regarded as intervals. Thus, given a query interval, we can report all k intervals in the data structure intersecting the query one in $O(k + \log s)$ time.

Theorem 24 *Given n segments without any overlap in the plane stored in a read-only array and a parameter value s between $O(1)$ and $O(n)$, Algorithm 5 correctly reports all K_h overlaps between input horizontal segments in $O((n^2/s)\log s + K_h)$ time using $O(s)$ words.*

Proof. Each subset S_i contains at most s horizontal segments. We can build the binary search tree in $O(s \log s)$ time. Then, for each leaf node containing more than one segment we reform them for filtering search. It is done in in linear time using linear space (since all the segment endpoints can be sorted in the preprocessing step). Then, we can report all K_{hi} interval overlaps in an output sensitive way using the filtering search. Hence, the total running time is given by

$$\sum_{\text{each } S_i} O(s \log s + K_{hi} + n \log s) = O(K_h + (n^2/s)\log s).$$

□

Algorithm 5: Horizontal Segment Overlap Reporting for a set of horizontal segments using the filtering search.

Input: A set S of n horizontal or vertical line segments in the plane stored in a read-only array and a parameter value s.
Output: All K_h overlaps among the input horizontal line segments.

```
/* Preprocessing stage:                                                    */
```

1 Partition the set S into $m = n/s$ disjoint subsets $S_1, S_2, \ldots, S_m$ using Index Partition.
2 **for** *each subset* S_i **do**
3 Let $H(S_i)$ be a set of all horizontal segments in S_i.
4 Build a binary search tree using y-coordinates of those horizontal segments.
5 At each leaf node containing more than one segment, build the data structure for filtering search. **for** *each horizontal segment* ℓ_q *in* $S_i \cup \cdots \cup S_m$ **do**
6 Report all overlaps of ℓ_q with $H(S_i)$ except (ℓ_q, ℓ_q) using the data structure.
7 **end**
8 **end**

Symbolic Perturbation

Once we have reported all overlaps among segments of the same slope (horizontal or vertical), we report all intersections between horizontal and vertical segments. What is required here is to avoid reporting segment overlaps among those of the same slope.

It is easy. For example, Algorithm 2 can be adapted as follows. In the first stage we do not need to report segment overlaps. For the purpose we vertically shift each horizontal segment, say ℓ_p, by $p\delta$ for a positive small constant δ. For each vertical segment, we extend it upper endpoint upward by $n\delta$. If the constant δ is small enough, this extension causes no problem. More exactly, this modification does not create any new intersection. After the modification we perform a standard plane sweep.

In the plane sweep algorithm we have three different types of events caused by left and right endpoints of horizontal segments and vertical segments. At a left endpoint of a horizontal segment, say ℓ_p, we insert ℓ_p into a data structure called a y-list which keeps all horizontal segments intersecting the current vertical sweep line in the sorted order. We order them by the lexicographical order using their y-coordinates and indices. That is, when two horizontal segments ℓ_p and ℓ_q overlap each other and $p < q$, we decide $\ell_p < \ell_q$. Or, more formally, for two horizontal segments ℓ_p and ℓ_q, $\ell_p < \ell_q$ holds if the y-coordinate of ℓ_p is smaller than that of ℓ_q or they are equal and $p < q$. This is equivalent to shifting each horizontal segment ℓ_p vertically by $p\delta$ for a small positive constant $\delta > 0$.

At a right endpoint of a horizontal segment, say ℓ_q, we just delete ℓ_q from the y-list. When the sweep line comes to a vertical segment, say ℓ_r, we locate its two endpoints in the y-list. For the lower endpoint, we use its y-coordinate so that it becomes below any horizontal segment of the same y-coordinate if any. For the upper endpoint, we use its y-coordinate plus $n\delta$ so that it lies above any horizontal line of the same y-coordinate if any. Once we locate the two endpoints in the y-list, we can report all k_r intersections on ℓ_r in $O(k_r + \log s)$ time.

In practice we do not use the constant δ. The same operations are done in a symbolic manner. This simple symbolic perturbation is effective to avoid duplicate report of overlapping segments. The time complexity of the resulting algorithm is just the same as before.

An example is given in Figure 5.3. There are 12 horizontal segments $\ell_1, \ldots, \ell_{12}$ and two vertical segment ℓ_{13} and ℓ_{14}. Among them the three segments $\ell_5, \ell_8, \ell_{10}$ and two endpoints (the upper endpoint of ℓ_{13} and the lower endpoint of ℓ_{14}) have the same y-coordinate in the figure. After the modification stated above, the three horizontal lines are vertically shifted by their indices. The upper endpoint of ℓ_{13} lies above all of them and the lower endpoint of ℓ_{14} below them.

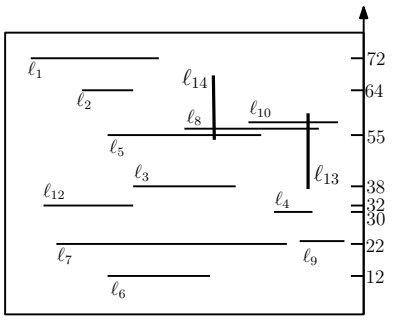

Figure 5.3: A set of horizontal segments with overlap. Three segment $\ell_{10}, \ell_8, \ell_5$ have the same y-coordinate 55, but they are slightly shifted in the figure.

5.2.5 Segments of at Most c Different Slopes

The algorithm for isothetic segments can be extended to a more general case where given segments have at most c different slopes, where c is $O(n/s)$. In Algorithm 3 above, by using Property Partition, we can get the subset of given set in which all segments are either horizontal or vertical. The number c of slope is two because given segments are isothetic.

Even segments in given set S have at most c different slopes, the Property Partition is convenient (see Algorithm 6). A given set S is partitioned into disjoint different subsets $S_1, S_2, \ldots, S_r$ so that $|S_i| \leq s$ and all segments of S_i have the same slope for each $i = 1, \ldots, r$. Then, for each pair (S_i, S_j), we apply the plane sweep algorithm for the union of the two sets to report all intersections. If the slope of segment in S_i and S_j is same, no intersections are reported. Since no two segments of the same slope intersect or overlap.

Algorithm 6: Segment intersection reporting for a set of segments of at most c different slopes.

```
   /* Preprocessing stage:                                              */
1  Partition the set S into disjoint subsets S_1, S_2, ..., S_r using Property Partition, where
   r ≤ n/s + c.
2  for each subset S_i, i = 1 ... r do
3  |   for each subset S_j, j = j + 1 ... r do
4  |   |   report all segments intersecting by BentleyOttmanPlaneSweep(S_i ∪ S_j).
5  |   end
6  end
```

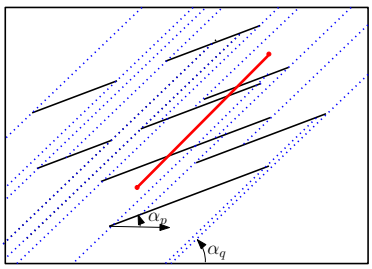

Figure 5.4: Trapezoidal decomposition defined by two distinct slopes α_p and α_q, where all the segments have the slope α_p and a query segment has the slope α_q.

Theorem 25 *Given n segments of at most c different slopes, where $c = O(n/s)$, in the plane stored in a read-only array and a parameter value s between $\Omega(1)$ and $O(n)$, Algorithm 5 correctly reports all K intersections between input segments in $O((n^2/s + c^2 s)\log s + K)$ time using $O(s)$ words.*

It is not so hard to adapt Algorithm 6 so as to report segment overlaps as well. We can use the same mechanism as before. For each pair of slopes we define a similar trapezoidal decomposition as shown in Figure 5.4. If we rotate segments of those slopes so that they are isothetic each other then the same mechanism works.

5.2.6 General Case

We have efficient algorithms when given segments do not have many different slopes. Unfortunately, none of our algorithms works efficiently without the condition. So, we need a completely

Algorithm 7: Segment intersection reporting for a general set of segments.

```
   /* Preprocessing stage:                                          */
 1 t = s^{1/(1+ε)}.
 2 Partition the set S into m = ⌈n/t⌉ disjoint subsets S_1, S_2, ..., S_m using Index Partition.
   /* 1st stage: Reporting intersections within each subset.        */
 3 for each subset S_i, i = 1...m do
 4 |   report all segment intersections by BentleyOttmanPlaneSweep(S_i)
 5 end
   /* 2nd stage: Reporting intersections between two subsets.       */
 6 for each subset S_i, i = 1...m do
 7 |   Build a data structure D_i for S_i.
 8 |   for each segment ℓ_r ∈ S_{i+1} ∪ ··· ∪ S_m do
 9 |   |   Report all intersections of ℓ_r with those segments in S_i using the data structure D_i.
10 |   end
11 end
```

different idea for a general case.

A key data structure is one proposed by Agarwal and Sharir [1] based on a so-called CSW data structure [24] by Chazelle, Sharir, and Welzl. We use the following result by Agarwal and Sharir [1].

Theorem 26 (Agarwal and Sharir [1]) *Given a collection S of n segments in the plane, a constant $\varepsilon > 0$, and a parameter s with $n^{1+\varepsilon} \leq s \leq n^2$, we can preprocess S into a data structure of size s, in time $O(s^{1+\varepsilon})$, so that, given any query segment ℓ_q, we can report all K segments of S intersecting ℓ_q in time $O(n^{1+\varepsilon}/\sqrt{s} + K)$, or can count the number of such segments in time $O(n^{1+\varepsilon}/\sqrt{s})$.*

We assume that there is no overlap among given segments. A basic framework of our algorithm is just the same as before. After partitioning a given set S into at most $m = n/s$ disjoint subsets $S_1, \ldots, S_m$, in the first stage we report all intersections within each subset, and then in the second stage we report all intersections between segments from distinct subsets. For the second stage, we build a data structure $\mathcal{D}_i$ for each subset S_i given by Agarwal and Sharir mentioned above and then report intersections for each segment in the remaining subset.

Theorem 27 *Given n segments in the plane stored in a read-only array and a parameter value s between $\Omega(1)$ and $O(n)$, Algorithm 6 correctly reports all K intersections between input segments in $O(n^2 s^{-\frac{1-\epsilon}{2(1+\epsilon)}} + K)$ time using $O(s)$ words for any small constant $\varepsilon > 0$.*

Proof. Given n segments and a parameter value s, let t be $s^{1/(1+\varepsilon)}$ for a small constant $\varepsilon > 0$. Then, we partition the set S into $m = \lceil n/t \rceil$ disjoint subset $S_1, \ldots, S_m$, each of $t = O(s)$ segments. For each subset S_i we construct a data structure $\mathcal{D}_i$ of size $O(t^{1+\varepsilon}) = O(s)$ in $O(t^{(1+\varepsilon)^2})$ time Theorem 6 [1]. Using this data structure, we can report all intersections of a query segment ℓ_r with those segments in S_i in time $O(t^{1/2(1+\varepsilon)} + K(S_i, \ell_r))$, where $K(S_i, \ell_r)$ is the number of those intersections of ℓ_r with the segments in S_i. Then, the total running time $T(n)$ is given by

$$\begin{aligned} T(n) &= \sum_{i=1}^{\lceil n/t \rceil} O(t \log t + K(S_i) + t^{(1+\varepsilon)^2} + n t^{\frac{1}{2}(1+\varepsilon)} \\ &\quad + \sum_{\ell_r \in S_{i+1} \cup \cdots \cup S_m} K(S_i, \ell_r)), \end{aligned}$$

where $K(S_i)$ is the number of intersections within the set S_i. Since we have

$$\sum_{i=1}^{\lceil n/t \rceil} K(S_i) + \sum_{\ell_r \in S_{i+1} \cup \cdots \cup S_m} K(S_i, \ell_r) = O(K),$$

we obtain

$$\begin{aligned} T(n) &= O\left(\tfrac{n}{t} t \log t + \tfrac{n}{t} t^{(1+\varepsilon)^2} + \tfrac{n}{t} n t^{\frac{1}{2}(1+\varepsilon)} + K\right) \\ &= O(n \log t + n t^{\varepsilon^2+2\varepsilon} + \tfrac{n^2}{\sqrt{t^{(1-\varepsilon)}}} + K). \end{aligned}$$

Replacing $t^{1+\varepsilon}$ with s, i.e. $t = s^{\frac{1}{\varepsilon}}$, we can obtain

$$T(n) = O\left(\frac{n}{1+\varepsilon} \log s + n s^{\frac{\varepsilon^2+\varepsilon}{1+\varepsilon}} + n^2 s^{-\frac{1-\varepsilon}{2(1+\varepsilon)}} + K\right).$$

□

5.3 Conclusions and Future Works of Chapter 4

In this paper we have presented adjustable work-space algorithms for detecting and reporting intersections among given segments. Those algorithms run in work-space of any size between $\Omega(1)$ and $O(n)$ words, assuming that n input segments are stored in a read-only array. In our conjecture, segments do not have many different slopes in reality. If the number of different slopes is bounded by $O(n/s)$ our algorithms run efficiently. However, if the assumption does not hold, our algorithm has to use a sophisticated data structure which is too impractical. So, one of the most important open problems is to devise a more practical algorithm for the general case.

Chapter 6

Polynomial-Time Algorithms for Subgraph Isomorphism in Small Graph Classes of Perfect Graphs

Given two graphs, Subgraph Isomorphism is the problem of deciding whether the first graph (the *base graph*) contains a subgraph isomorphic to the second graph (the *pattern graph*). This problem is NP-complete for very restricted graph classes such as connected proper interval graphs. Only a few cases are known to be polynomial-time solvable even if we restrict the graphs to be perfect. For example, if both graphs are co-chain graphs, then the problem can be solved in linear time.

In this paper, we present a polynomial-time algorithm for the case where the base graphs are chordal graphs and the pattern graphs are co-chain graphs. We also present a linear-time algorithm for the case where the base graphs are trivially perfect graphs and the pattern graphs are threshold graphs. These results answer some of the open questions of Kijima et al. [*Discrete Math.* 312, pp. 3164–3173, 2012]. To present a complexity contrast, we then show that even if the base graphs are somewhat restricted perfect graphs, the problem of finding a pattern graph that is a chain graph, a co-chain graph, or a threshold graph is NP-complete.

6.1 Introduction

The problem Subgraph Isomorphism is a very general and extremely hard problem which asks, given two graphs, whether one graph (the *base graph*) contains a subgraph isomorphic to the other graph (the *pattern graph*). The problem generalizes many other problems such as Graph Isomorphism, Hamiltonian Path, Clique, and Bandwidth. Clearly, Subgraph Isomorphism is NP-complete in general. Furthermore, by slightly modifying known proofs [37, 28], it can be shown that Subgraph Isomorphism is NP-complete when G and H are disjoint unions of paths or of complete graphs. Therefore, it is NP-complete even for small graph classes of perfect graphs such as proper interval graphs, bipartite permutation graphs, and trivially perfect graphs, while Graph Isomorphism can be solved in polynomial time for them [27, 51]. For these graph classes, Kijima et al. [48] showed that even if both input graphs are connected and have the same number of vertices, the problem remains NP-complete. They call the problem with such restrictions Spanning Subgraph Isomorphism.

Kijima et al. [48] also found polynomial-time solvable cases of Subgraph Isomorphism in which both graphs are chain, co-chain, or threshold graphs. Since these classes are proper subclasses of the aforementioned hard classes, those results together give sharp contrasts of computational complexity of Subgraph Isomorphism. However, the complexity of more subtle cases, like the one where the base graphs are proper interval graphs and the pattern graphs are co-chain graphs, remained open.

Table 6.1: NP-complete cases of Spanning Subgraph Isomorphism.

Base	Pattern	Complexity	Reference
Bipartite Permutation		NP-complete	[48]
Proper Interval		NP-complete	[48]
Trivially Perfect		NP-complete	[48]
Chain	Convex	NP-complete	[48]
Co-chain	Co-bipartite	NP-complete	[48]
Threshold	Split	NP-complete	[48]
Bipartite	Chain	NP-complete	This paper
Co-convex	Co-chain	NP-complete	This paper
Split	Threshold	NP-complete	This paper

Table 6.2: Polynomial-time solvable cases of Subgraph Isomorphism.

Base	Pattern	Complexity	Reference
Chain		$O(m+n)$	[48]
Co-chain		$O(m+n)$	[48]
Threshold		$O(m+n)$	[48]
Bipartite permutation	Chain	Open	
Chordal	Co-chain	$O(mn^2+n^3)$	This paper
Trivially perfect	Threshold	$O(m+n)$	This paper

6.1.1 Our results

In this paper, we study the open cases of Kijima et al. [48], and present polynomial-time algorithms for the following cases:

- the base graphs are chordal graphs and the pattern graphs are co-chain graphs,
- the base graphs are trivially perfect graphs and the pattern graphs are threshold graphs.

We also show that even if the pattern graphs are chain, co-chain, or threshold graphs and the base graphs are somewhat restricted perfect graphs, the problem remains NP-complete. The problem of finding a chain subgraph in a bipartite permutation graph, which is an open case of Kijima et al. [48], remains unsettled. See Tables 6.1 and 6.2 for the summary of our results.

6.1.2 Related results

Subgraph Isomorphism for trees can be solved in polynomial time [57], while it is NP-complete for connected outerplanar graphs [69]. Therefore, the problem is NP-complete even for connected graphs of bounded treewidth. On the other hand, it can be solved in polynomial time for 2-connected outerplanar graphs [50]. More generally, it is known that Subgraph Isomorphism for k-connected partial k-trees can be solved in polynomial time [56, 40]. Eppstein [36] gave a $k^{O(k)}n$-time algorithm for Subgraph Isomorphism on planar graphs, where k and n are the numbers of the vertices in the pattern graph and the base graph, respectively. Recently, Dorn [32] has improved the running time to $2^{O(k)}n$. For other general frameworks, especially for the parameterized ones, see the recent paper by Marx and Pilipczuk [54] and the references therein.

Another related problem is Induced Subgraph Isomorphism which asks whether the base graph has an induced subgraph isomorphic to the pattern graph. Damaschke [28] showed that

Induced Subgraph Isomorphism on cographs is NP-complete. He also showed that Induced Subgraph Isomorphism is NP-complete for the disjoint unions of paths, and thus for proper interval graphs and bipartite permutation graphs. Marx and Schlotter [55] showed that Induced Subgraph Isomorphism on interval graphs is W[1]-hard when parameterized by the number of vertices in the pattern graph, but fixed-parameter tractable when parameterized by the numbers of vertices to be removed from the base graph. Heggernes et al. [44] showed that Induced Subgraph Isomorphism on proper interval graphs is NP-complete even if the base graph is connected. Heggernes et al. [45] have recently shown that Induced Subgraph Isomorphism on proper interval graphs and bipartite permutation graphs can be solved in polynomial time if the pattern graph is connected. Belmonte et al. [18] showed that Induced Subgraph Isomorphism on connected trivially perfect graphs is NP-complete. This result strengthens known results since every trivially perfect graph is an interval cograph. They also showed that the problem can be solved in polynomial time if the base graphs are trivially perfect graphs and the pattern graphs are threshold graphs.

6.2 Preliminaries

All graphs in this paper are finite, undirected, and simple. Let $G[U]$ denote the subgraph of $G = (V, E)$ induced by $U \subseteq V$. For a vertex $v \in V$, we denote by $G - v$ the graph obtained by removing v from G; that is, $G - v = G[V \setminus \{v\}]$. The *neighborhood* of a vertex v is the set $N(v) = \{u \in V \mid \{u, v\} \in E\}$. A vertex $v \in V$ is *universal* in G if $N(v) = V \setminus \{v\}$. A vertex $v \in V$ is *isolated* in G if $N(v) = \emptyset$. A set $I \subseteq V$ in $G = (V, E)$ is an *independent set* if for all $u, v \in I$, $(u, v) \notin E$. A set $S \subseteq V$ in $G = (V, E)$ is a *clique* if for all $u, v \in S$, $(u, v) \in E$. A pair (X, Y) of sets of vertices of a bipartite graph $H = (U, V; E)$ is a *biclique* if for all $x \in X$ and $y \in Y$, $(x, y) \in E$. A *component* of a graph G is an inclusion maximal connected subgraph of G. A component is *non-trivial* if it contains at least two vertices. The *complement* of a graph $G = (V, E)$ is the graph $\bar{G} = (V, \bar{E})$ such that $\{u, v\} \in \bar{E}$ if and only if $\{u, v\} \notin E$. The *disjoint union* of two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ is the graph $(V_G \cup V_H, E_G \cup E_H)$, where $V_G \cap V_H = \emptyset$. For a map $\eta\colon V \to V'$ and $S \subseteq V$, let $\eta(S)$ denote the set $\{\eta(s) \mid s \in S\}$.

6.2.1 Definitions of the problems

A graph $H = (V_H, E_H)$ is *subgraph-isomorphic* to a graph $G = (V_G, E_G)$ if there exists an injective map η from V_H to V_G such that $\{\eta(u), \eta(v)\} \in E_G$ holds for each $\{u, v\} \in E_H$. We call such a map η a *subgraph-isomorphism* from H to G. Graphs G and H are called the *base graph* and the *pattern graph*, respectively. The problems Subgraph Isomorphism and Spanning Subgraph Isomorphism are defined as follows:

Problem 28 Subgraph Isomorphism
Instance: A pair of graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$.
Question: Is H subgraph-isomorphic to G?

Problem 29 Spanning Subgraph Isomorphism
Instance: A pair of connected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, where $|V_G| = |V_H|$.
Question: Is H subgraph-isomorphic to G?

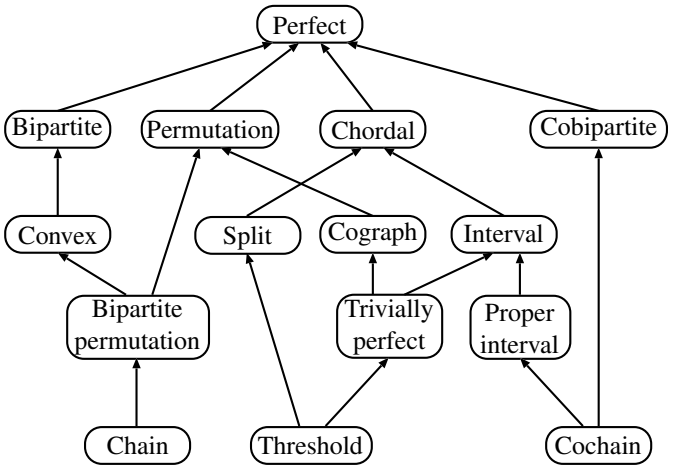

Figure 6.1: Graph classes.

6.2.2 Graph classes

Here we introduce the graph classes we deal with in this paper. For their inclusion relations, see the standard textbooks in this field [21, 38, 68]. See Figure 6.1 for the class hierarchy.

A bipartite graph $B = (X, Y; E)$ is a *chain* graph if the vertices of X can be ordered as $x_1, x_2, \ldots, x_{|X|}$ such that $N(x_1) \subseteq N(x_2) \subseteq \cdots \subseteq N(x_{|X|})$. A graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ is a *permutation graph* if there is a permutation π over V such that $\{i, j\} \in E$ if and only if $(i - j)(\pi(i) - \pi(j)) < 0$. A *bipartite permutation graph* is a permutation graph that is bipartite. A bipartite graph $H = (X, Y; E)$ is a *convex* graph if one of X and Y can be ordered such that the neighborhood of each vertex in the other side is consecutive in the ordering. It is known that a chain graph is a bipartite permutation graph, and that a bipartite permutation graph is a convex graph.

A graph is a *co-chain* graph if it is the complement of a chain graph. An *interval graph* is the intersection graph of a family of closed intervals of the real line. A *proper interval graph* is the intersection graph of a family of closed intervals of the real line where no interval is properly contained in another. A graph is *co-bipartite* if its complement is bipartite. In other words, co-bipartite graphs are exactly the graphs whose vertex sets can be partitioned into two cliques. From the definition, every co-chain graph is co-bipartite. It is known that every co-chain graph is a proper interval graph.

A graph is a *threshold* graph if there is a positive integer T (the *threshold*) and for every vertex v there is a positive integer $w(v)$ such that $\{u, v\}$ is an edge if and only if $w(u) + w(v) \geq T$. A graph is *trivially perfect* if the size of the maximum independent set is equal to the number of maximal cliques for every induced subgraph. It is known that a threshold graph is a trivially perfect graph, and that a trivially perfect graph is an interval graph.

A *split graph* is a graph whose vertex set can be partitioned into a clique and an independent set. A graph is chordal if every induced cycle is of length 3. Clearly, every threshold graph is a split graph, and every split graph is a chordal graph. It is known that every interval graph is a chordal graph. It is easy to see that any split graph (and thus any threshold graph) has at most one non-trivial component.

A graph is *perfect* if for any induced subgraph the chromatic number is equal to the size of a maximum clique. Graphs in all classes introduced in this section are known to be perfect.

6.3 Polynomial-Time Algorithms

In this section, we denote the number of the vertices and the edges in a base graph by n and m, respectively. For the input graphs G and H, we assume that $|V_G| \geq |V_H|$ and $|E_G| \geq |E_H|$, which can be checked in time $O(m + n)$.

6.3.1 Finding co-chain subgraphs in chordal graphs

It is known that co-chain graphs are precisely $\{I_3, C_4, C_5\}$-free graphs [43]; that is, graphs having no vertex subset that induces I_3, C_4, or C_5, where I_3 is the empty graph with three vertices and C_k is the cycle of k vertices. Using this characterization, we can show the following simple lemma.

Lemma 30 *A graph is a co-chain graph if and only if it is a co-bipartite chordal graph.*

Proof. To prove the if-part, let G be a co-bipartite chordal graph. Since G is co-bipartite, it cannot have I_3 as its induced subgraph. Since G is chordal, it does not have C_4 or C_5 as its induced subgraph. Therefore, G is $\{I_3, C_4, C_5\}$-free.

To prove the only-if-part, let G be a co-chain graph, and thus it is a co-bipartite graph. Suppose that G has an induced cycle C of length $k \geq 4$. Then k cannot be 4 or 5 since it does not have C_4 or C_5. If $k \geq 6$, then the first, third, and fifth vertices in the cycle form I_3. □

Now we can solve the problem as follows.

Theorem 31 Subgraph Isomorphism *is solvable in $O(mn^2 + n^3)$ time if the base graphs are chordal graphs and the pattern graphs are co-chain graphs.*

Proof. Let $G = (V_G, E_G)$ be the base chordal graph and $H = (V_H, E_H)$ be the pattern co-chain graph. We assume that G is not complete, since otherwise the problem is trivial.
Algorithm: We enumerate all the maximal cliques $C_1, \ldots, C_k$ of G. For each pair (C_i, C_j), we check whether H is subgraph-isomorphic to $G[C_i \cup C_j]$. If H is subgraph-isomorphic to $G[C_i \cup C_j]$ for some i and j, then we output "yes." Otherwise, we output "no."
Correctness: It suffices to show that H is subgraph-isomorphic to G if and only if there are two maximal cliques C_i and C_j of G such that H is subgraph-isomorphic to $G[C_i \cup C_j]$. The if-part is obviously true. To prove the only-if-part, assume that there is a subgraph-isomorphism η from H to G. Observe that for any clique C of H, there is a maximal clique C' of G such that $\eta(C) \subseteq C'$. Thus, since H is co-bipartite, there are two maximal cliques C_i and C_j such that $\eta(V_H) \subseteq C_i \cup C_j$. That is, H is subgraph-isomorphic to $G[C_i \cup C_j]$.
Running time: It is known that a chordal graph of n vertices with m edges has at most n maximal cliques, and all the maximal cliques can be found in $O(m + n)$ time [20, 42]. Since $G[C_i \cup C_j]$ is a co-chain graph by Lemma 30, testing whether H is subgraph-isomorphic to $G[C_i \cup C_j]$ can be done in $O(m + n)$ time [48]. Since the number of pairs of maximal cliques is $O(n^2)$, the total running time is $O(mn^2 + n^3)$. □

6.3.2 Finding threshold subgraphs in trivially perfect graphs

Here we present a linear-time algorithm for finding a threshold subgraph in a trivially perfect graph. To this end, we need the following lemmas.

Lemma 32 *If a graph G has a universal vertex u_G, and a graph H has a universal vertex u_H, then H is subgraph-isomorphic to G if and only if $H - u_H$ is subgraph-isomorphic to $G - u_G$.*

Proof. To prove the if-part, let η' be a subgraph-isomorphism from $H - u_H$ to $G - u_G$. Now we define $\eta\colon V_H \to V_G$ as follows:

$$\eta(w) = \begin{cases} u_G & \text{if } w = u_H, \\ \eta'(w) & \text{otherwise.} \end{cases}$$

Let $\{x, y\} \in E_H$. If $u_H \notin \{x, y\}$, then $\{\eta(x), \eta(y)\} = \{\eta'(x), \eta'(y)\} \in E_G$. Otherwise, we may assume that $x = u_H$ without loss of generality. Since u_G is universal in G, it follows that $\{\eta(x), \eta(y)\} = \{\eta(u_H), \eta(y)\} = \{u_G, \eta'(y)\} \in E_G$.

To prove the only-if-part, assume that η' is a subgraph-isomorphism from H to G. If there is no vertex $v \in V_H$ such that $\eta'(v) = u_G$, then we are done. Assume that $\eta'(v) = u_G$ for some vertex $v \in V_H$. Now we define $\eta\colon V_H \setminus \{u_H\} \to V_G \setminus \{u_G\}$ as follows:

$$\eta(w) = \begin{cases} \eta'(u_H) & \text{if } w = v, \\ \eta'(w) & \text{otherwise.} \end{cases}$$

Let $\{x, y\} \in E_H$. If $v \notin \{x, y\}$, then $\{\eta(x), \eta(y)\} = \{\eta'(x), \eta'(y)\} \in E_G$. Otherwise, we may assume without loss of generality that $v = x$. Since u_H is universal in H, it follows that $\{\eta(x), \eta(y)\} = \{\eta'(u_H), \eta'(y)\} \in E_G$. □

A component of a graph is *maximum* if it contains the maximum number of vertices among all the components of the graph. If a split graph has a non-trivial component, then the component is the unique maximum component of the graph.

Lemma 33 *A split graph H with a maximum component C_H is subgraph-isomorphic to a graph G if and only if $|V_H| \leq |V_G|$ and there is a component C_G of G such that C_H is subgraph-isomorphic to C_G.*

Proof. First we prove the only-if-part. Let η be a subgraph-isomorphism from H to G. We need $|V_H| \leq |V_G|$ to have an injective map from V_H to V_G. Since C_H is connected, $G[\eta(V(C_H))]$ must be connected. Thus there is a component C_G such that $\eta(V(C_H)) \subseteq V(C_G)$. Then $\eta|_{V(C_H)}$, the map η restricted to $V(C_H)$, is a subgraph isomorphism from C_H to C_G.

To prove the if-part, let η' be a subgraph-isomorphism from C_H to C_G. Let $R_H = V_H \setminus V(C_H) = \{u_1, \ldots, u_r\}$, and let $R_G = V_G \setminus \eta'(V(C_H)) = \{w_1, \ldots, w_s\}$. Since $|V_H| \leq |V_G|$ and $|V(C_H)| = |\eta'(V(C_H))|$, it holds that $r \leq s$. Now we define $\eta\colon V_H \to V_G$ as follows:

$$\eta(v) = \begin{cases} w_i & \text{if } v = u_i \in R_H, \\ \eta'(v) & \text{otherwise.} \end{cases}$$

Since H is a split graph, any component of H other than C_H cannot have two or more vertices. Thus the vertices in R_H are isolated in H. Therefore, the map η is a subgraph-isomorphism from H to G. □

The two lemmas above already allows us to have a polynomial-time algorithm. However, to achieve a linear running time, we need the following characterization of trivially perfect graphs.

A *rooted tree* is a directed tree with a unique in-degree 0 vertex, called the *root*. Intuitively, every edge is directed from the root to leaves in a directed tree. A *rooted forest* is the disjoint union of rooted trees. The *comparability graph* of a rooted forest is the graph that has the same vertex set as the rooted forest, and two vertices are adjacent in the graph if and only if one of the two is a descendant of the other in the forest. Yan et al. [72] showed that a graph is a trivially perfect graph if and only if it is the comparability graph of a rooted forest, and that

such a rooted forest can be computed in linear time. We call such a rooted forest a *generating forest* of the trivially perfect graph. If a generating forest is actually a rooted tree, then we call it a *generating tree*.

Theorem 34 Subgraph Isomorphism *is solvable in $O(m+n)$ time if the base graphs are trivially perfect graphs and the pattern graphs are threshold graphs.*

Proof. Let $G = (V_G, E_G)$ be the base trivially perfect graph and $H = (V_H, E_H)$ be the pattern threshold graph.
Algorithm: The pseudocode of our algorithm can be found in Algorithm 8. We use the procedure SGI which takes a trivially perfect graph as the base graph and a threshold graph as the pattern graph, and conditionally answers whether the pattern graph is subgraph-isomorphic to the base graph. The procedure SGI requires that

- both the graphs are connected, and
- the base graph has at least as many vertices as the pattern graph.

To use this procedure, we first attach a universal vertex to both G and H. This guarantees that both graphs are connected. We call the new graphs G' and H', respectively. By Lemma 32, (G', H') is a yes-instance if and only if so is (G, H). After checking that $|V_{G'}| \geq |V_{H'}|$, we use the procedure SGI.

In SGI(G, H), let u_G and u_H be universal vertices of G and H, respectively. There are such vertices since G and H are connected trivially perfect graphs [71]. Let C_H be a maximum component of $H - u_H$. For each connected component C_G of $G - u_G$, we check whether C_H is subgraph-isomorphic to C_G, by recursively calling the procedure SGI itself. If at least one of the recursive calls returns "yes," then we return "yes." Otherwise we return "no."
Correctness: It suffices to prove the correctness of the procedure SGI. If $|V_H| = 1$, then H is subgraph-isomorphic to G since $|V_G| \geq |V_H|$ in SGI. By Lemmas 32 and 33, H is subgraph-isomorphic to G if and only if there is a component C_G of $G - u_G$ such that C_H is subgraph-isomorphic to C_G. (Recall that any threshold graph is a split graph.) The procedure just checks these conditions. Also, when SGI recursively calls itself, the parameters C_G and C_H satisfy its requirements; that is, C_G and C_H are connected, and $|V(C_G)| \geq |V(C_G)|$.
Running time: For each call of SGI(G, H), we need the following:

- universal vertices u_G and u_H of G and H, respectively,
- a maximum component C_H of $H - u_H$,
- the components C_G of $G - u_G$, and
- the numbers of the vertices of C_G and $H - u_H$.

We show that they can be computed efficiently by using generating forests. Basically we apply the algorithm to generating forests instead of graphs.

Before the very first call of SGI(G, H), we compute generating trees of G and H in linear time. Additionally, for each node in the generating trees, we store the number of its descendants. This can be done also in linear time in a bottom-up fashion.

At some call of SGI(G, H), assume that we have generating trees of G and H. It is easy to see that the root of the generating trees are universal vertices. Hence we can compute u_G and u_H in constant time. By removing these root nodes from the generating trees, we obtain generating forests of $G - u_G$ and $H - u_H$. Each component of the generating forests corresponds

Algorithm 8: Finding a threshold subgraph H in a trivially perfect graph G.

```
G' := G with a universal vertex
H' := H with a universal vertex
if |V_{G'}| ≥ |V_{H'}| then
    return SGI(G', H')
  else return no
end

Require G and H are connected, and |V_G| ≥ |V_H|
Procedure SGI(G, H)
if |V_H| = 1 then
    return yes
end
u_G := a universal vertex of G
u_H := a universal vertex of H
C_H := a maximum component of H − u_H
forall components C_G of G − u_G do
    if |V(C_G)| ≥ |V(H − u_H)| then
        if SGI(C_G, C_H) = yes then
            return yes
        end
    end
end
return no
```

to a component of the corresponding graphs. Thus we can compute the components of $G - u_G$ and a maximum component of $H - u_H$, with their generating trees, in time proportional to the number of the children of u_G and u_H. The numbers of the vertices of C_G and $H - u_H$ can be computed easily in constant time, because we know the number of the descendants of each node in generating trees.

The recursive calls of SGI take only $O(n)$ time in total since it is proportional to the number of edges in the generating trees. Therefore, the total running time is $O(m + n)$. □

6.4 NP-completeness

It is known that for perfect graphs, Clique can be solved in polynomial time [39]. Since co-chain graphs and threshold graphs are very close to complete graphs, one may ask whether the problem of finding co-chain graphs or threshold graphs can be solved in polynomial time for perfect graphs. In this section, we show that this is not the case. More precisely, we show that even the specialized problem Spanning Subgraph Isomorphism is NP-complete for the case where the base graphs are somewhat restricted perfect graphs and the pattern graphs are co-chain or threshold graphs.

It is known that Maximum Edge Biclique, the problem of finding a biclique with the maximum number of edges, is NP-complete for bipartite graphs [63]. This implies that Subgraph Isomorphism is NP-complete if the base graphs are connected bipartite graphs and the pattern graphs are connected chain graphs, because complete bipartite graphs are chain graphs. We

sharpen this hardness result by showing that the problem is still NP-complete if we further restrict the pattern chain graphs to have the same number of vertices as the base graph. That is, we show that Spanning Subgraph Isomorphism is NP-complete when the base graphs are bipartite graphs and the pattern graphs are chain graphs.

Since the problem Spanning Subgraph Isomorphism is clearly in NP for any graph class, we only show its NP-hardness here. All the results in this section are based on the following theorem and lemma taken from Kijima et al. [48].

Theorem 35 (Kijima et al. [48]) Spanning Subgraph Isomorphism *is NP-complete if*

1. *the base graphs are chain graphs and the pattern graphs are convex graphs,*
2. *the base graphs are co-chain graphs and the pattern graphs are co-bipartite graphs, or*
3. *the base graphs are threshold graphs and the pattern graphs are split graphs.*

Lemma 36 (Kijima et al. [48]) *If $|V_H| = |V_G|$, then H is subgraph-isomorphic to G if and only if $\bar{G}$ is subgraph-isomorphic to $\bar{H}$.*

For a graph class $\mathcal{C}$, let co-$\mathcal{C}$ denote the graph class $\{\bar{G} \mid G \in \mathcal{C}\}$. The next lemma basically shows that if $\mathcal{C}$ satisfies some property, then the hardness of Spanning Subgraph Isomorphism for $\mathcal{C}$ implies the hardness for co-$\mathcal{C}$.

Lemma 37 *Let $\mathcal{C}$ and $\mathcal{D}$ be graph classes such that* co-$\mathcal{C}$ *and* co-$\mathcal{D}$ *are closed under universal vertex additions. If* Spanning Subgraph Isomorphism *is NP-complete when the base graphs belong to $\mathcal{C}$ and the pattern graphs belong to $\mathcal{D}$, then the problem is NP-complete also when the base graphs belong to* co-$\mathcal{D}$ *and the pattern graphs belong to* co-$\mathcal{C}$.

Proof. Given two connected graphs $G \in \mathcal{C}$ and $H \in \mathcal{D}$ with $|V_G| = |V_H|$, it is NP-complete to decide whether H is subgraph-isomorphic to G. By Lemma 36, H is subgraph-isomorphic to G if and only if $\bar{G}$ is subgraph-isomorphic to $\bar{H}$. By Lemma 32, $\bar{G}$ is subgraph-isomorphic to $\bar{H}$ if and only if $\bar{G}'$ is subgraph-isomorphic to $\bar{H}'$, where $\bar{G}'$ and $\bar{H}'$ are obtained from $\bar{G}$ and $\bar{H}$, respectively, by adding a universal vertex. Therefore, H is subgraph-isomorphic to G if and only if $\bar{G}'$ is subgraph-isomorphic to $\bar{H}'$. Clearly, $\bar{G}' \in$ co-$\mathcal{C}$ and $\bar{H}' \in$ co-$\mathcal{D}$, they are connected, and they have the same number of vertices. Thus the lemma holds. □

A graph is a *co-convex* graph if its complement is a convex graph. Clearly co-convex graphs are closed under additions of universal vertices.

Corollary 38 Spanning Subgraph Isomorphism *is NP-complete if*

1. *the base graphs are co-convex graphs and the pattern graphs are co-chain graphs,*
2. *the base graphs are bipartite graphs and the pattern graphs are chain graphs, or*
3. *the base graphs are split graphs and the pattern graphs are threshold graphs.*

Proof. The NP-completeness of the case (1) is a corollary to Theorem 35 (1) and Lemma 37. To prove (3), we need Theorem 35 (3), Lemma 37, and the well-known facts that threshold graphs and split graphs are self-complementary [38]. That is, the complement of a threshold graph is a threshold graph, and the complement of a split graph is a split graph.

For (2), we cannot directly apply the combination of Theorem 35 (2) and Lemma 37 since bipartite graphs and chain graphs are not closed under universal vertex additions. Fortunately, we can easily modify the proof of Theorem 35 (2) in Kijima et al. [48] so that the complements of the base graphs and the pattern graphs are also connected. Then, Lemma 36 implies the statement. Since it will be a repeat of a known proof with a tiny difference, we omit the detail. □

6.5 Conclusion of Chapter 6

We have studied (Spanning) Subgraph Isomorphism for classes of perfect graphs, and have shown sharp contrasts of its computational complexity. An interesting problem left unsettled is the complexity of Subgraph Isomorphism where the base graphs are bipartite permutation graphs and the pattern graphs are chain graphs. It is known that although the maximum edge biclique problem is NP-complete for general bipartite graphs [63], it can be solved in polynomial time for some super classes of bipartite permutation graphs (see [61]). Therefore, it might be possible to have a polynomial-time algorithm for Subgraph Isomorphism when the pattern graphs are chain graphs and the base graphs belong to an even larger class like convex graphs.

Bibliography

[1] P. K. Agarwal and M. Sharir. Applications of a new space-partitioning technique. *Discrete Computonal Geometry*, 9:11–38, 1993.

[2] A. Aggarwal and R. Anderson. A random nc algorithm for depth-first search. *Combinatorica*, 8(1):1–12, 1988.

[3] Borodin Allan, J. Fishcher Michael, G. Kirkpatrick David, A. Lynch Nancy, and Tompa Martin. A time-space tradeoff for sorting on non-oblivious machines. *Journal of Computer and System Sciences*, 22(3):351 – 364, 1981.

[4] R. Anderson and E. Mayr. Parallelism and the maximal path problem. *Information Processing Letters*, 24(2):121–126, 1987.

[5] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[6] T. Asano. Constant-work-space algorithms: how fast can we solve problems without using any extra array? In *Proceedings of the19th Annual International on Symposium Algorithms Computation (ISAAC), invited talk*, volume 5369 of Lecture Notes in Computer Science, page 1. Springer-Verlag, 2008.

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

[8] T. Asano, A. Elmasry, and J. Katajainen. Priority queues and sorting for read-only data. In *Proceedings of the 10th Annual Conference on Theory and Applications of Models of Computation, Lecture Notes in Computer Science*, volume 7876, pages 32–41. Springer-Verlag, 2013.

[9] T. Asano and D. Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *Proceedings of Algorithms and Data Structures, 13th International Symposium, Lecture Notes in Computer Science*, volume 8037, pages 61–72. Springer-Verlag, 2013.

[10] T. Asano, D. Kirkpatrick, K. Nakagawa, and O. Watanabe. *Mathematical Foundations of Computer Science 2014: 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, chapter $\tilde{O}(\sqrt{n})$-Space and Polynomial-time Algorithm for the Planar Directed Graph Reachability, pages 45–56. Springer Berlin Heidelberg, 2014.

[11] T. Asano and M. Konagaya. Zero-space data structure for farthest-point voronoi diagram. In *Abstract of the 4th Annual Meeting of Asian Association for Algorithms and Computation*, page 14, 2011.

[12] T. Asano and R. Kumar. A small-space algorithm for removing small connected components from a binary image. *IEICE Transactions*, 96(6):1044–1050, 2016.

[13] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-working-space algorithms for geometric problems. *Journal of computational geometry*, 2(1):46–68, 2011.

[14] T. Asano, W. Mulzer, and Y. Wang. Constant-work-space algorithm for a shortest path in a simple polygon. In *Proceeding of the 4th Workshop on Algorithms and Computation (WALCOM)*, pages 9–20, 2010.

[15] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the 11th ACM Symposium on Computational Geometry*, pages 211–219, 1995.

[16] G. Barnes, J. Buss, W. Ruzzo, and B. Schieber. A sublinear space, polynomial time algorithm for directed *s-t* connectivity. *SIAM Journal of Computing*, 27(5):1273–1282, 1998.

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

[18] Rémy Belmonte, Pinar Heggernes, and Pim van 't Hof. Edge contractions in subclasses of chordal graphs. *Discrete Appl. Math.*, 160:999–1010, 2012.

[19] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transaction on Computers*, C-28(9):643–647, 1979.

[20] Jean R. S. Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *The IMA Volumes in Mathematics and its Applications*, pages 1–29. Springer Verlag, 1993.

[21] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999.

[22] M. Timothy Chan and Y. Eric Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2006.

[23] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[24] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. In *Proceedings of the 6th ACM Symposium on Computational Geometry*, pages 23–33, 1990.

[25] B. M. Chazelle. Filetering search: a new approach to query-answering. *SIAM Journal on Computing*, 15:703–724, 1986.

[26] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *Proceedings of the 15th Canadian Conference on Computational Geometry*, pages 68–71, 2003.

[27] Charles J. Colbourn. On testing isomorphism of permutation graphs. *Networks*, 11:13–21, 1981.

[28] Peter Damaschke. Induced subgraph isomorphism for cographs is NP-complete. In *WG '90*, volume 487 of *Lecture Notes in Comput. Sci.*, pages 72–78, 1991.

[29] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, third edition, 2008.

[30] P. de la Tore and C. Kruskal. Fast parallel algorithms for lexicographic search and path-algebra problems. *Journal of Algorithms*, 19:1–24, 1995.

[31] P. de la Tore and C. Kruskal. Polynomially improved efficiency for fast parallel single-source lexicographic depth-first search, breadth-first search. *Theory of Computing Systems*, 34:275–298, 2001.

[32] Frederic Dorn. Planar subgraph isomorphism revisited. In *STACS 2010*, volume 5 of *LIPIcs*, pages 263–274, 2010.

[33] H. Ebara, N. Fukuyama, H. Nakano, and Y. Nakanishi. Roundness algorithms using the voronoi diagrams. In *Abstract of the First Canadian Conference on Computational Geometry*, page 41, 1989.

[34] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)*, pages 143–152, 2010.

[35] M. Elberfeld and K. Kawarabayashi. Embedding and canonizing graphs of bounded genus in logspace. In *Proceedings of the 46th Annual ACM Symposium on the Theory of Computing (STOC 2014)*, pages 383–392, 2014.

[36] David Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms Appl.*, 3:1–27, 1999.

[37] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[38] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. North Holland, second edition, 2004.

[39] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[40] Arvind Gupta and Naomi Nishimura. The complexity of subgraph isomorphism for classes of partial k-trees. *Theoret. Comput. Sci.*, 164:287–298, 1996.

[41] S. Har-Peled. A small-space algorithm for removing small connected components from a binary image. *Journal of computational geometry*, 7(2):19–45, 2016f.

[42] Pinar Heggernes. Treewidth, partial k-trees, and chordal graphs. Partial curriculum in INF334 - Advanced algorithmical techniques, Department of Informatics, University of Bergen, Norway, 2005.

[43] Pinar Heggernes and Dieter Kratsch. Linear-time certifying recognition algorithms and forbidden induced subgraphs. *Nordic J. Comput.*, 14:87–108, 2007.

[44] Pinar Heggernes, Daniel Meister, and Yngve Villanger. Induced subgraph isomorphism on interval and proper interval graphs. In *ISAAC 2010*, volume 6507 of *Lecture Notes in Comput. Sci.*, pages 399–409, 2010.

[45] Pinar Heggernes, Pim van 't Hof, Daniel Meister, and Yngve Villanger. Induced subgraph isomorphism on proper interval and bipartite permutation graphs. Submitted manuscript.

[46] T. Imai. Polynomial-time memory constrained shortest path algorithms for directed graphs (in japanese). In *Proceedings of the 12th Forum on Information Technology*, volume 1, pages 9–16, 2013.

[47] T. Imai, K. Nakagawa, A. Pavan, N. Vinodchandran, and O. Watanabe. An $o(n^{1/2+\epsilon})$-space and polynomial-time algorithm for directed planar reachability. In *Proceedings of 2013 IEEE Conference on Computational Complexity*, pages 277–286, 2013.

[48] Shuji Kijima, Yota Otachi, Toshiki Saitoh, and Takeaki Uno. Subgraph isomorphism in graph classes. *Discrete Math.*, 312:3164–3173, 2012.

[49] M. Konagaya and T. Asano. Reporting all segment intersections using an arbitrary sized work space. *IEICE Transactions*, 96-A(6):1066–1071, 2013.

[50] Andrzej Lingas. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. *Theoret. Comput. Sci.*, 63:295–302, 1989.

[51] George S Lueker and Kellogg S. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26:183–195, 1979.

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

[53] Harry G. Mairson and Jorge Stolfi. *Theoretical Foundations of Computer Graphics and CAD*, volume 40, chapter Reporting and Counting Intersections Between Two Sets of Line Segments, pages 307–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[54] Dániel Marx and Michał Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). *CoRR*, abs/1307.2187, 2013.

[55] Dániel Marx and Ildikó Schlotter. Cleaning interval graphs. *Algorithmica*, 65:275–316, 2013.

[56] Jiří Matoušek and Robin Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Math.*, 108:343–364, 1992.

[57] David W. Matula. Subtree isomorphism in $O(n^{5/2})$. In B. Alspach, P. Hell, and D.J. Miller, editors, *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 91–106. Elsevier, 1978.

[58] J. I. Munro. An implicit data structure supporting insertion, deletion and search in $o(\log^2 n)$ time. *Journal of Computer and System Science*, 33(1):66–74, 1986.

[59] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315 – 323, 1980.

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

[61] Doron Nussbaum, Shuye Pu, Jörg-Rüdiger Sack, Takeaki Uno, and Hamid Zarrabi-Zadeh. Finding maximum edge bicliques in convex bipartite graphs. *Algorithmica*, 64(2):311–325, 2012.

[62] C. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[63] René Peeters. The maximum edge biclique problem is NP-complete. *Discrete Appl. Math.*, 131:651–654, 2003.

[64] F. P. Preparata and M. I. Shamos. *Computational geometry. An introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.

[65] J. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[66] O. Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4):17:1–17:24, 2008.

[67] M. I. Shamos and D. J. Hoey. Geometric intersection problems. In *Proceedings of the 17th IEEE Symposium on Foundation of Computer Science*, pages 208–215, 1976.

[68] Jeremy P. Spinrad. *Efficient Graph Representations*, volume 19 of *Fields Institute monographs*. American Mathematical Society, 2003.

[69] Maciej M. Sysło. The subgraph isomorphism problem for outerplanar graphs. *Theoret. Comput. Sci.*, 17:91–97, 1982.

[70] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[71] E. S. Wolk. A note on "The comparability graph of a tree". *Proc. Amer. Math. Soc.*, 16:17–20, 1965.

[72] Jing-Ho Yan, Jer-Jeong Chen, and Gerard J. Chang. Quasi-threshold graphs. *Discrete Appl. Math.*, 69(3):247–255, 1996.

Publications

Journal paper

[1] M. Konagaya and T. Asano, “Reporting All Segment Intersections Using an Arbitrary Sized Work Space”, IEICE Transactions, Vol.E96-A, NO.6, pp.1066–1071, 2013.

[2] M. Konagaya, Y. Otachi and R. Uehara, “Polynomial-Time Algorithms for Subgraph Isomorphism in Small Graph Classes of Perfect Graphs”, Discrete Applied Mathematics, Vol.199, pp.37–45, 2016.

International conference

[3] M. Konagaya and T. Asano, “Zero-space Data Structure for Farthest-point Voronoi Diagram”, The 4th Annual Meeting of the Asian Association for Algorithms and Computation, p.44, 2011.

[4] M. Konagaya and T. Asano, “Algorithm for reporting all segment intersections using work space of arbitrary size”, In Proc, The 15th Japan-Korea Joint Workshop on Algorithms and Computation, pp.124–131, 2012.

[5] M. Konagaya, Y. Otachi and R. Uehara, “Polynomial-Time Algorithms for Subgraph Isomorphism in Small Graph Classes of Perfect Graphs”, The 11th Annual conference on Theory and Applications of Models of Computation, Lecture Notes in Computer Science, Volume 8402, pp.216–228, Springer, 2014.

[6] T. Asano, T. Izumi, M. Kiyomi, M. Konagaya, H. Ono, Y. Otachi, P. Schweitzer, J. Tarui and R. Uehara, “Depth-First Search Using $O(n)$ bits”, The 25th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science, Volume 8889, pp.553–564, Springer, 2014.

Domestic conference and symposium

[7] 小長谷松雄, 浅野哲夫, “Constant-space Data Structure for Farthest-point Voronoi Diagram”, 第 10 回情報科学技術フォーラム, RA-004, 2011.

[8] 小長谷松雄, 浅野哲夫, “Constant-space Data Structure for Farthest-point Voronoi Diagram”, 夏の LA シンポジウム, pp.S3:1–S3:5, 2011.

[9] 小長谷松雄, 浅野哲夫, “Algorithm for Reporting All Segment Intersections Using Work space of Arbitrary Size”, IEICE Tech. Rep., COMP2012-7, Vol. 112, No. 21, pp.45–52, 2012.

[10] 小長谷松雄, 浅野哲夫, “単純多角形内部の最短経路発見のためのメモリ調節可能アルゴリズム”, IEICE Tech. Rep., COMP2013-48, Vol. 113, No. 371, pp.59–62, 2013.

[11] 大舘陽太, 上原隆平, 小長谷松雄, “Polynomial-Time Algorithms for Subgraph Isomorphism in Small Graph Classes of Perfect Graphs”, IPSJ SIG Technical Report, 2013-AL-147-12, pp. 1-6, 2014/03/03-04.