Efficient Algorithms for Path Problems in Weighted Graphs

Virginia Vassilevska

August 20, 2008
CMU-CS-08-147

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy Blelloch, Chair
Manuel Blum
Anupam Gupta
Uri Zwick (Tel Aviv University)

Submitted in partial fulfillment of the requirements for the degree of Doctor in Philosophy.

This research was sponsored by the National Science Foundation under contracts no. CCR-0122581, no. CCR-0313148, and no. IIS-0121641. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: shortest paths, matrix multiplication, bottleneck paths, earliest arrivals

To my parents.

Abstract

Problems related to computing optimal paths have been abundant in computer science since its emergence as a field. Yet for a large number of such problems we still do not know whether the state-of-the-art algorithms are the best possible. A notable example of this phenomenon is the all pairs shortest paths problem in a directed graph with real edge weights. The best algorithm (modulo small polylogarithmic improvements) for this problem runs in *cubic* time, a running time known since the 1960s (by Floyd and Warshall). Our grasp of many such fundamental algorithmic questions is far from optimal, and the major goal of this thesis is to bring some new insights into efficiently solving path problems in graphs.

We focus on several path problems optimizing different measures: shortest paths, maximum bottleneck paths, minimum nondecreasing paths, and various extensions. For the all-pairs versions of these path problems we use an algebraic approach. We obtain improved algorithms using reductions to fast matrix multiplication. For maximum bottleneck paths and minimum nondecreasing paths we are the first to break the cubic barrier, obtaining truly subcubic strongly polynomial algorithms. We also consider a nonalgebraic, combinatorial approach, which is considered more efficient in practice compared to methods based on fast matrix multiplication. We present a combinatorial data structure that maintains a matrix so that products with given sparse vectors can be computed efficiently. This allows us to obtain good running times for path problems in unweighted sparse graphs.

This thesis also gives algorithms for some single source path problems. We obtain the first *linear* time algorithm for the single source minimum nondecreasing paths problem. We give some extensions to this, including an algorithm to find cheapest minimum nondecreasing paths.

Besides finding optimal paths, we consider the related problem of finding optimal cycles. In particular, we focus on the problem of finding in a weighted graph a triangle of maximum weight sum. We obtain the first truly subcubic algorithm for finding a maximum weight triangle in a node-weighted graph. We also present algorithms for the edge-weighted case. These algorithms immediately imply good algorithms for finding maximum weight k-cliques, or arbitrary maximum weight pattern subgraphs of fixed size.

Acknowledgements

This thesis can be seen as a collective effort of a large group of people. This group includes my advisor and coauthors, my family and a large number of friends who stood by me during difficult times. I am grateful to all of you for your help and for believing in me.

Firstly, I am deeply grateful to my family: to my mother Tanya and my father Panayot who listened to me (no matter whether I was bragging or complaining, laughing or crying), loved me and supported me, and gave me an enormous amount of valuable advice half of which I followed and the other half of which I wish I followed. I am thankful to my brother Alexander, to my uncle Yordan and aunt Gergana for their support, and to my grandmother Genka who has always been there for me. Мили мамо, татко, Сашо, бабо, вуйчо и Гери, благодаря ви много за непрестанната ви подкрепа и обич!

Throughout my Ph.D studies I was skillfully guided by my advisor, Guy Blelloch. I am really grateful to him for being the perfect advisor for me, giving me wonderful advice and letting me have freedom to work on my own things whenever I needed to! I would also like to thank the rest of my thesis committee, Anupam Gupta, Manuel Blum and Uri Zwick, for agreeing to see this thesis through and for giving me valuable advice about various extensions of my work.

I have written many papers (including the majority of the ones in this thesis) with one person who also happens to be my partner in life, Ryan Williams. I am deeply grateful for his unconditional support and amazing ability to calm me down and cheer me up. Being around him is truly a blessing and this thesis may not have been finished if it had not been for him.

During the last five years, I have had the pleasure to work with several different people on various projects: Raphael Yuster, Maverick Woo, Umut Acar, Srinath Sridhar, Daniel Golovin, Michelle Goodstein. Thank you for the awesome ideas and creativity! The Carnegie Mellon computer science faculty has been an invaluable resource for me. In particular, I would like to mention Avrim Blum, Lenore Blum and Gary Miller who gave me a lot of useful advice. I am also grateful to Guy, Anupam, Manuel and Raphael for writing recommendations for me when I was applying for jobs.

I would like to thank all of the friends I have made here at Carnegie Mellon. Hopefully I am not missing anyone: Liz Crawford, Shobha Venkataraman, Sue Ann Hong, Joey Gonzalez, Katrina Ligett, Noam Zeilberger, Alice Brumley, Sonia Chernova and James Hays, Mike and Kelly Crawford, our champion tennis team – Anne Yust, David Abraham, Gene Hambrick, Sam Ganzfried, Maxim Bichuch, Dan Wendlandt, Pan Papasaikas, Vijay Vasudevan, ... Finally, I am really grateful to Mrs. Teena Williams, Jialan Wang and Nora Tu for their love and support. You all made my life wonderful!

Table of Contents

Chapter 1

Introduction

Problems related to computing optimal paths are abundant in computer science; they have been studied since the emergence of computer science as a field, and are at the heart of the vast majority of applications in the real world. Yet for many of these problems we still do not know how far our techniques can go, and whether the state-of-the-art algorithms are the best possible. A notable example of this phenomenon is the shortest paths problem. For instance, for finding a shortest path between two fixed nodes in a directed graph with nonnegative real weights on the edges, there might exist an algorithm with running time only linear in the size of the input graph. Yet, the best known algorithm for the problem in a general computational model (Dijkstra's) has a logarithmic multiplicative overhead. Furthermore, the best algorithm for finding shortest paths between all pairs of nodes in a directed graph with real edge weights runs (modulo small polylogarithmic improvements) in *cubic* time, a running time known since the 1960s (by Floyd and Warshall). Our grasp of many fundamental algorithmic questions is far from optimal, and the major goal of this thesis is to find some new insights into efficiently solving different path problems in graphs.

A path in a graph is a sequence of nodes, every consecutive two linked by an edge. A path problem in a graph has three variants:

1. *single source–single destination* (also called $s-t$): given a graph and two nodes s and t, find an optimal path from s to t,
2. *single source*: given a graph and node s, for every node t find an optimal path from s to t,
3. *all pairs*: given a graph, for every two nodes s and t find an optimal path from s to t.

There are many measures for path optimality, depending on the problem. In the simple *reachability* problem, any path is optimal, as long as it exists. In the *shortest paths* problem, one is given a graph with real weights on the edges and a path between two nodes is optimal if it has the *minimum weight sum* over all paths between the nodes. An obvious application of this problem is automatically finding driving directions between physical locations. Another often used measure is the minimum edge on a path – the so called *bottleneck*. Maximizing this measure defines the *maximum bottleneck* paths problem. One application of this is finding directions between two locations for which all tunnels have as high a clearance as possible. The problem of minimizing the largest weight edge, on the other hand, can be applied to find the least congested route in a graph representing traffic patterns. A third measure for optimal paths is the weight of the last edge on a path, the edge weights on which form a *nondecreasing* sequence. Minimizing this last

edge weight defines the *minimum nondecreasing* or *earliest arrival* paths problem. The primary application for this problem is finding an itinerary for a trip between two physical locations that gets you to your destination as early as possible. In theoretical computer science, these problems have been studied alongside since the 1950s. They are also very real and applicable today. Other applications of path problems include robot motion planning, highway and power line engineering, network connection routing, various scheduling problems, sequence alignment in molecular biology, length-limited Huffman coding, and many many others.

In the most general setting, a path problem on an edge-weighted graph G is characterized by a function that maps the set of edges of each path to a number, so that the path problem on two nodes s and t seeks to optimize its function over all paths from s to t in G. We formalize this further in Chapter 2. All of the problems we consider in this thesis are solvable in polynomial time. NP-hard path problems such as Hamiltonian path do not fall into our framework.

We consider a known algebraic approach to all-pairs path problems that links the path problem to computing a matrix product over a semiring. We extend the framework to allow slightly more general algebraic structures, and outline an approach to reducing a given algebraic structure matrix product to matrix multiplication over a ring. We then focus on some particular path problems and obtain exciting new algorithms for them using this algebraic framework. Using algorithms for fast matrix multiplication [28] we are the first to break the cubic barrier for maximum bottleneck paths and minimum nondecreasing paths, obtaining truly subcubic strongly polynomial algorithms. This resolves two 40–year old open problems.

We also consider a nonalgebraic, combinatorial approach, which is considered more efficient in practice compared to methods based on fast matrix multiplication. We present a combinatorial data structure that maintains a matrix so that products with given sparse vectors can be computed efficiently. This allows us to obtain good running times for many path problems in sparse graphs by improving the running times of existing algorithms using our data structure.

This thesis also gives algorithms for some single source path problems. We obtain the first *linear* time algorithm for the single source minimum nondecreasing paths problem. We give some extensions to this, including algorithms to find shortest minimum nondecreasing paths and cheapest minimum nondecreasing paths.

Besides finding optimal paths, we consider the related problem of finding optimal cycles of a given fixed size. In particular, we focus on the problem of finding a triangle of maximum weight sum in a weighted graph. We obtain the first truly subcubic algorithm for finding a maximum weight triangle in a node-weighted graph, resolving a 30–year old open problem. We also present algorithms for the edge-weighted case. These algorithms immediately imply good algorithms for finding maximum weight k-cliques, or arbitrary maximum weight pattern subgraphs of fixed size.

1.1 Organization of the Thesis

Chapter 2 introduces the algebraic framework for matrix products and path problems. Chapter 3 focuses on particular matrix products and presents efficient algorithms for them. Chapter 4 considers problems related to efficiently finding cycles (*e.g.* triangles) and fixed size k-cliques or other small subgraphs. Chapter 5 focuses on the maximum bottleneck paths problem and some extensions. Chapter 6 considers the minimum nondecreasing path problem and its variants. Chapter 7 presents our nonalgebraic, combinatorial algorithms based on our efficient data structure for matrix–sparse vector products. Chapter 8 concludes the thesis, presenting some open problems.

1.2 Preliminaries and Notation

All of the graphs we consider in this thesis are directed, unless stated otherwise. We use the terms *node* and *vertex* interchangeably. For directed graphs we use the terms *edge* and *arc* interchangeably. The variables m and n refer to the number of edges and nodes in a graph, respectively. Each graph, unless stated otherwise, is considered to be weakly connected, so that $m \geq n-1$. A *DAG* refers to a directed acyclic graph. A *clique* is a complete graph, and a clique on k nodes is often referred to as a K_k.

For a positive integer k, we let $[k] := \{1, \ldots, k\}$. We use M^T to denote the transpose of a matrix M. $\tilde{O}(\cdot)$ suppresses polylogarithmic factors, *i.e.* $\tilde{O}(f(n)) = O(f(n)\text{polylog}(n))$. $\mathbb{Z}$ and $\mathbb{R}$ refer to the set of integers and reals, respectively.

Models of computation. We use the addition-comparison and word RAM models in different parts of the thesis. We use the standard addition-comparison computational model, along with random access to registers in all of our algebraic algorithms for path problems. The $O(\log n)$-word RAM model is used partially when discussing combinatorial algorithms, though the majority of our combinatorial algorithms work on the pointer machine model as well. The w-word RAM model is only used in our linear time algorithm for minimum nondecreasing paths, discussed in Chapter 6.

Chapter 2

Matrix Multiplication and Path Problems

In all branches of mathematics one can find some form of matrix multiplication. The ordinary, algebraic matrix product over a ring (such as the real numbers) is the most prevalent type of matrix product. The algebraic product of two $n \times n$ matrices A and B is an $n \times n$ matrix C so that for every pair of indices $i, j \in [n]$, $C[i, j]$ is given by the dot product of row i of A and column j of B. Computing the product of two matrices efficiently is a central problem in computational linear algebra and scientific computing; problems such as solving linear systems, inverting a matrix or computing a matrix determinant are intimately related to computing matrix-matrix products. Moreover, matrix multiplication has surprising applications in many areas of computer science with no immediate relation to linear algebra. Valiant [93] showed that given a context free grammar and a length n string, deciding if the string can be generated by the grammar can be done in asymptotically the same time as $n \times n$ matrix multiplication. Itai and Rodeh [54] showed that one can use fast matrix multiplication to vastly improve the complexity of clique finding. Alon, Galil and Margalit [3] showed that matrix multiplication can be used to obtain a truly subcubic running time for all pairs shortest paths in directed graphs with weights in $\{-1, 0, 1\}$. Vaidya [92] showed that even linear programming can be sped-up by using a subcubic algorithm for matrix products.

Because of its plentiful applications, matrix multiplication is widely studied in computer science. Until 1969, it was believed that matrix multiplication requires a cubic number of steps. Strassen's discovery of his $O(n^{2.81})$ time algorithm [85] was an exciting development in the field of algorithms and spawned a long series of new improvements to the running time of matrix multiplication algorithms, culminating in the current best of $O(n^{2.376})$ by Coppersmith and Winograd [28]. Interestingly, there is no better lower bound known than the trivial $O(n^2)$, the time that it takes to write down the output matrix.

Besides the algebraic matrix product of two matrices one could consider matrix products using different operations instead of integer sums and products. One such example is the so called distance product which instead of $+$ uses a min operation on integers and instead of $\cdot$ uses $+$ on integers. This chapter introduces an algebraic framework for matrix products using different operations. In particular, we focus on matrix products that correspond to all pairs path problems. Such frameworks have been proposed before, *e.g.* the semiring framework, which we describe late in this chapter. However, the known frameworks do not encompass all path problems in this thesis. In particular, they cannot capture the product corresponding to the all pairs nondecreasing paths

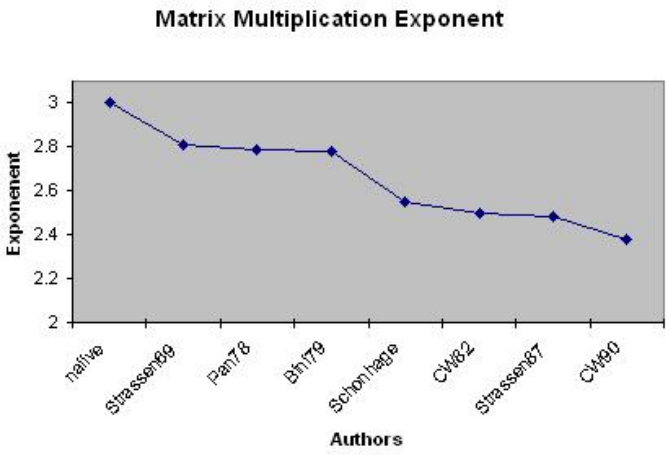

Figure 2.1: Improving the bound on ω.

problem. Our framework is slightly more general than the semiring framework and unifies the path problems considered in this thesis.

We begin with an overview of the algorithmic results on algebraic matrix products which we refer to simply by matrix multiplication. This nice progress on matrix multiplication algorithms allows us to obtain better algorithms for the rest of the products in this thesis, and it gives a motivation that other matrix products can have good algorithms.

2.1 A Brief History of Matrix Multiplication Algorithms

The study of the algorithmic complexity of (algebraic) matrix multiplication is really about determining the optimal value of the exponent ω for which $n \times n$ matrix multiplication is in $\tilde{O}(n^\omega)$ time. Clearly, $\omega \geq 2$ and Strassen showed that $\omega < 2.81$. The first progress on improving the upper bound on ω after Strassen's algorithm was obtained by Pan in 1978 who showed that $\omega < 2.79$ [71]. Bini et al. [9] decreased this bound slightly to 2.78. In 1981, Schönhage [80] made another breakthrough proving the so called asymptotic sum inequality for tensors which lead to the bound $\omega < 2.55$. Further research [72, 78, 27] showed $\omega < 2.50$. The laser method of Strassen [86] was the next big development. Strassen used it to improve the bound on ω to 2.48, and in 1990 Coppersmith and Winograd used it to achieve the current best known bound of $\omega < 2.376$. In 2003 Cohn and Umans [25] developed a new group-theoretic framework for designing algorithms for matrix multiplication. This framework led to some interesting new algorithms by Cohn et al. [24], the best of which has the same complexity as Coppersmith and Winograd's algorithm. The authors made some group-theoretical conjectures which would imply that $\omega = 2$. A more detailed survey on the history of matrix multiplication algorithms can be found in the book by Bürgisser et al. [13], or in Pan's survey [70]. Figure 2.1 shows how the bound on ω has changed between 1969 and now.

2.2 Generalized Matrix Product

In many applications one needs matrix products which use different operations instead of $+$ and $\cdot$. One example is the so called Boolean matrix product in which the matrices A and B have entries in $\{0,1\}$ and the output matrix C is given by $C[i,j] = \bigvee_k (A[i,k] \wedge B[k,j])$, where $\vee$ and $\wedge$ are the Boolean OR and AND operators respectively. In general, we can define a matrix product over a general algebraic structure as follows.

Let R be a set of elements equipped with two operations $\oplus : R \times R \to R$ and $\odot : R \times R \to R$. Let $\oplus$ be commutative and associative: for all $x, y, z \in R$, $x \oplus y = y \oplus x$ and $(x \oplus y) \oplus z = x \oplus (y \oplus z)$. Then for any finite set $S \subseteq R$, $\bigoplus_{a \in S} a$ is well-defined – we do not have to specify the order in which the elements are summed.

Let n, d, e be positive integers. Let A be an $n \times d$ matrix with entries from R and let B be a $d \times e$ matrix with entries from R. We define the $(\oplus, \odot)$- product of A and B as the $n \times e$ matrix C given by

$$C[i,j] = \bigoplus_{k \in [d]} (A[i,k] \odot B[k,j]), \forall i \in [n], j \in [e].$$

Fact 1 *Let T be an upper bound on the time to evaluate $\odot$ or $\oplus$ on any two elements of R. Then the $(\oplus, \odot)$-product of an $n \times d$ by a $d \times e$ matrix can be computed in $O(ndeT)$ time.*

Examples. Table 2.1 gives some examples. Two $\leq$ operators are used:

1. $\leq^b$: $\mathbb{R} \cup \{-\infty, \infty\} \times \mathbb{R} \cup \{-\infty, \infty\} \to \{0,1\}$ is given by $x \leq^b y = 1$ iff $x \leq y$;
2. $\leq^r$: $\mathbb{R} \cup \{-\infty, \infty\} \times \mathbb{R} \cup \{-\infty, \infty\} \to \mathbb{R} \cup \{-\infty, \infty\}$ is given by $x \leq^r y = y$ if $x \leq y$ and ∞ otherwise.

Similar $\geq$ operators can also be defined.

1. $\geq^b$: $\mathbb{R} \cup \{-\infty, \infty\} \times \mathbb{R} \cup \{-\infty, \infty\} \to \{0,1\}$ is given by $x \geq^b y = 1$ iff $x \geq y$;
2. $\geq^r$: $\mathbb{R} \cup \{-\infty, \infty\} \times \mathbb{R} \cup \{-\infty, \infty\} \to \mathbb{R} \cup \{-\infty, \infty\}$ is given by $x \geq^r y = y$ if $x \geq y$ and $-\infty$ otherwise.

Many of the above matrix products have symmetric ones obtained by "inverting" the operators: $(\wedge, \vee)$-product, $(\min, \max)$-product, $(\max, \geq^r)$-product.

2.3 Matrix Products and Graph Paths

A path or walk in a graph is a sequence of vertices every consecutive two of which are linked by an edge. If the path does not contain any repeating nodes it is called simple. The length of a path is the number of edges in the path. We give a connection between paths and matrix products.

2.3.1 Three-layered graphs

A three-layered graph $G = (V, E)$ is a directed tripartite graph on partitions $L = \{\ell_1, \ldots, \ell_{|L|}\}$, $M = \{m_1, \ldots, m_{|M|}\}$ and $Q = \{q_1, \ldots, q_{|Q|}\}$, $V = L \cup M \cup Q$. The edge set E of G consists of $L \times M$ and $M \times Q$. Figure 2.2 shows an example.

Product	R	$\oplus$	$\odot$	0	1
Algebraic	$\mathbb{R}$	$+$	$\cdot$	0	1
Boolean	$\{0,1\}$	$\vee$	$\wedge$	0	1
Distance	$\mathbb{R}^+ \cup \{0,\infty\}$	min	$+$	∞	0
Reliability	$[0,1]$	max	$\cdot$	0	1
MaxMin	$\mathbb{R} \cup \{-\infty,\infty\}$	max	min	$-\infty$	∞
Dominance	$\mathbb{R} \cup \{-\infty,\infty\}$	$+$	$\leq^b$	0	N/A
$(\min, \leq^r)$	$\mathbb{R} \cup \{-\infty,\infty\}$	min	$\leq^r$	∞	$-\infty$
$(+, \min)$	$\mathbb{R} \cup \{\infty\}$	$+$	min	0	∞

Table 2.1: Some $(\oplus, \odot)$-products appearing in the literature. This thesis discusses all products in the table except the reliability product.

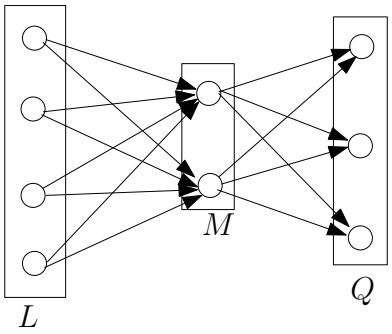

Figure 2.2: An example of a three-layered graph with $|L| = 4$, $|M| = 2$, and $|Q| = 3$.

Suppose that G has a weight function $w : E \to R$ defined on its edges. Let $(R, \oplus, \odot)$ be an algebraic structure defined on R as given in the previous section. For every pair of nodes $\ell_i \in L$ and $q_j \in Q$, define the length $\ell(\ell_i, q_j)$ of the shortest path from ℓ_i to q_j as $\bigoplus_{k\in[|M|]}(w(\ell_i, m_k) \odot w(m_k, q_j))$.

Let A be an $|L| \times |M|$ matrix with entries $A[i, j] = w(\ell_i, m_j)$. Similarly, let B be an $|M| \times |Q|$ matrix with entries $B[j, k] = w(m_j, q_k)$. Then the (i, j)th entry of the $(\oplus, \odot)$-product C of A and B is exactly

$$C[i, j] = \bigoplus_{k\in[|M|]} (w(\ell_i, m_k) \odot w(m_k, q_j)) = \ell(\ell_i, q_j).$$

Hence we have shown a relationship between shortest paths in three-layered graphs with weights from an algebraic structure and matrix products over the same structure.

2.3.2 Paths in general graphs

Let us now consider a general directed graph with edge weights from R. We use the following relatively straightforward notation using "$\to$": for a path P ending and node p and path Q starting at p, we denote by $P \to Q$ the path formed by first taking P and then continuing with Q; for nodes $x_0, x_1, \ldots, x_k$ such that there is an edge (x_i, x_{i+1}) for all $i = 1, \ldots, k-1$, we denote by $x_1 \to x_2 \to \ldots \to x_k$ the path that goes through each x_i consecutively, following the edges (x_i, x_{i+1}).

We define the weight of a path inductively as follows. If the path has length 0, it has weight 0. If path P is just an edge (x, y), its weight $w(P)$ is defined to be $w(x, y)$. If path P has length $t > 1$, *i.e.* $P = x_0 \to x_1 \to \ldots \to x_t$, then its weight is given by

$$w(P) = (w(x_0 \to \ldots \to x_{t-1})) \odot w(x_{t-1}, x_t).$$

Let P_{xy} denote the set of all finite paths between node x and node y. We would like to define the shortest distance between x and y as $D[x, y] = \bigoplus_{P\in P_{xy}} w(P)$. Unfortunately, as P_{xy} is infinite, $D[x, y]$ might not be well-defined. We will give some restrictive properties of $(R, \oplus, \otimes)$ for which $D[x, y]$ is well-defined. We note that there has been a lot of work on path algebra/semiring frameworks for path problems (see e.g. [61, 65]) that investigates a restriction of our current setting. See Section 2.4 for more details.

Additive identity and multiplicative annihilator. Suppose that there is an element $0 \in R$ such that $(R, \oplus, 0)$ is an Abelian monoid: $\oplus$ is commutative and associative, and 0 is an identity element, *i.e.* $x \oplus 0 = 0 \oplus x = x$ for all $x \in R$. Moreover, suppose that $0 \odot x = x \odot 0 = 0$ for all $x \in R$, *i.e.* that 0 is a multiplicative annihilator.

Given a directed graph $G = (V, E)$ with edge weights from R, we will construct a complete graph $G' = (V, V \times V)$ with edge weights $w' : (V \times V) \to R$ such that paths in G' consisting of edges only from E will have the same weights in G' as in G. Paths containing at least one edge from $(V \times V) \setminus E$ will have weight 0. This means that $\bigoplus_{P \text{ from } x \text{ to y}} w'(P) = \bigoplus_{P \text{ from } x \text{ to y}} w(P)$ for all $x, y \in V$, and shortest distances are preserved. We call (G', w') the *R-completion* of G.

This is relatively easy to accomplish. We let $w'(e) = w(e)$ if $e \in E$ and $w'(e) = 0$ otherwise. Then clearly any path with edges in E will have the same weight as in G. Consider a path P with at least one edge not in E. Let (x_i, x_{i+1}) be the first edge in P that is not in E. Then $P' = x_0 \to$

$x_1 \to \ldots x_i$ has weight $w(P_i)$. The weight of $P' \to x_{i+1}$ is $(w(P')) \odot w(x_i, x_{i+1}) = w(P') \odot 0 = 0$. If path P consists of a path P' of weight 0 followed by a single edge e, then

$$w'(P) = (w'(P')) \odot w'(e) = 0 \odot w'(e) = 0,$$

Hence any path containing at least one edge not in E has weight 0.

Left multiplicative identity. A left multiplicative identity of an algebraic structure $(R, \oplus, \odot)$ is an element 1 from R such that for all $x \in R$, $1 \odot x = x$. Table 2.1 states left multiplicative identities for all structures mentioned until now. Notice that it is not necessary that $x \odot 1 = x$. For instance, the element $-\infty$ is a left multiplicative identity for $(\mathbb{R} \cup \{-\infty, \infty\}, \min, \leq^r)$, but for all $x \in \mathbb{R} \cup \{\infty\}$ $x \leq^r -\infty = \infty$.

Let $\bar{R} = (R, \oplus, \odot, 0, 1)$ refer to an algebraic structure $(R, \oplus, \odot)$ with left multiplicative identity 1 and additive identity 0, such that $(R, \oplus)$ is a commutative monoid, and 0 is a multiplicative annihilator.

Suppose that $\bar{R} = (R, \oplus, \odot, 0, 1)$ is given. Let G be a directed graph on n nodes and edge weights w from R. Let (G', w') be the R-completion of G. We define the *adjacency matrix* A of G as the $n \times n$ matrix with entries from R so that for every $i, j \in [n]$:

$$A[i,j] := \begin{cases} w'(i,j) & \text{if } i \neq j \\ 1 & \text{otherwise.} \end{cases}$$

We can also define the $n \times n$ left *identity matrix* I over R as follows.

$$I[i,j] := \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{otherwise.} \end{cases}$$

Let $M(n, R)$ be the set of $n \times n$ matrices over R. Recall that multiplication of two $n \times n$ matrices $A, B \in M(n, R)$ is defined as follows. For every $i, j \in [n]$,

$$(A \odot B)[i,j] = \bigoplus_{k \in [n]} (A[i,k] \odot B[k,j]).$$

Clearly, for any $n \times n$ matrix B over R, and every $i, j \in [n]$, $(I \odot A)[i,j] = I[i,i] \odot A[i,j] = A[i,j]$, and hence $(I \odot A) = A$.

We define matrix addition under $\oplus$ as componentwise $\oplus$-addition, *i.e.* for all $i, j \in [n]$ and any two $n \times n$ matrices A, B over R, we have $(A \oplus B)[i,j] = (A[i,j] \oplus B[i,j])$. For $M(n, R)$ this $\oplus$ operation inherits the properties of the $\oplus$ operation for R: $\oplus$ is still associative and commutative. If Z is the $n \times n$ matrix of all 0s, then Z is an additive identity for $M(n, R)$. Furthermore, for any $A \in M(n, R)$, $Z \odot A = A \odot Z = Z$. Hence $(M(n, R), \oplus, \odot, Z, I)$ inherits all the properties of $(R, \oplus, \odot, 0, 1)$.

Paths of length 2 and squaring the adjacency matrix. We will now investigate the relationship between paths of length 2 and the square of the adjacency matrix of a general graph. This is somewhat similar to the three-layered graph case.

$$
\begin{aligned}
A^2[i,j] &= \bigoplus_k (A[i,k] \odot A[k,j]) = && (2.1)\\
&= (A[i,i] \odot A[i,j]) \oplus (A[i,j] \odot A[j,j]) \oplus \bigoplus_{k \neq i,j} (A[i,k] \odot A[k,j]) = && (2.2)\\
&= (1 \odot A[i,j]) \oplus (A[i,j] \odot 1) \oplus \bigoplus_{k \neq i,j} (A[i,k] \odot A[k,j]) = && (2.3)\\
&= (A[i,j] \oplus (A[i,j] \odot 1)) \oplus \bigoplus_{k \neq i,j} (A[i,k] \odot A[k,j]). && (2.4)
\end{aligned}
$$

The first thing we notice is that $A^2[i,j]$ is exactly the distance between i and j if we only consider paths of length 2.

Definition 2.3.1 *We say that* $\bar{R} = (R, \oplus, \odot, 0, 1)$ *is* 1-monotone *if* $(x \oplus (x \odot 1)) = x$ *for all* $x \in R$

Using equality (2.4) we also get that if $\bar{R}$ is 1-monotone, then $A^2[i,j]$ gives the distance between i and j when only considering *simple paths* of length at most 2.

Right distributivity and additive idempotence. We say that $\bar{R} = (R, \oplus, \odot, 0, 1)$ is right-distributive if for all $x, y, z \in R$, $(x \oplus y) \odot z = (x \odot z) \oplus (y \odot z)$. $\bar{R}$ is idempotent if for all $x \in R$, $x \oplus x = x$.

Assume from now on that $\bar{R}$ has all the properties described above and is also right-distributive and idempotent. In the standard definitions of abstract algebra, $\bar{R}$ is an idempotent non-associative near-semiring with a left multiplicative identity, and in which 0 is both a left and a right annihilator. We will call $\bar{R}$ an *algebraic path structure*. Formally:

Definition 2.3.2 *An* algebraic path structure *is an algebraic structure* $\bar{R} = (R, \oplus, \odot, 0, 1)$ *with the following properties:*

- *(***closure***) for all* $x, y \in R$, $x \oplus y \in R$ *and* $x \odot y \in R$
- *(***additive commutativity***) for all* $x, y \in R$, $x \oplus y = y \oplus x$
- *(***additive associativity***) for all* $x, y, z \in R$, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- *(***additive identity***) for all* $x \in R$, $x \oplus 0 = 0 \oplus x = x$
- *(***multiplicative annihilator***) for all* $x \in R$, $x \odot 0 = 0 \odot x = 0$
- *(***left multiplicative identity***) for all* $x \in R$, $1 \odot x = x$
- *(***right distributivity***) for all* $x, y, z \in R$, $(x \oplus y) \odot z = (x \odot z) \oplus (y \odot z)$
- *(***idempotence***) for all* $x \in R$, $x \oplus x = x$

Let P^t_{xy} be the set of paths from x to y in G of length at most t, for $t \geq 2$. For $t \geq 2$, define $D^t[x,y] = \bigoplus_{P \in P^t_{xy}} w(P)$. Then for $t > 2$,

$$\begin{aligned} D^t[x,y] &= \bigoplus_{P \in P^t_{xy}} w(P) = \left(\bigoplus_{P \in P^{t-1}_{xy}} w(P) \right) \oplus \left(\bigoplus_{z \in V} \bigoplus_{P' \in P^{t-1}_{xz}} \left(w(P') \odot w'(z,y) \right) \right) && (2.5) \\ &= \left(\bigoplus_{P \in P^{t-1}_{xy}} w(P) \right) \oplus \left(\bigoplus_{z \in V} \left(\bigoplus_{P' \in P^{t-1}_{xz}} w(P') \right) \odot w'(z,y) \right) && (2.6) \\ &= D^{t-1}[x,y] \oplus \bigoplus_{z \in V} \left(D^{t-1}[x,z] \odot w'(z,y) \right). && (2.7) \end{aligned}$$

It is tempting to define the shortest distance between two nodes x and y as $D[x,y] = D[x,y] \oplus (\bigoplus_z (D[x,z] \odot w'(z,y)))$, but unfortunately $D[x,y] = 0$ for all $x, y \in V$ is a solution to this equation. We can use idempotence to obtain a better definition.

Because of our definition of the adjacency matrix, $I \oplus A = A$. For $t \geq 2$, let D^t be the $n \times n$ matrix with x, y entry $D^t[x,y]$. Then equation 2.7 is equivalent to $D^t = D^{t-1} \oplus (D^{t-1} \odot A)$. Let $D^1 = A$. Then we also have $D^2 = A \oplus (A \odot A) = D^1 \oplus (D^1 \odot A)$.

Notice that for all $t \geq 2$,

$$D^{t-1} \oplus D^t = D^{t-1} \oplus D^{t-1} \oplus (D^{t-1} \odot A) = D^{t-1} \oplus (D^{t-1} \odot A) = D^t.$$

We show that $D^t = I \oplus (D^{t-1} \odot A)$ for all $t \geq 2$.

By induction on t. The base case is $t = 2$: $D^2 = A \oplus (A \odot A) = I \oplus (I \odot A) \oplus (A \odot A) = I \oplus (I \oplus A) \odot A = I \oplus (A \odot A) = I \oplus (D^1 \odot A)$.

Suppose $D^{t-1} = I \oplus (D^{t-2} \odot A)$. Since we have

$$\begin{aligned} D^t &= D^{t-1} \oplus (D^{t-1} \odot A) = I \oplus (D^{t-2} \odot A) \oplus (D^{t-1} \odot A) && (2.8) \\ &= I \oplus ((D^{t-2} \oplus D^{t-1}) \odot A) = I \oplus (D^{t-1} \odot A). && (2.9) \end{aligned}$$

Now we can give a better definition which is exactly the definition used when using path algebras [61]:

Definition 2.3.3 *The* shortest distance problem *for a graph with adjacency matrix A over an algebraic path structure is to find a solution (if it exists) of the equation $D = I \oplus (D \odot A)$. We refer to such a solution as a* shortest distance matrix *of the graph.*

If $D = I \oplus (D \odot A)$ has two solutions D_1 and D_2, then $D_1 \oplus D_2$ is also a solution:

$$D_1 \oplus D_2 = I \oplus I \oplus (D_1 \odot A) \oplus (D_2 \odot A) = I \oplus (D_1 \oplus D_2) \odot A.$$

Then, for some algebraic path structures we can restrict the transitive closure definition as follows

Definition 2.3.4 *Let $\bar{R} = (R, \oplus, \otimes, 0, 1)$ be an algebraic path structure for which $\bigoplus_{x \in S} x$ is well-defined for all (finite or infinite) sets $S \subseteq R$. Let A be a matrix over $\bar{R}$, such that A is the adjacency matrix of a graph G. Let $\Delta \subseteq M(n, R)$ be the set of all matrices that are solutions to the equation $D = I \oplus (D \odot A)$. Then the shortest distance matrix of G with adjacency matrix A is the matrix $\bar{D}$ with*

$$\bar{D} = \bigoplus_{D \in \Delta} D.$$

In many cases (such as for min and max), $\oplus$ has the property that for all $x, y \in R$, $x \oplus y \in \{x, y\}$. This defines a total order on R by $x \leq y$ iff $x \oplus y = x$. If Δ contains an infimum with respect to this ordering, then the definition above can be applied even if R is not well-ordered. Under Definition 2.3.4 the shortest distance matrix is also referred to as the *minimum* solution to $D = I \oplus (D \odot A)$. In general, however, we will not use Definition 2.3.4. It will suffice to use Definition 2.3.3.

Transitive closure and the path condition. The transitive closure T of a matrix A is defined as a solution of the equation $T = I \oplus (T \odot A)$ in a similar way to the definition of the shortest distance matrix (2.3.3, 2.3.4). Hence, when A is the adjacency matrix of a graph as in the previous context, T is also the shortest distance matrix of the graph. The transitive closure of A might not be defined but there are many cases in which it is. Some examples can be found in Section 2.4.

Definition 2.3.5 *Let $\bar{R} = (R, \oplus, \odot, 0, 1)$ be an algebraic path structure. $\bar{R}$ is said to satisfy the* path condition *if for all $x, y \in R$,*

$$x \oplus (x \odot y) = x.$$

The path condition generalizes the notion of 1-monotonicity. Even though this condition seems like a huge restriction, notice that the distance, reliability, MaxMin and $(\min, \leq^r)$-products all satisfy the path condition. These are exactly the products we will employ for path problems.

We will prove the following.

Theorem 2.3.1 *Let $\bar{R} = (R, \oplus, \odot, 0, 1)$ be an algebraic path structure satisfying the path condition. Then for any graph G on $\bar{R}$:*

- *the shortest distance matrix D of G is well-defined, and*
- *D can be computed in $O(n(M(n,n) + S(n,n)))$ time, where $M(n,n)$ and $S(n,n)$ are the time to multiply and sum (respectively) two $n \times n$ matrices over $\bar{R}$.*

Proof. Consider a path P from x to y composed of a path P_1 from x to z, followed by a path C from z back to z, followed by a path P_2 from z to y. Let $C = z \to z_1 \to \ldots \to z_p \to z$, and $P_2 = z \to y_1 \to \ldots \to y_k \to y$. Then the weight of P is

$$w(P) = (((w(P_1 \to C)) \odot w(z, y_1)) \odot \ldots) \odot w(y_k, y).$$

Consider $w(P_1 \to P_2) \oplus w(P)$. This equals

$$(((w(P_1) \oplus w(P_1 \to C)) \odot w(z, y_1)) \odot \ldots) \odot w(y_k, y).$$

From the path condition we have the following base case for our induction on j:

$$w(P_1) = w(P_1) \oplus ((w(P_1)) \odot w(z, z_1)).$$

Assume that

$$w(P_1) = w(P_1) \oplus (((w(P_1)) \odot w(z, z_1)) \odot \ldots) \odot w(z_{j-1}, z_j).$$

Then by the path condition and the above assumption,

$$\begin{aligned} w(P_1) &= w(P_1) \oplus (((w(P_1)) \odot w(z, z_1)) \odot \ldots) \odot w(z_{j-1}, z_j) && (2.10)\\ &= w(P_1) \oplus [(((w(P_1)) \odot w(z, z_1)) \odot \ldots) \odot w(z_{j-1}, z_j)] \oplus && (2.11)\\ \oplus\ & [(((((w(P_1)) \odot w(z, z_1)) \odot \ldots) \odot w(z_{j-1}, z_j)) \odot w(z_j, z_{j+1})] && (2.12)\\ &= w(P_1) \oplus ((((w(P_1)) \odot w(z, z_1)) \odot \ldots) \odot w(z_j, z_{j+1}). && (2.13) \end{aligned}$$

By the above inductive argument we obtain that

$$w(P_1) = w(P_1) \oplus w(P_1 \to C).$$

By the distributive property we also obtain that

$$w(P_1 \to P_2) = w(P_1 \to P_2) \oplus w(P_1 \to C \to P_2).$$

Then for any two nodes x, y, their distance can be defined as

$$D[x, y] = \bigoplus_{P \in P_{xy}} w(P) = \bigoplus_{P \in SP_{x,y}} w(P),$$

where $SP_{x,y}$ is the set of all simple paths from x to y. As the number of simple paths from x to y is finite (it is at most $(n-2)!$), the distance is always well-defined.

Since any simple path has length at most n, in order to compute D, it suffices to compute D^n, i.e. starting from $D^1 = A$, for i ranging from 2 to n, compute $D^i = I \oplus (D^{i-1} \odot A)$. This computation requires doing n sums and products of $n \times n$ matrices. □

2.4 Semiring Frameworks

We give a brief summary of some results on path problems using the so called semiring or path algebra framework. Most of this can be found in the wonderful survey by Bernd Mahr [61]. Semirings are more restricted algebraic structures than the ones we considered in the previous section. Lengauer and Theune [60] considered converting some weaker algebraic structures into semirings by adding the properties of associativity and distributivity. Although this approach is very general, it increases the running times of the transitive closure algorithms. From now on we only consider semirings.

A *semiring or* path algebra is an algebraic structure $(R, \oplus, \odot, 0, 1)$, such that $(R, \oplus, 0)$ and $(R, \odot, 1)$ are both monoids, $\oplus$ is commutative, $\odot$ distributes over $\oplus$ (both from the right and from the left), and 0 is both a right and a left multiplicative annihilator. A semiring is idempotent if for all $x \in R$, $x \oplus x = x$. An algebraic path structure is an idempotent semiring that is missing the following properties: $\odot$-associativity, left $\odot$-distributivity over $\oplus$, and right multiplicative identity.

An idempotent semiring does not need the path condition in order to guarantee that shortest distances are well-defined. Instead, it is sufficient if it is *closed*.

Closed semirings. A semiring is closed if there is a unary operation $*$ so that $a^* = 1 \oplus (a \odot a^*)$ for all $a \in R$. For closed idempotent semiring there is always a solution to the shortest distance

equation $D = I \oplus (D \odot A)$. This solution is called the *Kleene* matrix A^*. For graphs on n nodes, $A^* = A^{(n)}$, where $A^{(0)} = A$, and for $r > 0$ and $i, j \in [n]$,

$$A_{ij}^{(r)} = A_{ij}^{(r-1)} \oplus \left(A_{ir}^{(r-1)} \odot (A_{rr}^{(r-1)})^* \odot A_{rj}^{(r-1)} \right).$$

The algorithms textbook by Aho, Hopcroft and Ullman [1] gives a theorem concerning the algorithmic complexity of computing the transitive closure of a matrix over a closed semiring. Informally, this theorem, originally due to Fischer and Meyer [39], Furman [45, 46], and Munro [67], states that the transitive closure of such a matrix A can be computed in approximately the same time as the time to multiply two matrices of the same size as A over the same semiring.

Theorem 2.4.1 ([39, 46, 67], [1], pp. 204–206) *Let $T(n)$ be the time to compute the transitive closure of an $n \times n$ matrix over a closed idempotent semiring $\bar{R}$. Let $M(n)$ be the time to compute the product of two $n \times n$ matrices over $\bar{R}$. If $M(2n) \geq 4M(n)$ and $T(3n) \leq 27T(n)$, then $T(n) = \Theta(M(n))$.*

The theorem makes two assumptions that are very reasonable, as we expect $M(n) = \Omega(n^2)$ (we need to write the output matrix), and $T(n) = O(n^3)$ (with closed semirings we can use the Floyd-Warshall algorithm for transitive closure, see below). Furthermore, with some work these assumptions can be weakened to $M(3n) = \Omega(M(n))$ and $T(3n) = O(T(n))$.

Matrix stability. There is even a weaker condition for the transitive closure to be well-defined for matrices over idempotent semirings. This condition is not on the semiring itself but on the given matrix. It requires that there is a constant $q > 0$, so that the matrix is q-stable.

Definition 2.4.1 *A matrix A is q-stable if*

$$\bigoplus_{r=0}^{q} A^i = \bigoplus_{r=0}^{q+1} A^i,$$

where $A^0 = I$ and $A^i = A^{i-1} \odot A$ for $i > 0$.

The following theorem due to Floyd and Warshall [40, 99], among others, asserts that q-stability is a sufficient condition for the transitive closure of a matrix to be well-defined.

Theorem 2.4.2 ([40, 99]) *Let R be an idempotent semiring. Let A be a q-stable square matrix over R, then $A^* = \bigoplus_{i=0}^{q} A^i$ is a solution to the shortest distance equation $D = I \oplus (D \odot A)$.*

Theorem 2.4.2 also gives an $O(n^3)$ time algorithm for finding the transitive closure of an $n \times n$ q-stable matrix A. This algorithm is the immediate generalization of the Floyd-Warshall algorithm for the shortest paths problem:

```
for all   i, j = 1 to n,
          D[i][j] := A[i][j];

for all   k, i, j = 1 to n,
          D[i][j] := D[i][j] ⊕ (D[i][k] ⊙ D[k][j]);
```

Various conditions imply the stability of a matrix. For instance, A is $(n-1)$-stable if A is the adjacency matrix of a DAG, or if A is over an idempotent semiring and for all r, $A_{ii}^r \oplus 1 = 1$. A special case of this second condition is when A is the adjacency matrix of a graph G with real weights on its edges such that G has no cycles of negative weight sum. Then A is considered to be over the so called tropical semiring $(\mathbb{R} \cup \{-\infty, \infty\}, \min, +, \infty, 0)$. This ends our discussion of semirings. The next chapter will present good algorithms for most algebraic structure products given in Table 2.1.

Chapter 3

Efficient Algorithms for Some Matrix Products

In the following sections we discuss algorithms for the $(\oplus, \otimes)$ matrix products defined in the previous chapter. Our algorithms are made asymptotically efficient using the subcubic results on fast matrix multiplication [28]. We use the following general techniques.

1. **Bucketting:** This technique deals with preprocessing the entries of each input matrix and assigning them in a 1-to-1 fashion to some number of *buckets*, so that each bucket has a small number of entries. After this, for each input matrix X and each bucket b, one creates a new matrix X_b. This bucketting allows later computations to narrow the search space.

2. **Bucket Processing:** For each fixed bucket b, one takes the matrices A_b and B_b created by the bucketting step and multiplies them using a different matrix product $(\oplus', \otimes')$. The algorithm for this different matrix product might use these techniques recursively, or if it is just the algebraic matrix product, fast matrix multiplication is used.

3. **Exhaustive Search:** The bucket processing step provides some information which allows us to choose a small number of buckets on which the problem is solved by exhaustive search. Because the buckets are small by construction, this step takes less time than exhaustive search on the original input.

In all of our matrix product algorithms below the above techniques are used repeatedly.

3.1 Dominance Product

We begin with the dominance product, which will be used in all of the rest of our matrix product algorithms. Recall that this product is over the algebraic structure $(\mathbb{R} \cup \{-\infty, \infty\}, +, \leq^b)$, where on input the ordered pair (x, y), the $\leq^b$ operation returns 1 if $x \leq y$ and 0 otherwise.

Given a set of points $\{v_1, \ldots, v_n\}$ in $\mathbb{R}^d$, the well-known *dominating pairs problem* in computational geometry is to find all pairs of points (v_i, v_j) such that for all $k = 1, \ldots, d$, $v_i[k] \leq v_j[k]$. This problem has many applications, for instance, in VLSI circuit design and databases.

When the number of dimensions of the space is linear in the number of points, one can compute the dominating pairs by employing the dominance product. In particular, the points are laid out as rows in a matrix A for which the columns are the coordinates. The dominance product of A

with its transpose A^t counts for each pair of points v_i, v_j the number of coordinates in which v_j dominates v_i. Then, (v_i, v_j) is a dominating pair if and only if the i, j entry in $A \odot A^t$ is exactly n.

Matousek [62] gave an elegant algorithm computing the dominance product. This algorithm uses practically the same techniques described in the beginning of this chapter.

Theorem 3.1.1 (Matousek [62]) *Let A and B be real $n \times n$ matrices. The dominance product $C = (A \odot B)$ can be computed in $O\left(n^{\frac{3+\omega}{2}}\right)$ time.*

Proof. We outline Matousek's approach. Consider the rows of A and columns of B together as $2n$ points in n-dimensional space. Let the rows of A be the first n points, $\{v_1, \ldots, v_n\}$, and the columns of B be the second n points, $\{v_{n+1}, \ldots, v_{2n}\}$. For each coordinate $j = 1, \ldots, n$, sort the points $\{v_i, i = 1, \ldots, 2n\}$ by coordinate j. This takes $O(n^2 \log n)$ time.

Define the *jth rank of point v_i*, denoted as $r_j(v_i)$, to be the position of v_i in the sorted list for coordinate j. Let $s \in [\log n, n]$ be a parameter to be determined later. This parameter corresponds to a bound on the size of a bucket, in our techniques. We have a bucket for each $k \in [n/s]$. Define n/s pairs of $(0,1)$ matrices (A_1, B_1), ..., $(A_{n/s}, B_{n/s})$ as follows:

$$A_k[i,j] = 1 \iff r_j(v_i) \in [ks, ks+s),$$

$$B_k[i,j] = 1 \iff r_j(v_{i+n}) \geq ks + s.$$

This concludes the **bucketting** step.

Now, multiply A_k with B_k^T, obtaining a matrix C_k for each bucket $k = 1, \ldots, n/s$. Then $C_k[i,j]$ equals the number of coordinates c such that $v_i[c] \leq v_j[c]$, $r_c(v_i) \in [ks, ks+s)$, and $r_c(v_j) \geq ks + s$. Therefore, letting

$$C = \sum_{k=1}^{n/s} C_k,$$

we have that $C[i,j]$ is the number of coordinates c such that $\lfloor r_c(v_i)/s \rfloor < \lfloor r_c(v_{n+j})/s \rfloor$. This concludes the **bucket processing** step.

Suppose we compute a matrix E such that $E[i,j]$ is the number of coordinates c such that $v_i[c] \leq v_j[c]$ and $\lfloor r_c(v_i)/s \rfloor = \lfloor r_c(v_{n+j})/s \rfloor$. Then, defining $D := C + E$, we will have the desired dominance product matrix

$$D[i,j] = |\{k \mid v_i[k] \leq v_j[k]\}|.$$

To compute E, we use the n sorted lists. For each pair $(i,j) \in [n] \times [n]$, we look up v_{n+i}'s position p in the sorted list for coordinate j. By reading off the adjacent points less than v_{n+i} in this sorted list (*i.e.* the points at positions $p-1$, $p-2$, *etc.*), and stopping when we reach a point v_k such that $\lfloor r_j(v_k)/s \rfloor < \lfloor r_j(v_{n+i})/s \rfloor$, we obtain the list $v_{i_1}, \ldots, v_{i_\ell}$ of $\ell \leq s$ points such that $i_x \in [n]$, $v_{i_x}[j] \leq v_{n+i}[j]$ and $\lfloor r_j(v_{n+i})/s \rfloor = \lfloor r_j(v_{i_x})/s \rfloor$. For each $x = 1, \ldots, \ell$, if $i_x \leq n$, we add a 1 to $E[i_x, i]$. Assuming constant time lookups and constant time probes into a matrix, this entire process takes only $O(n^2 s)$ time. This concludes the **exhaustive search** step.

The runtime of the above procedure is $O(n^2 s + \frac{n}{s} n^\omega)$. Choosing $s = n^{\frac{\omega-1}{2}}$, the time bound becomes $O(n^{\frac{3+\omega}{2}})$. □

We note that the dominance product allows us to easily obtain a subcubic algorithms for the $(+,<)$, $(+,>)$ and $(+,=)$-products, where $<, >, =$ on input a, b return 1 if $a < b$, $a > b$ and $a = b$ respectively, and 0 otherwise.

Corollary 3.1.1 *If the dominance product of two arbitrary real $n \times n$ matrices can be computed in $O(D(n))$ time, then the $(+,=)$, $(+,<)$ and $(+,>)$-products of two arbitrary real $n \times n$ matrices can all be computed in $O(D(n))$ time. Hence these products are all computable in $O(n^{\frac{3+\omega}{2}})$.*

Proof. Let A and B be the matrices for which we want to compute $C, C_>$ and $C_>$ such that for all $i, j \in [n]$

$$\begin{aligned} C_=[i,j] &= \sum_{k=1}^{n}(A[i,k] = B[k,j]), \\ C_<[i,j] &= \sum_{k=1}^{n}(A[i,k] < B[k,j]), \\ C_>[i,j] &= \sum_{k=1}^{n}(A[i,k] > B[k,j]). \end{aligned}$$

Compute the dominance product $C_\leq$ of A and B and the dominance product $C_\geq$ of B and A. For every $i, j \in [n]$, let

$$\begin{aligned} C_=[i,j] &= C_\leq[i,j] + C_\geq[i,j] - n, \\ C_<[i,j] &= C_\leq[i,j] - C_=[i,j], \\ C_>[i,j] &= C_\geq[i,j] - C_=[i,j]. \end{aligned}$$

□

There has been no progress on obtaining faster algorithms for the dominance product since Matousek's algorithm. Since no lower bounds for the problem are known it is still possible that there is an $O(n^\omega)$ algorithm for the dominance product of two $n \times n$ matrices. It is also possible that the $(+,=)$-product of two matrices is computationally easier than the dominance product. We currently also do not know a better algorithm for the so called *existence*-dominance product which has a 1 in the i, j entry if the dominance product in that entry is n and 0 otherwise. In the following pages we give better algorithms for the dominance product for two special cases.

The low dimensional case. Here we give a slight but useful extension of the Matousek's algorithm that uses rectangular matrix multiplication to get an improved time bound, when the points are in $\mathbb{R}^d$ with $d << n$. This result appears in [95].

Theorem 3.1.2 *Given an $n \times d$ matrix A and a $d \times n$ matrix B, the dominance matrix C of A and B can be computed in $O(n^{1.898} \cdot d^{0.696} + n^{2+\varepsilon})$ time for all $\varepsilon > 0$ if $d = O(n^{\frac{\omega-1}{2}})$, or in $O(n \cdot d^2 + n^\omega)$ time for all d.*

So for example, when $d = o(n^{0.688})$, we can compute the dominance matrix in $o(n^\omega)$.

Let $\omega(n, d, m)$ denote the exponent of multiplying an $n \times d$ by a $d \times m$ matrix. We recall a Lemma by Huang and Pan [53] also appearing in [106]:

Lemma 3.1.1 ([53]) *Let $\alpha = \sup\{0 \leq r \leq 1 \mid w(n, n^r, n) = 2 + o(1)\} > 0.294$. Then for all $d \geq n^\alpha$, one can multiply an $n \times d$ by a $d \times n$ matrix in time*

$$O(d^{\frac{\omega-2}{1-\alpha}} \cdot n^{\frac{2-\omega\alpha}{1-\alpha}}) = O(d^{0.533} n^{1.844}).$$

Proof of Theorem 3.1.2. We begin analogous to Matousek's approach. The rows of A and columns of B are considered as $2n$ points in d-dimensional space: the rows of A are the first n points, $\{v_1, \ldots, v_n\}$, and the columns of B are the second n points, $\{v_{n+1}, \ldots, v_{2n}\}$. As before, for each coordinate $j = 1, \ldots, d$, sort the points $\{v_i, i = 1, \ldots, 2n\}$ by coordinate j. This takes $O(nd \log n)$ time.

Recall, the *jth rank of point* v_i, denoted as $r_j(v_i)$, is the position of v_i in the sorted list for coordinate j. Let $s \in [\log n, n]$ be a parameter corresponding to a bound on the size of a bucket. Define n/s pairs of $n \times d$ $(0,1)$ matrices (A_1, B_1), ..., $(A_{n/s}, B_{n/s})$ as follows. For $k \in [n/s]$, $i \in [n]$, $j \in [d]$,

$$A_k[i,j] = 1 \iff r_j(v_i) \in [ks, ks+s),$$

$$B_k[i,j] = 1 \iff r_j(v_{i+n}) \geq ks + s.$$

This concludes the **bucketting** step.

The *bucket processing* step differs from the original Matousek approach. Instead of defining matrices C_k, we simply compute

$$C = \begin{bmatrix} A_1 & A_2 & \cdots & A_{n/s} \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n/s} \end{bmatrix},$$

that is, we are multiplying an $n \times dn/s$ and a $dn/s \times n$ matrix. This concludes the **bucket processing** step.

As in the Matousek approach we compute a matrix E such that $E[i,j]$ is the number of coordinates c such that $v_i[c] \leq v_j[c]$ and $\lfloor r_c(v_i)/s \rfloor = \lfloor r_c(v_j)/s \rfloor$. Then, defining $D := C + E$, we will have the desired dominance product matrix

$$D[i,j] = |\{k \mid v_i[k] \leq v_j[k]\}|.$$

Computing E in this situation proceeds as before and can be verified to take $O(n \cdot d \cdot s)$ time. This concludes the **exhaustive search** step.

As before, our goal is to choose s optimally so that this runtime is minimized. Results of Huang and Pan [53] (see Lemma 3.1.1) on rectangular matrix multiplication, coupled with the optimal choice of s show that we can either compute the dominance matrix in

$$O(n^{\frac{2\omega - \omega\alpha - 2}{\omega - \alpha - 1}} \cdot d^{\frac{2\omega - 4}{\omega - \alpha - 1}} + n^{2+\varepsilon}) \leq O(n^{1.898} d^{0.696})$$

time for all $\varepsilon > 0$. We can only make this choice of s when $d = O(n^{\frac{\omega - 1}{2}})$. By a slight modification of the algorithm we can obtain an $O(n \cdot d^2 + n^\omega)$ runtime for all $d = O(n)$. To do this, again sort the columns of A and rows of B but now bucket the sorted lists into buckets of size d. Create two matrices A' and B', both $n \times n$. The columns of A' and rows of B' are labeled by pairs (j, b) where j is a column of A (row of B) and b is a bucket from 1 to n/d. (There are at most n such pairs.) $A'[i, (j,b)]$ is 1 if $A[i,j]$ is in bucket b of the jth sorted list, and $A'[i, (j,b)] = 0$ otherwise. $B'[(j,b), k]$ is 1 if there is some $b' > b$ such that $B[j,k]$ is in bucket b' of the jth sorted list, and $B'[(j,b), k] = 0$ otherwise. Within buckets we do exhaustive search in overall $O(nd^2)$ time and to count between buckets we compute $A' \cdot B'$ over the integers in $O(n^\omega)$ summing the results. $\square$

The sparse case. In some of our applications that use a dominance product, we only want to perform comparisons with certain entries of the matrices. For example, suppose matrices A and B are over $\mathbb{R} \cup \{-\infty, \infty\}$, such that A has mostly ∞ entries, while B has mostly finite entries. Then, in the computation of the dominance product $A \odot B$, many of the comparisons ($A[i,k] \leq B[k,j]$) are false; it only makes sense to compare the *finite* entries of A with entries in B. To this end, we design a special algorithm for dominance product in the case where one wishes to ignore large portions of the A-matrix. The algorithm again uses the techniques described in the beginning of this chapter. It was published in [97].

Theorem 3.1.3 (Sparse Dominance Product) *Let A and B be $n \times n$ matrices over $\mathbb{R} \cup \{-\infty, \infty\}$. Let $S \subseteq [n] \times [n]$ such that $|S| = m \geq n$. Let C be the matrix such that*

$$C[i,j] = |\{k \mid (i,k) \in S \text{ and } A[i,k] \leq B[k,j]\}|.$$

There is an algorithm `SD` *that, given A, B, and S, outputs C in $O(\sqrt{m} \cdot n^{\frac{1+\omega}{2}})$ time.*

Proof. Call the entries of A with coordinates in S the *relevant* entries of A. For every $j = 1, \ldots, n$, let L_j be the sorted list containing the relevant entries from A in column j, along with the entries from B^T in column j. Let g_j be the number of relevant entries of A in L_j, for all j. Clearly, $\sum_j g_j = m$. Pick a parameter r and partition each L_j into r consecutive buckets, such that every bucket contains at most $\lceil g_j/r \rceil$ relevant entries of A. Note that the bucket sizes are not necessarily uniform.

To conclude the **bucketting** step, for every bucket number $b = 1, \ldots, r$, create Boolean matrices A_b and B_b:

$$A_b[i,j] := \begin{cases} 1 & \text{if } A[i,j] \text{ is in bucket } b \text{ of } L_j \\ 0 & \text{otherwise,} \end{cases}$$

$$B_b[j,k] := \begin{cases} 1 & \text{if } B[j,k] \text{ is in bucket } b' \text{ of } L_j \text{ and } b' > b \\ 0 & \text{otherwise.} \end{cases}$$

For each bucket number b, compute $C_b = A_b \cdot B_b$. This step takes $O(rn^\omega)$ time and computes for every pair i, k and bucket number b, the number of j such that $A[i,j] \leq B[j,k]$, where $A[i,j]$ is in bucket b of L_j, and $B[j,k]$ is in a different bucket of L_j. This concludes the **bucket processing** step.

Initialize an $n \times n$ matrix D to be all zeros. In every bucket b of L_j, there are at most $\lceil g_j/r \rceil$ relevant entries of A and some number t_{jb} of entries from B. Compare every A-entry with every B^T-entry in bucket b of column j in $O(t_{jb} \cdot \lceil g_j/r \rceil)$ time; in particular, for each $A[i,j] \leq B^T[k,j]$ where $A[i,j]$ and $B^T[k,j]$ are in bucket b, increment $D[i,k]$. Over all j and b, this takes time on the order of

$$\begin{aligned} \sum_j \sum_b t_{jb} \cdot \lceil g_j/r \rceil &\leq \sum_j (1 + g_j/r) \sum_b t_{jb} \\ &= \sum_j (1 + g_j/r) n \\ &= n^2 + \sum_j g_j n/r \\ &= n^2 + mn/r. \end{aligned}$$

After all buckets of all lists are processed, $D[i,k]$ contains the number of j such that $A[i,j] \leq B[j,k]$, where $A[i,j]$, $B^T[k,j]$ are in the same bucket of L_j. This concludes the **exhaustive search** step.

Finally, set $C = \sum_{b=1}^{r} C_b + D$. It is easy to verify from the above that the algorithm returns the desired C. The overall runtime of the above procedure is $O(n^2 + mn/r + rn^\omega)$. Choosing $r = \sqrt{m} \cdot n^{\frac{1-\omega}{2}}$, the runtime is minimized to $O(\sqrt{m} \cdot n^{\frac{1+\omega}{2}})$. □

Corollary 3.1.2 *There is an* $O(\min\{(|S_A| \cdot |S_B|)^{\frac{\omega-2}{\omega-\alpha-1}} n^{\frac{(2-\alpha\omega)}{\omega-\alpha-1}}, n^\omega + \sqrt{|S_A| \cdot |S_B|} \cdot n^{\frac{\omega-1}{2}})\}$ *algorithm for sparse dominance product, where* S_A *and* S_B *are subsets of* $[n] \times [n]$*, and the resulting matrix has* $C[i,j] = |\{k \mid A[i,k] \leq B[k,j], (i,k) \in S_A, (k,j) \in S_B\}|$.

Proof. Suppose A has m_1 relevant entries and B has m_2. Sort each column k of A and row k of B together. For each k, let L_k be the sorted list for column/row k. Bucket each list L_k into consecutive buckets, so that each bucket has at most m_1/D elements of A. Compare elements within bucket b of list k in

$$\sum_b g_{bk} \lceil m_1/D \rceil$$

where g_{bk} is the number of B-elements in bucket b of L_k. Overall, the runtime is $O(m_2 + m_1 m_2/D)$. To handle comparisons between buckets we do the following (similar to our approach for rectangular dominance product). Create matrices C and C' where C is $n \times O(D)$ and C' is $O(D) \times n$. The columns of C and rows of C' have indices (k,b) for bucket b of L_k, provided L_k has at least 2 buckets. We set $C[i,(k,b)]$ to be 1 if $A[i,k]$ is in bucket b of L_k. We set $C'[(k,b'),j]$ to be 1 if $B[k,j]$ is in some bucket $b > b'$ of L_k. Then clearly $C[i,(k,b)] \cdot C'[(k,b),j] = 1$ iff there is some $b' > b$ such that $B[k,j]$ is in bucket b' of L_k but $A[i,k]$ is in bucket $b < b'$ of L_k and when we sum these we always count different comparisons. The number of coordinates (k,b) is at most

$$\sum_{k:\ L_k \ \text{ has} \geq 2 \text{ buckets}} (\lceil m_{1k} D/m \rceil) \leq D + D = 2D.$$

If $D \geq n$, we can compute the product of C and C' in $O((D/n)n^\omega)$ time. For this case, the best value for D is

$$m_1 m_2/D = Dn^{\omega-1} \implies D = \sqrt{m_1 m_2}/n^{\frac{\omega-1}{2}}$$

The final runtime is then $O(\sqrt{m_1 m_2} n^{\frac{\omega-1}{2}})$.

If $D = o(n)$, then the product of C and C' can be computed in $O(D^{\frac{\omega-2}{1-\alpha}} n^{\frac{2-\alpha\omega}{1-\alpha}})$ time by Lemma 3.1.1 where the current best value for α is 0.294. The best value for D is then

$$m_2 m_1/D = D^{\frac{\omega-2}{1-\alpha}} n^{\frac{2-\alpha\omega}{1-\alpha}} \implies D = \frac{(m_1 m_2)^{\frac{1-\alpha}{\omega-\alpha-1}}}{n^{\frac{(2-\alpha\omega)}{\omega-\alpha-1}}}.$$

The final runtime becomes asymptotically

$$(m_1 m_2)^{\frac{\omega-2}{\omega-\alpha-1}} n^{\frac{(2-\alpha\omega)}{\omega-\alpha-1}} = (m_1 m_2)^{0.33} n^{1.21}.$$

□

The runtime of the corollary above can be further improved for $\sqrt{m_1 m_2} = o(n^{\frac{\omega+1}{2}})$ using the fast sparse matrix multiplication of Yuster and Zwick [104].

One can combine Theorem 3.1.3 and the rectangular matrix multiplication from Lemma 3.1.1 to obtain a more general result.

Corollary 3.1.3 *Let A be an $n \times d$ matrix and B be a $d \times n$ matrix over $\mathbb{R} \cup \{-\infty, \infty\}$. Let $S \subseteq [n] \times [d]$ such that $|S| = m$ with $n \leq m \leq n^\omega/d$. Let C be the matrix such that*

$$C[i,j] = |\{k \mid (i,k) \in S \text{ and } A[i,k] \leq B[k,j]\}|.$$

There is an algorithm that, given A, B, and S, outputs C in $O(\sqrt{m}d^{0.267}n^{1.422})$ time.

Proof. The matrices A_b and B_b in the proof of Theorem 3.1.3 are now $n \times d$ and $d \times n$ respectively. Hence computing $C_b = A_b \cdot B_b$ takes

$$O(rd^{\frac{w-2}{1-\alpha}}n^{\frac{2-\omega\alpha}{1-\alpha}}) = O(rd^{0.533}n^{1.844})$$

time by Lemma 3.1.1. The exhaustive search portion of the algorithm takes $O(nd + mn/r)$ time. We minimize the runtime:

$$rd^{\frac{w-2}{1-\alpha}}n^{\frac{2-\omega\alpha}{1-\alpha}} = mn/r \implies r = \sqrt{m}/(d^{\frac{\omega-2}{2(1-\alpha)}}n^{\frac{1+\alpha-\omega\alpha}{2(1-\alpha)}}).$$

The runtime becomes

$$O(\sqrt{m}d^{\frac{\omega-2}{2(1-\alpha)}}n^{\frac{3-\alpha-\omega\alpha}{2(1-\alpha)}}) = O(\sqrt{m}d^{0.267}n^{1.422}).$$

□

3.2 $(+, \min)$-Product

The proof approach of Theorem 3.1.1 can be extended to obtain a similar algorithm for the $(+, \min)$-product.

Theorem 3.2.1 *Given two $n \times n$ matrices A and B over $\mathbb{R} \cup \{\infty\}$, their $(+, \min)$-product C can be computed in $O\left(n^{\frac{3+\omega}{2}}\right)$ time.*

Proof. The proof is a slight generalization of the proof of Theorem 3.1.1. As before, consider the rows of A and columns of B together as $2n$ points in n-dimensional space: be the first n points, $\{v_1, \ldots, v_n\}$ are the rows of A and the second n points, $\{v_{n+1}, \ldots, v_{2n}\}$ are the columns of B. Again, for each coordinate $j = 1, \ldots, n$, sort in $O(n^2 \log n)$ time $\{v_i, i = 1, \ldots, 2n\}$ by coordinate j.

We will first compute $C^1[i,j] = \sum_{c:\ A[i,c] \leq B[c,j]} A[i,c]$, then $C^2[i,j] = \sum_{c:\ A[i,c] > B[c,j]} B[c,j]$ and then sum them in $O(n^2)$ time. We will only consider C^1 in the following as computing C^2 is analogous.

Recall, $r_j(v_i)$ is the position of v_i in the sorted list for coordinate j. Let $s \in [\log n, n]$ be a parameter to be determined later, corresponding to a bound on the size of a bucket. Define the n/s pairs of matrices (A_1, B_1), …, $(A_{n/s}, B_{n/s})$ as follows:

$$A_k[i,j] = \begin{cases} A[i,j] & \text{if } r_j(v_i) \in [ks, ks+s), \\ 0 & \text{otherwise.} \end{cases}$$

$$B_k[i,j] = \begin{cases} 1 & \text{if } r_j(v_{n+i}) \geq ks + s, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that this is the first place where the algorithm differs from that in Theorem 3.1.1. This concludes the **bucketting** step.

Now, multiply A_k with B_k^T, obtaining a matrix C_k for each bucket $k = 1, \ldots, n/s$. Then $C_k[i,j]$ equals the sum of entries $A[i,c]$ over coordinates c such that $r_c(v_i) \in [ks, ks+s)$, and $r_c(v_{n+j}) \geq ks+s$. In particular, for all such c we have $A[i,c] \leq B[c,j]$. Therefore, letting

$$C' = \sum_{k=1}^{n/s} C_k,$$

we have that $C'[i,j]$ is the sum of $A[i,c]$ over all c such that $A[i,c] \leq B[c,j]$, $\lfloor r_c(v_i)/s \rfloor < \lfloor r_c(v_{n+j})/s \rfloor$. This concludes the **bucket processing** step.

Suppose we compute a matrix E such that $E[i,j]$ is the sum of $A[i,c]$ over all coordinates c with $A[i,c] \leq B[c,j]$, and $\lfloor r_c(v_i)/s \rfloor = \lfloor r_c(v_{n+j})/s \rfloor$. Then, defining $C^1 := C' + E$, we will have the desired product matrix

$$C^1[i,j] = \sum_{c:\ A[i,c] \leq B[c,j]} A[i,c].$$

To compute E, we use the n sorted lists, as in Theorem 3.1.1. For each pair $(i,j) \in [n] \times [n]$, we look up v_{n+i}'s position p in the sorted list for coordinate j. By reading off the adjacent points less than v_{n+i} in this sorted list (*i.e.* the points at positions $p-1$, $p-2$, *etc.*), and stopping when we reach a point v_k such that $\lfloor r_j(v_k)/s \rfloor < \lfloor r_j(v_{n+i})/s \rfloor$, we obtain the list $v_{i_1}, \ldots, v_{i_\ell}$ of $\ell \leq s$ points such that $v_{i_x}[j] \leq v_{n+i}[j]$ and $\lfloor r_j(v_{n+i})/s \rfloor = \lfloor r_j(v_{i_x})/s \rfloor$. For each $x = 1, \ldots, \ell$, if $i_x \leq n$, we add $A[i_x, j]$ to $E[i_x, i]$. Assuming constant time lookups and constant time probes into a matrix, this entire process takes only $O(n^2 s)$ time. This concludes the **exhaustive search** step.

The runtime of the above procedure is $O(n^2 s + \frac{n}{s} n^\omega)$. Choosing $s = n^{\frac{\omega-1}{2}}$, the time bound becomes $O(n^{\frac{3+\omega}{2}})$. □

Results similar to Theorems 3.1.2 and 3.1.3 are also possible for the $(+, \min)$-product.

An application of the $(+, \min)$-product. In the following application, we show how the dominance and $(+, \min)$-products can be used to improve the runtime for solving a problem arising in computational economics. Suppose we have a set $C = \{1, \ldots, k\}$ of commodities, a set B of buyers and a set S of sellers. Let $|C| = k$, $|B| = |S| = n$. Each buyer $b_i \in B$ has a set of commodities $C_i \subseteq C$. Buyer i also has a maximum price p_{ij} that i is willing to pay for item j in C_i. Each seller $s_i \in S$ owns a set $S_i \in C$ of commodities that he is willing to sell, but for each commodity j he has a reserve price r_{ij} that needs to be met in order for the item to be exchanged.

Let us imagine that each buyer wants to do business with only one seller, and each seller wants to target a single buyer. Each of them wish to be matched up so that they maximize the number of transactions. Buyer i prefers seller j to seller j', if j has more cheap items than j' does, with respect to the values of i. Formally, the number of commodities k for which $r_{jk} \leq p_{ik}$ exceeds the number of commodities k for which $r_{j'k} \leq p_{ik}$.

More generally, the preferences of a buyer (or seller) may depend on more than just the number of items that can be exchanged. These preferences may depend on both the sums of maximum prices that the buyer has for the items, and on the sums of reserves that the seller has for them.

Ideally, each buyer wants to talk to the seller from whom he can obtain the cheapest large set of items. Unfortunately, this is not always possible for all buyers, even when the prices and reserves are all equal. This is evidenced by the following example: Buyer 1 wants to buy item 2, buyer 2 wants to buy items 1 and 2, seller 1 has item 1, seller 2 has items 1 and 2. When the preferences

are only according to the number of items that can be exchanged, buyer 1 will not be able to get any items.

In a realistic setting, we want to find a buyer-seller pairing so that there is no pair (b, s) for which buyer b is not paired with seller s, such that both b and s would benefit from breaking their matches and pairing among each other. This is the stable matching problem, for which optimal algorithms are known when the preferences are known (*e.g.*, Gale-Shapley [48] can be implemented to run in $O(n^2)$). However, for large k, the major bottleneck in our setting is that of computing the preference lists of the buyers and sellers.

The obvious approach gives an $O(kn^2)$ algorithm. Using the dominance product we can compute a matrix C so that C_{ij} is the number of items k for which $p_{ik} \geq r_{jk}$, *i.e.* the number of items that can be exchanged if i and j were paired. Then the rows and columns of C can be sorted in order to obtain a preference list for each buyer or seller. The entire procedure then would take $O(n\sqrt{kM(n,k)} + n^2 \log n)$ where $M(n,k)$ is the time required to multiply an $n \times k$ by a $k \times n$ matrix.

To do this, for each buyer i we create a k-dimensional vector $b_i = \{b_{i1}, \ldots, b_{ik}\}$ so that $b_{ij} = p_{ij}$ if $j \in C_i$, and $b_{ij} = -\infty$ if $j \notin C_i$. For each seller i we create a k-dimensional vector $s_i = \{s_{i1}, \ldots, s_{ik}\}$ so that $s_{ij} = r_{ij}$ if $j \in S_i$, and $s_{ij} = \infty$ if $j \notin S_i$. Let S be the matrix with rows s_i and let B be the matrix with columns b_i. Computing the dominance product matrix of S and B computes exactly the number of items ℓ which buyer i wants to buy, seller j wants to sell, and $p_{i\ell} \geq r_{j\ell}$.

By using the $(+, \min)$-product instead we can actually compute, for all buyers i and seller j, both

- the sum of prices of all items that each i and j can exchange (compute and negate the $(+, \min)$-product of $(-B)$ with $(-S)$), and
- the sum of reserve prices for these exchange items (compute the $(+, \min)$-product of S with B).

Doing this will allow us to compute any preference function on these two values, and will give a way to solve the more general buyer-seller stable matching problem fast. In particular, the preference lists for all buyers and sellers can be determined in $O(n\sqrt{kM(n,k)} + n^2 \log n + n^2 T)$, where T is the maximum time to compute the preference functions of the buyers/sellers, given the buyer price and seller reserve sums for a buyer-seller pair. For instance, if $k = n$ and $T = O(\log n)$, the runtime of finding a buyer-seller stable matching is $O(n^{\frac{3+\omega}{2}}) = O(n^{2.688})$.

3.3 Generalized Dominance Product

Let $(\oplus, \odot)$ be a matrix product of $n \times n$ matrices over some algebraic structure R containing an element $0 \in R$ such that for all $x \in R$, $0 \odot x = x \odot 0 = 0$ and $x \oplus 0 = 0 \oplus x = x$. The $(\oplus, \odot)$-dominance product is defined as follows.

Definition 3.3.1 *Let A and B be two $n \times n$ matrices. The entries of A and B are pairs of elements: $A[i,k] = (A^1[i,k], A^2[i,k])$ and $B[i,k] = (B^1[i,k], B^2[i,k])$ such that $A^1[i,k], B^1[i,k] \in R$ and $A^2[i,k], B^2[i,k] \in \mathbb{R}$. The* $(\oplus, \odot)$-dominance product *matrix C is given by*

$$C[i,j] = \bigoplus_{k \in [n], A^2[i,k] \leq B^2[k,j]} (A^1[i,k] \odot B^1[k,j]).$$

Clearly, an $O(t(n))$ time algorithm for the $(\oplus, \odot)$-dominance product of two $n \times n$ matrices can be used to compute the $(\oplus, \odot)$-product of two $n \times n$ matrices in $O(t(n))$ time by simply adding a second component equal to 1 to all matrix entries. We show that an efficient algorithm for the $(\oplus, \odot)$-product can be used to obtain an efficient algorithm for the $(\oplus, \odot)$-dominance product.

Theorem 3.3.1 *Suppose there is an $O(T(n))$ time algorithm for the $(\oplus, \odot)$-product of two $n \times n$ matrices. Then there is an $O(n^{1.5}\sqrt{T(n)})$ time algorithm for the corresponding $(\oplus, \odot)$-dominance product.*

In particular, when $T(n) = o(n^3)$, the algorithm from the theorem is also $o(n^3)$.

Proof of Theorem 3.3.1. The proof proceeds just as in Matousek's proof of Theorem 3.1.1. Let A and B be the matrices with entries $(A^1[i,j], A^2[i,j])$ and $(B^1[i,j], B^2[i,j])$. We sort for each k the k-th column of A and k-th row of B together in nondecreasing order of their second entry components. We bucket the sorted lists L_k into buckets of size roughly s, for a parameter s to be set later. For each bucket $b = 1, \ldots, \lceil n/s \rceil$, create matrices A_b and B_b with

$$A_b[i,j] = \begin{cases} A^1[i,j] & \text{if } A[i,j] \text{ is in bucket } b \text{ of } L_j, \\ 0 & \text{otherwise,} \end{cases}$$

$$B_b[i,j] = \begin{cases} B^1[i,j] & \text{if } B[i,j] \text{ is in some bucket } b' > b \text{ of } L_j, \\ 0 & \text{otherwise.} \end{cases}$$

We compute the $(\oplus, \odot)$-product of A_b and B_b over all b and compute the componentwise $\oplus$ of all the results to obtain a matrix C. This takes $O(nT(n)/s)$ time, provided $T(n) = \Omega(n^2)$. Now, for every entry $A[i,k]$, let b be the bucket of L_k in which it lies. Go through all roughly s other entries $B[k,j]$ in that bucket and if $A^2[i,k] \leq B^2[k,j]$, add (using the $\oplus$ operation) $(A^1[i,k] \odot B^1[k,j])$ to $C[i,j]$. This takes $O(n^2 s)$ time overall provided that the $\oplus$ and $\otimes$ operations take $O(1)$ time. To minimize the runtime we set $s = \sqrt{nT(n)/n^2}$ and we obtain an $O(n^{1.5}\sqrt{T(n)})$ time algorithm for the $(\oplus, \odot)$-dominance product. □

3.4 MaxMin Product and $(\min, \leq^r)$-Product

Recall the MaxMin product of two $n \times n$ matrices A and B over $\mathbb{R} \cup \{-\infty, \infty\}$ is defined as the matrix C such that $C[i,j] = \max_k \min\{A[i,k], B[k,j]\}$ for $i, j \in [n]$. The MaxMin product is a natural generalization of the Boolean matrix product to totally ordered sets of arbitrary size. In mathematics, this product is defined over the so called *subtropical* semiring $((\mathbb{R} \cup \{-\infty, \infty\}), \max, \min, -\infty, \infty)$. In computer science the MaxMin product has been studied mostly in the context of all pairs maximum bottleneck paths (e.g. [74]), a flow problem which we will discuss in Chapter 6. Besides its importance in flow problems, the MaxMin product is also an important operation in fuzzy logic, where it is known as the *composition of relations* ([35], pp.73).

As we showed in the previous chapter, there is an easy $O(n^3)$ algorithm for the MaxMin product of two $n \times n$ matrices. Obtaining a truly subcubic algorithm (*i.e.* running in $O(n^{3-\delta})$ time for constant $\delta > 0$) was given as an explicit goal by Shapira, Yuster and Zwick [83]. We resolve this open problem by showing how to reduce the MaxMin product of two $n \times n$ matrices to computing two $(\min, \leq^r)$-products of $n \times n$ matrices. We then give a truly subcubic algorithm for the $(\min, \leq^r)$-product of two matrices. Recall that the $(\min, \leq^r)$-product of two matrices A and B over $\mathbb{R} \cup \{-\infty, \infty\}$ is the matrix C' with $C'[i,j] = \min\{B[k,j] \mid A[i,k] \leq B[k,j]\}$ ($C'[i,j] = \infty$ if $A[i,k] > B[k,j] \forall k \in [n]$) for $i, j \in [n]$.

Lemma 3.4.1 *Suppose the* $(\min, \leq^r)$*-product of any two* $n \times n$ *matrices over* $\mathbb{R} \cup \{-\infty, \infty\}$ *can be computed in* $T(n)$ *time. Then the MaxMin product of any two* $n \times n$ *matrices over* $\mathbb{R} \cup \{-\infty, \infty\}$ *can be computed in* $O(T(n))$ *time.*

Proof. Suppose we are given $n \times n$ matrices A and B and we wish to compute their MaxMin product. We create the matrices $(-A)$ and $(-B)$ in $O(n^2)$ time. We then compute and negate the $(\min, \leq^r)$-product of $(-B)$ and $(-A)$ to produce a matrix A'. For every pair i, j, entry $A'[i, j]$ gives us the *maximum* $A[i, k]$ (over all k) such that $A[i, k] \leq B[k, j]$. Afterwards, we compute and negate the $(\min, \leq^r)$-product of $(-A)$ and $(-B)$ to produce a matrix B'. For every pair i, j, entry $B'[i, j]$ gives us the *maximum* $B[k, j]$ (over all k) such that $B[k, j] \leq A[i, k]$. This entire computation takes $O(n^2 + T(n))$ time. We can assume that $T(n) = \Omega(n^2)$ as one would at least have to read the input. Finally, we create a matrix C in $O(n^2)$ time with

$$C[i, j] = \max\{A'[i, j], B'[i, j]\}, \ \forall i, j \in [n].$$

□

We now show how one can compute the $(\min, \leq^r)$-product in truly sub-cubic time, reducing it to sparse dominance with bucketting.

Theorem 3.4.1 ($(\min, \leq^r)$-Product) *Given two* $n \times n$ *matrices* A *and* B*, their* $(\min, \leq^r)$*-product matrix* C *can be computed in* $O(n^{2+\frac{\omega}{3}})$ *time. Moreover, for each pair* i, j*, the algorithm returns a witness* k *such that* $A[i, k] \leq B[k, j] = C[i, j]$*.*

Proof. We begin with the **bucketting** phase. For every row i of matrix A, make a sorted list R_i of the entries in that row. Pick a parameter g. Partition the entries of each sorted list R_i into buckets, so that for every R_i there are $\lceil n/g \rceil$ buckets with at most g entries in each bucket. For every bucket value $b = 1, \ldots, \lceil n/g \rceil$, set up an instance (A, B, S_b) of the sparse dominance product from Theorem 3.1.3, where

$$S_b = \{(i, j) |\ A[i, j] \text{ is in bucket } b \text{ of } R_i\}.$$

This completes the bucketting phase. The **bucket processing** phase consists of actually computing the sparse dominance product and of returning some information for the exhaustive search phase. Notice that for every bucket value b, we have $|S_b| \leq ng$. By Theorem 3.1.3, all matrices $C_b = \texttt{SD}(A, B, S_b)$ can be computed in

$$O\left(\frac{n}{g} \cdot \sqrt{ng} \cdot n^{\frac{1+\omega}{2}}\right) = O\left(\frac{n^{2+\frac{\omega}{2}}}{\sqrt{g}}\right).$$

For every pair i, j, we determine the *largest* bucket $b_{i,j}$ in R_i for which there exists a k such that $A[i, k] \leq B[k, j]$. (This is obtained by taking the largest $b_{i,j}$ such that $C_{b_{i,j}}[i, j] \neq 0$. Note we can easily compute $b_{i,j}$ during the computation of the C_b.) This concludes the bucket processing phase.

In the **exhaustive search** phase, for every i, j we examine the entries in bucket $b_{i,j}$ of R_i to obtain the maximum $A[i, k]$ (and hence the corresponding k) such that $A[i, k] \leq B[k, j]$. Since there are at most g entries in a bucket, each pair i, j can be processed in $O(g)$ time. Therefore, this last step takes $O(n^2 g)$ time.

To pick a value for g that minimizes the runtime, we set $n^2 g = \frac{n^{2+\omega/2}}{\sqrt{g}}$, obtaining $g = n^{\frac{\omega}{3}}$. The running time is hence $O(n^{2+\frac{\omega}{3}})$. □

Plugging in the current best value for ω by Coppersmith and Winograd [28], the above time bound becomes $O(n^{2.79})$. If $\omega = 2$, then the above algorithm can run in $O(n^{2.67})$. Using Lemma 3.4.1 we also obtain the following corollary.

Corollary 3.4.1 *The MaxMin product C of two $n \times n$ matrices A and B can be computed in $O(n^{2+\frac{\omega}{3}}) = O(n^{2.79})$ time. Furthermore, for each pair $i, j \in [n]$, the algorithm returns a witness k such that $\min\{A[i,k], B[k,j]\} = C[i,j]$.*

3.5 Distance Product

Let A and B be two $n \times n$ matrices with entries in $\mathbb{R} \cup \{\infty\}$. Recall that the *distance product* $C := A \star B$ is an $n \times n$ matrix with $C[i,j] = \min_{k=1\ldots,n} A[i,k] + B[k,j]$. Clearly, C can be computed in $O(n^3)$ time in the addition-comparison model, as noted in the previous chapter. In fact, Kerr [57] showed that the distance product requires $\Omega(n^3)$ on a straight-line program using $+$ and min. However, Fredman [42] showed that the distance product of two square matrices of order n can be performed in $O(n^3(\log\log n/\log n)^{1/3})$ time. Following a sequence of improvements over Fredman's result, Chan [18] gave an $O(n^3(\log\log n)^3/\log^2 n)$ time algorithm for distance products.

Computing the distance product quickly has long been considered as the key to a truly sub-cubic APSP algorithm, since it is known that the time complexity of APSP is no worse than that of the distance product of two arbitrary $n \times n$ matrices. Practically all APSP algorithms with runtime of the form $O(n^\alpha)$ have, at their core, some form of distance product. Therefore, any improvement on the complexity of distance product is interesting.

Seidel [82] and Galil and Margalit [50] developed $\tilde{O}(n^\omega)$ algorithms for APSP in undirected unweighted graphs. However, for arbitrary edge weights, the best published algorithm for APSP and hence for the distance product is a recent $O(n^3 \log\log^3 n/\log^2 n)$ time algorithm by Chan [18]. When the edge weights are integers in $[-M, M]$, the problem is solvable in $\tilde{O}(Mn^\omega)$ by Shoshan and Zwick [84], and $\tilde{O}(M^{0.681}n^{2.575})$ by Zwick [106], respectively. Earlier, a series of papers in the 70's and 80's starting with Yuval [105] attempted to speed up APSP directly using fast matrix multiplication. Unfortunately, these works require a model that allows infinite-precision operations in constant time.

Here we show that for some nontrivial value of K, *the K most significant bits of the distance product $A \star B$* can be computed in sub-cubic time, again with no exponential dependence on edge weights. In previous work, Zwick [106] shows how to compute *approximate* distance products. Given any $\varepsilon > 0$, his algorithm computes distances d_{ij} such that the difference of d_{ij} and the exact value of the distance product entry is at most $O(\varepsilon)$. The running time of his algorithm is $O(\frac{W}{\varepsilon} \cdot n^\omega \log W)$. Unfortunately, guaranteeing that the distances are within ε of the right values, does not necessarily give any of the bits of the distances. Our strategy is to use the dominance matrix product.

Proposition 3.5.1 *Let $A, B \in (\mathbb{Z} \cup \{\infty\})^{n\times n}$. The k most significant bits of all entries in $A \star B$ can be determined in $O(2^k \cdot n^{\frac{3+\omega}{2}} \log n)$ time, assuming a comparison-based model.*

Proof. For a matrix M, let $M[i,:]$ be the ith row, and $M[:,j]$ be the jth column. For a constant K, define the set of vectors

$$M^L(K) := \{(M[i,1] - K, \ldots, M[i,n] - K) \mid i = 1, \ldots, n\}.$$

Also, define

$$M^R(K) := \{(-M[1,i], \ldots, -M[n,i]) \mid i = 1, \ldots, n\}.$$

Now consider the set of vectors $S(K) = A^L(K) \cup B^R(K)$. Using a dominance product computation on $S(K)$, one can obtain the matrix $C(K)$ defined by

$$C(K)[i,j] := \begin{cases} 0 & \text{if } \exists k \text{ s.t. } u_i[k] < v_j[k], u_i \in A^L(K), v_j \in B^R(K) \\ 1 & \text{otherwise} \end{cases}$$

Then for any i, j,

$$\min_k\{A[i,k] + B[k,j]\} \geq K \iff C(K)[i,j] = 1.$$

Let W be the smallest power of 2 larger than $\max_{ij}\{A[i,j]\} + \max_{ij}\{B[i,j]\}$. Then $C(\frac{W}{2})$ gives the most significant bit of each entry in $A \star B$. To obtain the second most significant bit, compute $C(\frac{W}{4})$ and $C(\frac{3W}{4})$. The second bit of $(A \star B)[i,j]$ is given by the expression:

$$(\neg C(W)[i,j] \wedge C(\tfrac{3W}{4})[i,j]) \vee (\neg C(\tfrac{W}{2})[i,j] \wedge C(\tfrac{W}{4})[i,j]).$$

In general, to recover the first k bits of $(A \star B)$, one computes $C(\cdot)$ for $O(2^k)$ values of K. In particular, to obtain the ℓ-th bits, compute

$$\bigvee_{s=0}^{2^{\ell-1}-1} [\neg C(W(1 - \frac{s}{2^{\ell-1}})) \wedge C(W(1 - \frac{s}{2^{\ell-1}} - \frac{1}{2^\ell}))].$$

To see this, notice that for a fixed s, if (for any $i, j \in [n]$)

$$W(1 - \frac{s}{2^{\ell-1}} - \frac{1}{2^\ell}) \leq \min_k A[i,k] + B[k,j] < W(1 - \frac{s}{2^{\ell-1}}),$$

then the ℓ-th bit of $\min_k A[i,k] + B[k,j]$ must be 1, and if the ℓ-th bit of $\min_k A[i,k] + B[k,j]$ is 1, then there must exist an s with the above property.

The values for $C(W(1 - \frac{s}{2^{\ell-1}}))$ for even s are needed for computing the $(\ell-1)$-st bits, hence to compute the ℓ-th bits, at most $2^{\ell-2} + 2^{\ell-1}$ dominance computations are necessary. To obtain the first k bits of the distance product, one needs only $O(2^k)$ dominance product computations. □

3.6 Parallel Algorithms

It is well known that on an EREW-PRAM one can compute matrix multiplication with $O(n^\omega)$ processors in $O(\log n)$ time. We show the following.

Theorem 3.6.1 *Let $(\oplus, \odot)$ be a matrix product for which we have obtained an $O(n^\alpha)$ algorithm in this chapter. Then this $(\oplus, \odot)$-product can be computed with $O(n^\alpha)$ processors on a EREW-PRAM in $O(\log n)$ time.*

Proof. Each of our algorithms followed three steps:

1. **Bucketting:** The preprocessing step in this chapter sorts $O(n^2)$ elements and then creates B pairs of new matrices for some value B. The sorting step can be done on an EREW-PRAM using Cole's mergesort algorithm with $O(n^2)$ processors and $O(\log n)$ time. Each pair of matrices creates a new problem instance so we have Bn^2 processors, each holding an entry of an instance.

2. **Bucket Processing:** For each fixed bucket instance $b \in [B]$, one takes the matrices A_b and B_b created by the bucketting step and multiplies them using a different matrix product $(\oplus', \otimes')$. We assume that this product can be computed using $O(n^\beta)$ processors and $O(\log n)$ time for some appropriate β ($\beta = \omega$ for Boolean matrix multiplication). Hence we have $O(Bn^\beta)$ processors solving this step in $O(\log n)$ time.

3. **Exhaustive Search:** We can add a processor computing each step of the exhaustive search in parallel. Each exhaustive search step in our algorithms is an element comparison which is independent from the other comparisons, and hence this can be done. After the comparisons, the best result from all comparisons is returned. This can be accomplished in $O(\log n)$ time and $O(n^3/B)$ processors by a binary search approach: add processors comparing disjoint pairs of results, then processors comparing disjoint pairs of the next results and so on; the number of new processors halves at each step and we get an asymptotic number of n^3/B (this is like a binary tree circuit).

The final number of processors is minimized when $B = n^{\frac{3-\beta}{2}}$ and the number of processors is $O(n^{\frac{3+\beta}{2}})$, which is the sequential running time that we would obtain for the $(\oplus, \odot)$-product using these techniques. □

Chapter 4

Finding Small Subgraphs

Finding cliques or other types of subgraphs in a larger graph is a classical problem in complexity theory and algorithmic combinatorics. Finding large cliques has many applications, for instance in computational biology, classification theory, coding theory, computer vision, economics, information retrieval, signal transmission theory etc. For detailed example applications, see [14].

Finding a maximum clique is NP-Hard, and also hard to approximate [51]. This problem is also conjectured to be *not* fixed parameter tractable [33]. The problem of finding (induced) subgraphs on k vertices in an n-vertex graph has been studied extensively (see, e.g., [4, 5, 22, 37, 58, 68, 73, 103]). All known algorithms for finding an induced subgraph on k vertices have running time $n^{\Theta(k)}$. Many of these algorithms use fast matrix multiplication to obtain improved exponents.

In this chapter (as in the rest of the thesis) we only consider polynomial time solvable problems. We give efficient algorithms for instances of the following general problem:

Definition 4.0.1 *Let H be a fixed graph of constant size k. The* H-subgraph problem *on instance G is defined as follows: Given an input graph G, find a subgraph H' of G isomorphic to H.*

If H' is to be an induced subgraph of G, we refer to the problem as the induced H-subgraph problem*. If G has real weights on its vertices or edges, we further require that H' is an induced subgraph of G of* maximum weight sum *over all such induced subgraphs of G isomorphic to H. The problem is then called the* maximum H-subgraph problem*.*

We note that finding a maximum k-clique is at least as hard as finding a maximum induced H-subgraph, but the opposite is not necessarily true, as we show in the section on finding k-cliques.

Lemma 4.0.1 *Suppose one can find a maximum weight k-clique in an n node graph in $O(T(n))$ time. Then, for any fixed graph H on k nodes, one can find a maximum induced H-subgraph of an n node graph in $O(n^2 + T(n))$ expected time.*

Proof. Let $G = (V, E)$ be the given graph. Let G have weights $w_V : V \to \mathbb{R}$ and $w_E : E \to \mathbb{R}$. Create k sets $S_1, \ldots, S_k$ initially empty. For every node v of the input graph G, with probability $p_i = 1/k$ place v in S_i. In expectation, we add each node to 1 set S_i. We will create a graph G' based on G and H. G' will have $O(n)$ nodes in expectation.

We will first define the edges of G'. Let $H = (V_H, E_H)$ have nodes $h_1, \ldots, h_k$. Consider every pair of indices $i, j \in [k]$, $i \neq j$. For every two nodes $u \in S_i$, $v \in S_j$, place an edge between u and v in G' if $(h_i, h_j) \in E_H$ and $(u, v) \in E$, or if $(h_i, h_j) \notin E_H$, $(u, v) \notin E$ and $u \neq v$. This transformation

takes expected $O(n^2)$ time as k is fixed and for every node of G there are only a constant number expected copies in G'.

Consider a subgraph $H' = (V_{H'}, E_{H'})$ of G isomorphic to H so that the isomorphism maps node h'_i of H' to node h_i of H. Then with probability at least $1/k^k$ for every $i = 1, \ldots, k$, node h'_i of H' is in set S_i. Furthermore, the nodes corresponding to the h'_i in G' form a clique as for every $i, j \in [k]$, $i \neq j$, $(h'_i, h'_j) \in E_{H'}$ if and only if $(h_i, h_j) \in E_H$ as H and H' are isomorphic.

Now suppose that a subgraph H' of G' is a clique. Since no two nodes in the same set S_i are linked by an edge, the nodes $h'_1, \ldots, h'_k$ are in different sets S_i. Suppose for $i = 1, \ldots, k$, node $h'_i \in S_i$. As H' is a clique, it must be the case that all h'_i correspond to distinct nodes of G. Furthermore, because of the construction of G', since (h'_i, h'_j) is an edge in G', then (h'_i, h'_j) is an edge in G if and only if (h_i, h_j) is an edge in H. Thus H and H' must be isomorphic and H' is an induced H-subgraph in G.

We now define the weights of G'. We let every node u of G' inherit its weight $w_V(u)$ from G. If (u, v) is an edge in G' with $u \in S_i$ and $v \in S_j$ such that $(h_i, h_j) \in E_H$, then let (u, v) inherit its weight $w_E(u, v)$ from G. Otherwise, set its weight to 0. In this way, k-cliques of G' have the same weight sum as their corresponding H-subgraphs in G. □

4.1 Unweighted Subgraphs

We first investigate the H-subgraph problem in the unweighted setting.

4.1.1 Cycles

We consider the H-subgraph problem when H is C_k, a cycle on k nodes. We give some reductions between the directed and undirected versions of the problem obtaining Figure 4.1.1.

A major result on the C_k-subgraph problem was proven by Yuster and Zwick who showed that an undirected even cycle in an n node graph can be found in $O(n^2)$ time:

Theorem 4.1.1 (Yuster and Zwick [102]) *Let $k > 1$ be a constant. Given an undirected graph G on n nodes, a (not necessarily induced) cycle subgraph C_{2k} on $2k$ nodes, if it exists, can be found in $O(n^2)$ time.*

Interestingly, for fixed k, the best known runtime bound for finding an odd C_k in an undirected graph, or an even directed C_k in a directed graph is $O(n^\omega)$. Hence the problems of finding even directed cycles, and finding odd undirected cycles seem more difficult than the problem of finding an even undirected cycle. Our reductions in this section give some intuition about why this might be the case.

We will denote an undirected cycle on k nodes by C_k and a directed cycle on k nodes by $\overrightarrow{C_k}$. We first show that finding a directed k-cycle is at least as hard as finding an undirected k-cycle. We then show that finding an odd undirected k-cycle is at least as hard as finding a directed k-cycle. Finally we note that finding a k-cycle is at least as hard as finding a $k-1$-cycle. Our results are shown pictorially in Figure 4.1.1.

Theorem 4.1.2 *Let k be fixed. Suppose that there is an algorithm which given a directed n node graph G finds a $\overrightarrow{C_k}$ subgraph of G (if it exists) in $O(T(n))$ time. Then there is an $O(n^2 + T(n))$ expected time algorithm which given an undirected n node graph G', finds a C_k subgraph of G', if it exists.*

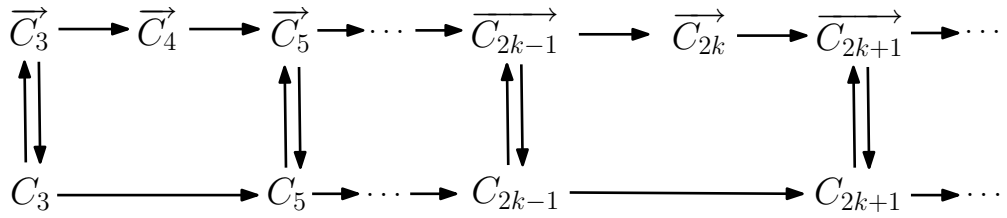

Figure 4.1: An arrow from cycle C of length k to cycle C' means that there is an expected $O(f(k)n^2)$ time reduction from the C-subgraph problem on n node graphs to the C'-subgraph problem on $O(n)$ node graphs for some computable function f. Theorem 4.1.2 proves $C_k \to \vec{C}_k$. Theorem 4.1.3 proves $\vec{C}_{2k+1} \to C_{2k+1}$. Theorem 4.1.4 proves $\vec{C}_k \to \vec{C}_{k+1}$. The arrows are composable, and we also obtain $C_{2k-1} \to C_{2k+1}$.

Proof. Given an undirected graph G' on n nodes in which we want to find a C_k, we will construct in expected $O(n^2)$ time an $O(n)$ node graph G such that finding a $\overrightarrow{C_k}$ in G corresponds to finding a C_k in G'.

Suppose we have k colors labeled $1, \ldots, k$. Then, for every node v of G', color v with color i with probability $1/k$. If a node does not get any colors, flip its coins again. We expect that after $O(1)$ coin flips per node each node will have at least one color, and an expected $O(1)$ colors.

For every node v of G' and color i with which v is colored, create a node v_i in G. For each $i = 1, \ldots, k$, create a directed edge (u_i, v_{i+1}) in G, if (u, v) is an edge of G' and u is colored i and v is colored $i + 1$. Here the indices are taken $\mod 2k$.

Any directed k-cycle of G, $v_1^1 \to v_2^2 \to \ldots \to v_k^k \to v_1^1$ corresponds to an undirected k-cycle of G', $v^1 \to v^2 \to \ldots \to v^k \to v^1$.

Suppose $v^1 \to v^2 \to \ldots \to v^k \to v^1$ is an undirected k-cycle of G'. Then with probability $1/k^k$, for each $i = 1, \ldots, k$, v^i received color i. Hence with probability $1/k^k$, $v_1^1 \to v_2^2 \to \ldots \to v_k^k \to v_1^1$ is a directed cycle in G'. In expectation we need to repeat this reduction $k^k = O(1)$ times in order for the directed cycle to appear in G'. □

If we consider only odd cycles, the converse of Theorem 4.1.2 also holds:

Theorem 4.1.3 *Let $k > 0$ be fixed. Suppose that there is an algorithm that given an undirected n node graph G finds a C_{2k+1} subgraph of G (if it exists) in $O(T(n))$ time. Then there is an $O(n^2 + T(n))$ time algorithm that given a directed n node graph G', finds a $\overrightarrow{C_{2k+1}}$ subgraph of G', if it exists.*

Proof. We proceed very similarly to Theorem 4.1.2. Given a directed graph G' on n nodes in which we want to find a $\overrightarrow{C_{2k+1}}$, we will construct in expected $O(n^2)$ time an $O(n)$ node graph G such that finding a C_{2k+1} in G corresponds to finding a $\overrightarrow{C_{2k+1}}$ in G'.

Suppose we have $2k + 1$ colors labeled $1, \ldots, (2k + 1)$. Then, for every node v of G', color v with color i with probability $1/(2k + 1)$. If a node does not get any colors, flip its coins again. We expect that after $O(1)$ coin flips per node each node will have at least one color, and an expected $O(1)$ colors.

For every node v of G' and color i with which v is colored, create a node v_i in G. For each $i = 1, \ldots, (2k + 1)$, create an undirected edge (u_i, v_{i+1}) in G, if (u, v) is a directed edge of G' and u is colored i and v is colored $i + 1$. Here the indices are taken $\mod (2k + 1)$.

First consider a $(2k+1)$-cycle of G', say $v^1 \to v^2 \to \ldots \to v^{2k+1} \to v^1$. With probability $1/(2k+1)^{2k+1}$ this cycle appears in G as the $(2k+1)$-cycle $v_1^1 \to v_1^2 \to \ldots \to v_{2k+1}^{2k+1} \to v_1^1$.

Now, consider any $(2k+1)$-cycle C of G. Suppose that there is some $i \in [2k+1]$ such that C does not contain any edge $(u_i, v_{i+1 \mod (2k+1)})$. Then C must be even. Hence the only possible undirected $(2k+1)$-cycles in G are of the form $v_1^1 \to v_1^2 \to \ldots \to v_{2k+1}^{2k+1} \to v_1^1$, and they correspond to directed cycles $v^1 \to v^2 \to \ldots \to v^{2k+1} \to v^1$ of G'.

We expect to repeat the reduction $(2k)^{2k} = O(1)$ times for a $(2k+1)$ cycle to appear in G if a $\overrightarrow{C_{2k+1}}$ in G' exists. The reduction itself takes $O(n^2)$ time. □

Finally, we show that as k grows the time complexity of finding a C_k and a $\overrightarrow{C_k}$ grows.

Theorem 4.1.4 *Let $k > 2$ be fixed. Suppose that there is an algorithm that given a directed n node graph G finds a C_{k+1} subgraph of G (if it exists) in $O(T(n))$ time. Then there is an $O(n^2 + T(n))$ time algorithm which given a directed n node graph G', finds a $\overrightarrow{C_k}$ subgraph of G', if it exists.*

Proof. We proceed very similarly to Theorems 4.1.3 and 4.1.2. Given a directed graph G' on n nodes in which we want to find a $\overrightarrow{C_k}$, we will construct in expected $O(n^2)$ time an $O(n)$ node graph G such that finding a C_{k+1} in G corresponds to finding a $\overrightarrow{C_k}$ in G'.

Suppose we have k colors labeled $1, \ldots, k$. Then, for every node v of G', color v with color i with probability $1/(k)$. If a node does not get any colors, flip its coins again. We expect that after $O(1)$ coin flips per node each node will have at least one color, and an expected $O(1)$ colors.

For every node v of G' and color i with which v is colored, create a node v_i in G. For each $i = 1, \ldots, k-1$, create an undirected edge (u_i, v_{i+1}) in G, if (u, v) is a directed edge of G' and u is colored i and v is colored $i+1$. Here the indices are *not* taken $\mod k$, so that we do not have edges (u_k, v_1).

For every node v_1 of G, add a new node v_1' to G and an edge (v_1', v_1). For every two nodes u_k and v_1' of G such that (u, v) is a directed edge of G', add an edge (u_k, v_1') to G.

Clearly, any directed $(k+1)$-cycle of G has the form $v_1^1 \to v_2^2 \to \ldots \to v_k^k \to v_1'^1 \to v_1^1$ and hence corresponds to a k-cycle $v^1 \to v^2 \to \ldots \to v^k \to v^1$ of G'.

Conversely, a k-cycle of G' appears in G with probability $1/k^k$. Hence we expect that after $O(1)$ repetitions of this reduction that if G' has a k-cycle, then G will contain a $(k+1)$-cycle. The reduction takes expected $O(n^2)$ time. □

It is likely that our above two results can be derandomized, perhaps using perfect hash families, as in [4]. The dependence on k in our algorithms can also probably be improved.

4.1.2 k-Clique and Related Problems

The k-clique problem is one of the fundamental problems in computer science. It is one of Karp's original NP-complete problems [56]. In parametrized complexity k-clique plays a central role: one of the major problems in this area is to resolve whether $W[1] = FPT$, and k-clique is $W[1]$-complete [34]. Furthermore, results by Chen *et al.* [20] imply that if there is an $n^{o(k)}$ algorithm for k-clique [1], then many of the classical NP-complete problems are solvable in subexponential time.

The naive algorithm for k-clique runs in $O(n^k)$ time by examining all k-tuples of vertices. In 1978, Itai and Rodeh [54] showed that a 3-clique (triangle) in an n-node graph can be found in the same time as $n \times n$ Boolean matrix multiplication. Nowadays this means that there is an $O(n^{2.376})$

[1] here k is assumed to be any unbounded function of n

algorithm [28] for 3-clique; we denote this exponent by ω. In 1984 ([64], p. 46), László Lovász asked if there is an $O(n^{0.01k})$ algorithm for general k-clique. He noted that Fan Chung and Richard Karp had used Strassen's fast matrix multiplication [85] to obtain better algorithms for 100-clique. In the next year, Nešetřil and Poljak [69] showed how to generalize Itai and Rodeh's [54] reduction to obtain an $O(n^{\omega k})$ time algorithm for $3k$-clique. Since that result very little progress has been made. In 2004 Eisenbrand and Grandoni [37] used rectangular matrix multiplication to obtain improved running times for some values of k, *e.g.* for $k \geq 5$ such that $k = 2 \mod 3$.

Although these algebraic matrix product approaches yield good theoretical time bounds, the known fast matrix multiplication algorithms that run in $O(n^{3-\varepsilon})$ time for $\varepsilon > 0$ ([85, 28]) are very inefficient in practice. Another issue is that all of the above mentioned algorithms for k-clique use $n^{\Omega(k)}$ space. In fact, in the same 1984 article ([64], p.46), László Babai asked if there exists an $o(n^k)$ algorithm for k-clique using $O(n^c)$ space for some constant c independent of k. This question has until now remained unanswered. In fact, the problem of designing better time and space efficient k-clique algorithms has been recently reraised by Woeginger [101].

Combinatorial, nonalgebraic algorithms based on preprocessing and table look-up offer less impressive asymptotic improvements but seem to be much more practical than algorithms based on algebraic fast matrix multiplication (see Chapter 7 for combinatorial algorithms in sparse graphs). Using a "Four Russians" type approach for combinatorial matrix multiplication [6, 79, 7, 17] one can obtain an $O(n^k/(k \log^2 n))$ time algorithm for k-clique that hides no enormous constant factors in its runtime. Nevertheless, even this algorithm is space-inefficient – it requires $\Omega(n^{2k/3})$ space.

In this section we answer Babai's question in the affirmative. We give a combinatorial algorithm for k-clique that runs in $O(n^k/\log^{k-1} n)$ time for every fixed $k \geq 3$ and uses $O(kn^{\varepsilon})$ space, for any constant $\varepsilon > 0$ independent of k. This algorithm is not only space-efficient but also: (a) it beats the runtimes of the current best combinatorial algorithms for finding k-clique for $k > 3$, and (b) it can be adapted easily to obtain an algorithm for maximum node-weighted k-clique with *identical* runtime. (We will give an algebraic algorithm for the node-weighted k-clique problem in the next section. Also note that Czumaj and Lingas [30] have shown that maximum weighted $3k$-clique can be found in $O(n^{\omega k+\varepsilon})$ for all $\varepsilon > 0$ but we cannot apply their method here because of the extra $O(n^{\varepsilon})$ factor.)

In the second part of the section we consider finding k-node subgraphs that are almost cliques: *induced* subgraphs that are cliques missing one edge. This scenario is a generalization of the well-studied recognition problem for P_3-free or diamond-free graphs. A folklore result shows that an induced P_3 in a graph can be found in $O(n^2)$ time. Kloks, Kratsch and Müller [58] show that an induced diamond in a graph can be found in $O(n^{\omega} + m^{3/2})$ time and Eisenbrand and Grandoni [37] improve this running time to $O(m^{3/2})$ by a combinatorial algorithm. We generalize these results to show that a $K_k - e$ in a graph can be found (if it exists) combinatorially in $O(m^{(k-1)/2})$ time and $O(m+n)$ space. This running time is better than any known combinatorial algorithm for k-clique.

Algorithm for k-Clique. We begin with a small-space combinatorial algorithm which given a node-weighted graph finds a k-clique of maximum weight sum. The basic idea used in the algorithm is to reduce the problem to finding maximum node-weighted $(k-1)$-cliques in many small $O(\log n)$ size subgraphs, and then attempt to complete these cliques by adding an extra node. To use small space, our algorithm proceeds in iterations so that each new iteration reuses the space used by the previous ones.

Theorem 4.1.5 *Let $\varepsilon > 0$ and $k \geq 3$. Let $g(k) = (2(k-1))^{k-1}$. Let $G = (V, E)$ be a given graph.*

There is an algorithm for k-clique that runs in $O\left(g(k) \cdot \frac{n^k}{(\varepsilon \log n)^{k-1}}\right)$ time and uses $O(kn^\varepsilon)$ space.

Proof. Let $G = (V, E)$, $\varepsilon > 0$ and k be given. Let $\varepsilon' = \varepsilon/(2(k-1))$.

Partition the nodes into $n/(\varepsilon' \log n)$ parts of $\varepsilon' \log n$ nodes each. The partition is chosen by grouping consecutive nodes according to the order of the columns in the adjacency matrix. This ensures that any k chunks of a row corresponding to k of the $(\varepsilon' \log n)$-size parts can be concatenated in $O(k)$ time, assuming constant time look-ups and an $O(\log n)$-word RAM.[2]

We will process all $(k-1)$-tuples of parts as follows. Fix a $(k-1)$-tuple of parts. Obtain the union U of these parts, and create a look-up table T_U with $(\varepsilon'(k-1)\log n)$-bit keys. For all $2^{\varepsilon'(k-1)\log n}$ subsets of U determine whether the corresponding induced subgraph of G contains a $(k-1)$ clique. Store the result for each subset (a k-clique, if found) in T_U, with the binary $(\varepsilon'(k-1)\log n)$-length vector representing the subset as the key. When a new union is processed, the look-up table information is overwritten.

Now, for every node $v \in V$, concatenate the portions of the neighborhood vector for v corresponding to U in $O(k)$ time to obtain a $(\varepsilon'(k-1)\log n)$-bit key. Look up in T_U using this key whether the induced neighborhood of v in U contains a $(k-1)$-clique. If this is so, return the union of the clique and v.

Creating all tables T_U takes

$$O((n/(\varepsilon' \log n))^{k-1} \cdot n^{\varepsilon'(k-1)} \cdot (\varepsilon'(k-1)\log n)^{k-1}) = O(n^{(1+\varepsilon')(k-1)} \cdot (k-1)^{k-1})$$

time, and $O(k \log n 2^{\varepsilon'(k-1)\log n})$ space over all. The entire procedure (assuming $O(1)$ time look-ups in T_U) takes $O(n \cdot (n/(\varepsilon' \log n))^{k-1}))$ time, *i.e.*

$$O(n^{(1+\varepsilon')(k-1)} \cdot (k-1)^{k-1} + n^k/(\varepsilon' \log n)^{k-1})$$

time overall. Since we set $\varepsilon' = \varepsilon/(2(k-1))$, the runtime becomes $O((2(k-1))^{k-1} \cdot n^k/(\varepsilon \log n)^{k-1})$. The space usage becomes asymptotically

$$k \log n 2^{\varepsilon'(k-1)\log n} = kn^{\varepsilon/2} \log n = O(kn^\varepsilon).$$

□

One can modify the algorithm so that a maximum weight clique can be found: for every $(\varepsilon'(k-1)\log n)$-node subgraph of a union of $(k-1)$-tuples of node partitions, compute the the maximum weight $(k-1)$ clique and store that clique in the look-up table for the union. Since the algorithm goes over all possible choices for a k-th node, the maximum weight clique can be returned.

Corollary 4.1.1 *Let $\varepsilon > 0$ and $k \geq 3$. Let $g(k) = (2(k-1))^{k-1}$. Let $G = (V, E)$ be a graph with arbitrary real weights on its nodes. There is an algorithm for maximum node-weighted k-clique that runs in $O\left(g(k) \cdot \frac{n^k}{(\varepsilon \log n)^{k-1}}\right)$ time and uses $O(kn^\varepsilon)$ space.*

We can use the algorithm of Theorem 4.1.5 to obtain an algorithm with a running time depending on the number of edges in the graph. We use an idea used in many other results (*e.g.* [5]) – either the clique has all high degree vertices, or it has a low degree vertex. In both cases we can reduce the problem to finding a clique in a strictly smaller subgraph.

[2]The same can be accomplished on a pointer machine using some tricks.

Theorem 4.1.6 *Let $k \geq 5$ and $\varepsilon > 0$. Let $g(k) = (2(k-1))^{k-1}$. There is an algorithm for k-clique that runs in $O(km^{\varepsilon/2})$ space and $O\left(g(k) \cdot \frac{m^{\frac{k}{2}}}{(\varepsilon \log m)^{(k-2-\frac{1}{k-1})}}\right)$ time on graphs with m edges.*

Proof. Let D be a parameter. Consider the set S of all nodes of degree at least D. There are at most m/D such nodes. Use the algorithm of Theorem 4.1.5 to find a k-clique contained in S if one exists in $O(g(k)(m/D)^k/(\varepsilon \log(m/D))^{k-1})$ time. If no k-clique is found in S, any k-clique in the graph must contain a node of degree $< D$. Go through all edges (u, v) incident on low degree nodes, and look for a $(k-2)$-clique in the intersection of the neighborhoods of u and v, again using Theorem 4.1.5. This takes time $O(mg(k-2)D^{k-2}/(\varepsilon \log D)^{k-3})$.

To minimize the final running time of the clique algorithm, we set

$$mD^{k-2}/(\varepsilon \log D)^{k-3} = \Theta((m/D)^k/(\varepsilon \log(m/D))^{k-1}),$$

and hence $D = \sqrt{m}/(\varepsilon \log m)^{\frac{1}{k-1}}$ suffices. When we plug back in, the running time becomes $O(g(k)m^{k/2}/(\varepsilon \log m)^{(k-2-\frac{1}{k-1})})$. The space usage is $O(k(m/D)^{\varepsilon})$ in the first part of the procedure, and $O((k-2)D^{\varepsilon})$ in the second part. The space usage in the first part dominates and is $O(km^{\varepsilon/2})$. □

Finding an induced $K_k - e$. In graph theory, we use $K_k - e$ to denote a k-clique missing one edge. A $K_3 - e$ is also called a P_3 (a path on 3 nodes), and a $K_4 - e$ is a diamond. Diamond-free graphs, graphs containing no induced $K_4 - e$, are well studied in graph theory, in particular in connection to the strong perfect graph theorem [23] which was shown by Tucker [91] to hold for these graphs. In terms of the computational complexity of recognizing diamond-free graphs, Kloks *et al.* [58] showed that an induced diamond in a graph can be found in $O(n^{\omega} + m^{3/2})$ time. Eisenbrand and Grandoni [37] improved this running time to $O(m^{3/2})$ combinatorially. We generalize this result by showing that finding an induced $K_k - e$ in a given graph, or recognizing $(K_k - e)$-free graphs can be done combinatorially in $O(m^{(k-1)/2})$ time and $O(m + n)$ space on graphs with m edges. Our proofs are really short and simple. We include this result to show that removing even just one edge from a k-clique seems to make finding an induced subgraph easier: our algorithm is faster than any known combinatorial algorithm for k-clique.

We begin by a lemma that gives a linear time algorithm for finding a P_3. To our knowledge, the previous best algorithm had a running time of $O(n^2)$. The following lemma also gives an example of a case for which finding a k-clique seems harder than finding a not necessarily induced H-subgraph on k nodes. Recall that in the beginning of the chapter we showed that finding a k-clique is at least as hard as finding an H-subgraph on k nodes.

Lemma 4.1.1 *One can find an induced P_3 or determine that the graph is P_3-free in $O(m + n)$ time.*

Proof. We will label nodes by their first processed neighbor. Give an ordering to the nodes. Begin with node 1. When processing node i, if i has not been labeled yet, label it i and go through all its neighbors. Label all unlabeled neighbors i. If some neighbor x is labeled j, then j, x, i form a P_3.

Suppose node i is already labeled with j. Then, if $deg(i) \neq deg(j)$, find a node x that is in the difference of their neighborhoods and return x, i and j as a P_3. Otherwise, suppose $deg(i) = deg(j)$. Check whether all the neighbors of i are labeled j. If so, move on to node $i+1$. On the other hand, if some neighbor x of i is labeled with $j' \neq j$, then

- if $j' > j$, then when j' was processed, there was no edge between x and j and hence j, i, x form a P_3;
- if $j > j'$, then when j was processed, there was no edge between i and j' and hence i, x, j' form a P_3.

If some neighbor x does not have a label, then x does not have an edge to j and hence j, i, x form a P_3. If by the end of the procedure no P_3 was returned, then the graph is a disjoint union of cliques and hence contains no induced P_3. This is because every node i has the same neighbors as its first encountered neighbor. This entire labeling procedure takes $O(m + n)$ time. □

Because finding induced P_3 and $K_4 - e$ can be done efficiently, we can give an inductive procedure to find induced $K_k - e$ for general $k \geq 3$.

Theorem 4.1.7 *An induced $K_k - e$ in a graph with m edges can be found in $O(m^{(k-1)/2})$ time, for any $k \geq 3$.*

Proof. If $k = 3$, then use the algorithm from Lemma 4.1.1 to find a $K_3 - e$ in $O(m)$ time. If $k = 4$, use Eisenbrand and Grandoni's algorithm to find a $K_4 - e$ in $O(m^{3/2})$ time. In both cases this is $O(m^{(k-1)/2})$.

Suppose $k > 4$. Let $i = k \mod 2$, $i \in \{0, 1\}$. Consider all $(k - 4 + i)$-cliques in the graph. There are at most $m^{(k-4+i)/2}$ such cliques as each such clique can be represented as a $(k-4+i)/2$-matching. For each $(k-4+i)$-clique, in $O(km)$ time find the intersection N of the neighborhoods of the nodes in the clique. Then, in $O(m^{(3-i)/2})$ time find an induced $K_{4-i} - e$ in N, or determine that it does not exist. (Note, $K_{4-i} - e$ is either $K_3 - e$ or $K_4 - e$.) The overall running time becomes $O(km^{(k-2+i)/2} + m^{(k-4+i+3-i)/2}) = O(km^{(k-1)/2})$. When k is even, the running time is $O(m^{(k-1)/2})$. □

The algorithm from Theorem 4.1.7 runs in linear space because both Eisenbrand and Grandoni's algorithm for finding diamonds and our algorithm from Lemma 4.1.1 use linear space, and each of the separate recursive calls to these algorithms are on smaller graphs.

4.2 Weighted Subgraphs

In this section we present algorithms for solving the maximum (induced) H-subgraph problem in a real vertex-weighted or edge-weighted graph. Some of our algorithms are based, in part, on fast matrix multiplication, just as in previous chapters. In several cases, our algorithms use fast *rectangular* matrix multiplication algorithms of Coppersmith [26] and Huang and Pan [53]. All of our results are applicable to both directed and undirected graphs. Likewise, all of our results on the maximum H-subgraph problem hold for the analogous minimum H-subgraph problem. In most of our algorithms, we use the *addition-comparison* model for handling real numbers. That is, real numbers are only allowed to be compared or added. In particular, these algorithms are strongly polynomial.

Prior to this work, for h-node graphs H, the only known algorithm for maximum H-subgraph in the vertex-weighted case (moreover, the All-Pairs version of the problem) was the naïve $O(n^h)$ algorithm. In general, reductions to fast matrix multiplication tend to fail miserably in the case of real-weighted graph problems. The most prominent example of this is the famous All-Pairs Shortest Paths (APSP) problem, for which no truly subcubic algorithm is known.

For simplicity, we present our results only for the case when $|V(H)| = 0 \mod 3$. The results are easily extendable for general H. We use the following folklore lemma.

Lemma 4.2.1 (Folklore) *If a maximum weight triangle in an edge- or node-weighted undirected graph can be found in $T(n)$ time, then for any $3h$-node directed graph H, a maximum H-subgraph of an edge- or node-weighted graph can be found in $O(T(n^h))$ time.*

Proof. Let $H = (V(H), E(H))$ be a constant size $3h$-node subgraph with $V(H) = \{x_1, \ldots, x_{3h}\}$. Let $G = (V, E)$ be given graph with edge weights $w_e : E \to \mathbb{R}$ and node weights $w_v : V \to \mathbb{R}$; $|V| = n$. We construct a new, undirected, graph G' on $\leq n^h$ nodes with edge weights $w'_e : E \to \mathbb{R}$ and node weights $w'_v : V \to \mathbb{R}$ as follows. G' will be a tripartite graph with partitions P_0, P_1 and P_2.

For every ordered h-tuple of distinct nodes $(v_1, \ldots, v_h)$ of G and for each partition P_i, create a node of G' in P_i if the following condition holds: for every $k = 1, \ldots, h$, $\ell = 1, \ldots, h$, $(x_{ih+k}, x_{ih+\ell}) \in E(H)$ iff $(v_k, v_\ell) \in E$.

G' has $\leq 3n^h$ nodes. Let the weight $w'_v(t)$ of a node $t = (v_1, \ldots, v_h)$ of G' be defined as

$$w'_v(t) := \sum_{i=1}^{h} w_v(v_h) + \sum_{(v_k, v_\ell) \in E} w_e(v_k, v_\ell).$$

We now define the edge relation for G'. We add an edge between two nodes $t = (v_1, \ldots, v_h)$ and $t' = (v'_1, \ldots, v'_h)$ of G' if the following is true:

- $t \in P_i$, $t' \in P_j$ and $i \neq j$;
- for every $k = 1, \ldots, h$, $\ell = 1, \ldots, h$: $v_k \neq v'_\ell$;
- for every $k = 1, \ldots, h$, $\ell = 1, \ldots, h$: $(x_{ih+k}, x_{jh+\ell}) \in E(H)$ if and only if $(v_k, v'_\ell) \in E$ and $(x_{jh+k}, x_{ih+\ell}) \in E(H)$ if and only if $(v'_k, v_\ell) \in E$;

The weight of an edge in G' between $t = (v_1, \ldots, v_h)$ and $t' = (v'_1, \ldots, v'_h)$ is given by

$$w'_e(t, t') := \left[\sum_{(v_k, v'_\ell) \in E} w_e((v_k, v'_\ell)) \right] + \left[\sum_{(v'_k, v_\ell) \in E} w_e((v'_k, v_\ell)) \right].$$

G' has the property that its triangles correspond exactly to the H-subgraphs of G. In particular, a maximum weight triangle in G' is a maximum weight H-subgraph of G. Hence, if a maximum weight triangle can be found in $T(n)$ time in an undirected graph on n nodes. Then a maximum H-subgraph of an n-node graph can be found in $O(T(3n^h))$ time. □

Because of the previous lemma, in the remainder of the chapter we focus on finding maximum weight triangles. The running times for maximum H-subgraph easily follow.

Another folklore result states that finding a maximum edge-weighted triangle is as hard as the general case of finding a maximum triangle in a node- and edge-weighted graph.

Lemma 4.2.2 (Folklore) *One can reduce the problem of finding a maximum triangle in a general weighted graph to the case in which the graph has only edge weights.*

Proof. Suppose $G = (V, E, w_e, w_v)$ is given. We simply transform it to $G' = (V, E, w)$ where $w : E \to \mathbb{R}$ is given by $w(u, v) = w_e(u, v) + (w_v(u) + w_v(v))/2$. □

We obtain truly subcubic algorithms for finding a maximum weight triangle in a node-weighted graph. The edge-weighted case of triangle finding seems genuinely harder than the node-weighted case. We are only able to relate its time complexity to the time to compute the distance product of two matrices.

4.3 Maximum Triangles in Edge Weighted Graphs

We first show how to reduce the maximum H-subgraph problem in edge-weighted graphs to the problem of computing a distance product. In particular, we prove the following.

Theorem 4.3.1 *If $G = (V, E)$ is a graph with n vertices, and $w : E \to \mathbb{R}$ is a weight function, then a maximum weight triangle of G (if one exists) can be found in $O(D(n))$ time where $D(n)$ is the time to compute the distance product of two $n \times n$ real matrices, $D(n) = O(n^3 \log\log^3 n/\log^2 n)$.*

Proof. Let A be the negative of the adjacency matrix of G, *i.e.* for all $i, j \in [n]$,

$$A[i, j] = \begin{cases} -w(i, j) & \text{if } (i, j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

Let C be the negative of the distance product of A and A, *i.e.*

$$C[i, j] = -\min_{k\in[n]}(A[i, k] + A[k, j]) = -\min_{k\in[n]}(-w(i, k) - w(k, j)) = \max_{k\in[n]}(w(i, k) + w(k, j)).$$

Go through all $i, j \in [n]$ and compute in $O(n^2)$ additional time

$$W = \max_{i,j}[w(j, i) + C[i, j]] = \max_{i,j,k}[w(j, i) + w(i, k) + w(i, k)].$$

W is the weight of the maximum weight triangle in G, and the i, j pair for which it is achieved can be used to find the actual triangle in $O(n)$ time. The overall runtime is hence $O(D(n) + n^2)$ where $D(n)$ is the runtime to compute the distance product of A and A. The current best known algorithm for the distance product is by Chan [18] and runs in $O(n^3 \log\log^3 n/\log^2 n)$ time. □

The above theorem actually also shows that all pairs maximum weight triangles in edge-weighted graphs can be computed in $O(n^3 \log\log^3 n/\log^2 n)$ time. The all pairs version of the problem is actually just as hard as computing the distance product as the distance product of two $n \times n$ matrices A and B can be viewed as an instance of all pairs maximum weight triangles in an edge-weighted tripartite graph $(V_1 \cup V_2 \cup V_3, E, w)$ such that $|V_1| = |V_2| = |V_3| = n$: let v_i^j be the ith vertex of V_j for $j = 1, 2, 3$, $i \in [n]$; then for all $i, j \in [n], (v_i^1, v_j^2) \in E$, and $w(v_i^1, v_j^2) = -A[i, j]$; for all $i, j \in [n], (v_i^2, v_j^3) \in E$, and $w(v_i^2, v_j^3) = -B[i, j]$; for all $i, j \in [n], (v_i^3, v_j^1) \in E$, and $w(v_i^3, v_j^1) = 0$.

Corollary 4.3.1 *Computing the distance product of two $n \times n$ matrices and computing all pairs maximum weight triangles in an edge weighted graph on n nodes are computationally asymptotically equivalent.*

4.4 Maximum Node-Weighted Triangle in Sub-Cubic Time

We begin by deriving algorithms for the maximum node weighted triangle problem based on the dominance product. Afterwards we show how one can use rectangular matrix multiplication to reduce the running time.

4.4.1 A dominance product based approach

We first present a weakly polynomial deterministic algorithm, then a randomized strongly polynomial algorithm.

Theorem 4.4.1 *On graphs with integer weights, a maximum vertex-weighted triangle can be found in $O(n^{(\omega+3)/2} \cdot \log W)$ time, where W is the maximum weight of a triangle. On graphs with real weights, a maximum vertex-weighted triangle can be found in $O(n^{(\omega+3)/2} \cdot B)$ time, where B is the maximum number of bits in a weight.*

Proof. The idea is to obtain a procedure that, given a parameter K, returns an edge (i, j) from a triangle of weight at least K. Then one can binary search to find the weight of the maximum triangle, and try all possible vertices k to get the triangle itself.

We first explain the binary search. Without loss of generality, we assume that all edge weights are at least 1. Let W be the maximum weight of a triangle. Start by checking if there is a triangle of weight at least $K = 1$ (if not, there are no triangles). Then try $K = 2^i$ for increasing i, until there exists a triangle of weight 2^i but no triangle of weight 2^{i+1}. This i will be found in $O(\log W)$ steps. After this, we search on the interval $[2^i, 2^{i+1})$ for the largest K such that there is a triangle of weight K. This takes $O(\log W)$ time for integer weights, and $O(B)$ time for real weights with B bits of precision.

We now show how to return an edge from a triangle of weight at least K, for some given K. Let $V = \{1, \ldots, n\}$ be the set of vertices. We will create two $n \times n$ matrices, A and B. For every $i \in V$, we make a row of A, $A_i = (e(1), \ldots, e(n))$, where

$$e(j) = \begin{cases} K - w(i) & \text{if there is an edge from } i \text{ to } j \\ \infty & \text{otherwise.} \end{cases}$$

(In implementation, we can of course substitute a sufficiently large value in place of ∞.) We also make a column of B, $B^i = (e'(1), \ldots, e'(n))^T$, where

$$e'(j) = \begin{cases} w(i) + w(j) & \text{if there is an edge from } i \text{ to } j \\ -\infty & \text{otherwise.} \end{cases}$$

Compute the dominance product matrix $D(K)$ of the matrices A and B. For all edges (i, j) in the graph, check if there exists a k such that $A_i[k] \leq B^j[k]$ by examining entry $D(K)[i, j]$. If such a k exists, then we know there is a vertex k such that

$$K - w(i) \leq w(j) + w(k) \implies K \leq w(i) + w(k) + w(j),$$

that is, there exists a triangle of weight at least K using edge (i, j). Observe that the above works for both directed and undirected graphs. □

In the above, the binary search over all possible weights prevents our algorithm from being strongly polynomial. We would like to have an algorithm that, in a comparison-based model, has a runtime with *no* dependence on the bit lengths of weights. Here we present a randomized algorithm that achieves this.

Theorem 4.4.2 *On graphs with real weights, a maximum vertex-weighted triangle can be found in* $O(n^{(\omega+3)/2} \cdot \log n)$ *expected worst-case time.*

We would like to somehow binary search over the collection of triangles in the graph to find the maximum. As this collection is $O(n^3)$, we would then have our strongly polynomial bound. Ideally, one would like to pick the "median" triangle from a list of all triangles, sorted by weight. But as the number of triangles can be $\Omega(n^3)$, forming this list is hopeless. Instead, we shall show how dominance computations allow us to efficiently and uniformly sample a triangle at random, whose weight is from any prescribed interval (W_1, W_2). If we pick a triangle at random and measure its weight, there is a good chance that this weight is close to the median weight. In fact, a binary search that randomly samples for a pivot can be expected to terminate in $O(\log n)$ time.

Let $W_1, W_2 \in \mathbb{R} \cup \{-\infty, \infty\}$, $W_1 < W_2$, and G be a vertex-weighted graph.

Definition 4.4.1 $\mathcal{C}(W_1, W_2)$ *is defined to be the collection of triangles in* G *whose total weight falls in the range* $[W_1, W_2]$.

Lemma 4.4.1 *One can sample a triangle uniformly at random from* $\mathcal{C}(W_1, W_2)$*, in* $O(n^{(\omega+3)/2})$ *time.*

Proof. From the proof of Theorem 4.4.1, one can compute a matrix $D(K)$ in $O(n^{(\omega+3)/2})$ time, such that $D(K)[i,j] \neq 0$ iff there is a vertex k such that (i,k) and (k,j) are edges, and $w(i)+w(j)+w(k) > K$. In fact, the i,j entry of $D(K)$ is the *number* of distinct vertices k with this property.

Similarly, one can compute matrices $E(K)$ and $L(K)$ such that $E(K)[i,j]$ and $L(K)[i,j]$ contain the number of vertices k such that (i,k) and (k,j) are edges, and $w(i) + w(j) + w(k) \leq K$ (for E) or $w(i) + w(j) + w(k) < K$ (for L). (This can be done by flipping the signs on all coordinates in rows A_i of A and columns B^i of B from Theorem 4.4.1, then computing dominances, disallowing equalities for L.)

Therefore, if we take $F = E(W_2) - L(W_1)$, then $F[i,j]$ is the number of vertices k where there is a path from i to k to j, and $w(i) + w(j) + w(k) \in [W_1, W_2]$.

Let f be the sum of all entries $F[i,j]$. For each $(i,j) \in E$, choose (i,j) with probability $F[i,j]/f$. By the above, this step uniformly samples an edge from a random triangle. Finally, we look at the set of vertices S that are neighbors to both i and j, and pick each vertex in S with probability $\frac{1}{|S|}$. This step uniformly samples a triangle with edge (i,j). The final triangle is therefore chosen uniformly at random. □

Observe that there is an interesting corollary to the above.

Corollary 4.4.1 *In any graph, one can sample a triangle uniformly at random in* $O(n^\omega)$ *time.*

Proof. (Sketch) Multiplying the adjacency matrix with itself counts the number of 2-paths from each vertex to another vertex. Therefore one can count the number of triangles and sample just as in the above. □

We are now prepared to give the strongly polynomial algorithm.

Proof of Theorem 4.4.2: Start by choosing a triangle t uniformly at random from all triangles. By the corollary, this is done in $O(n^\omega)$ time.

Measure the weight W of t. Determine if there is a triangle with weight in the range (W, ∞), in $O(n^{(\omega+3)/2})$ time. If not, return t. If so, randomly sample a triangle from (W, ∞), let W' be its weight, and repeat the search with W'.

It is routine to estimate the runtime of this procedure, but we include it for completeness. Let $T(n, k)$ be the expected runtime for an n vertex graph, where k is the number of triangles in the current weight range under inspection. In the worst case,

$$T(n,k) \leq \frac{1}{k}\sum_{i=1}^{k-1} T(n,k-i) + c \cdot n^{(\omega+3)/2}$$

for some constant $c \geq 1$. But this means

$$T(n,k-1) \leq \frac{1}{k-1}\sum_{i=1}^{k-2} T(n,k-i) + c \cdot n^{(\omega+3)/2},$$

so

$$\begin{aligned} T(n,k) &\leq \left(\frac{1}{k} + \frac{k-1}{k}\right) \cdot T(n,k-1) \\ &\quad + \left(1 - \frac{k-1}{k}\right) cn^{(\omega+3)/2} \\ &= T(n,k-1) + \frac{c}{k} n^{(\omega+3)/2}, \end{aligned}$$

which solves to $T(n,k) = O(n^{(\omega+3)/2} \log k)$. □

4.4.2 A rectangular matrix product based approach

We reduce finding a maximum weight triangle in a node-weighted graph to computing the *maximum witness product* of two matrices. The maximum witness product of two $n \times n$ Boolean matrices A and B is the $n \times n$ matrix C with entries

$$C[i,j] = \max\left\{\{k \mid A[i,k] \cdot B[k,j] = 1\} \cup \{-\infty\}\right\}, \ \forall i,j \in [n].$$

We can in fact reduce the more general All Pairs Maximum Weights Triangle problem (APMWT) to computing a maximum witness product as follows. Suppose we are given a node-weighted graph $G = (V, E, w)$ and we want to find for all pairs of nodes $u, v \in V$ a triangle going through u and v of maximum weight sum. First sort all nodes in nondecreasing order of their weights. We obtain a sorted list $u_1, \ldots, u_n$. Create an $n \times n$ adjacency matrix A for G as follows: for all $i, j \in [n]$,

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Then computing the maximum witness product of A and its transpose A^T gives for every pair of nodes u, v the maximum weight node x (if any) for which u, v, x form a triangle.

The following result was proven initially by Kowaluk and Lingas [59], but we discovered it independently. It follows by some bucketting and rectangular matrix product.

Theorem 4.4.3 ([59]) *The maximum witness product of two $n \times n$ Boolean matrices can be computed in $O(n^{2.575})$ time.*

Proof. Let A and B be the two given matrices. Let r be a parameter. Create $\lceil \frac{n}{r} \rceil$ pairs of $n \times r$ matrices A_t and B_t (for $t = 0, \ldots, \lceil \frac{n}{r} \rceil - 1$) as follows. For all $i \in [n]$ and $j \in [r]$, $A_t[i,j] = A[i, tr+j]$ and $B_t[j,i] = B[tr+j, i]$. In other words, we let A_t contain the t-th block of r consecutive columns of A and B_t contain the t-th block of r consecutive rows of B.

For all t, compute the Boolean product of A_t and B_t using a rectangular matrix product algorithm [53]. By Lemma 3.1.1 this takes $O(r^{0.533} n^{1.844} \frac{n}{r})$ time.

After doing this we know for every $i, j \in [n]$ the maximum index block B_{ij} of r columns of A that contains a column k such that $A[i,k] \cdot B[k,j] = 1$. Since each B_{ij} contains at most r columns, we can try each one of them by brute force and find the maximum $k \in B_{ij}$ for which $A[i,k] \cdot B[k,j] = 1$. Over all $i, j \in [n]$ this takes $O(n^2 r)$ time.

To minimize the runtime, we set

$$n^2 r = r^{0.533} n^{1.844} \frac{n}{r} \implies r^{1.467} = n^{0.844} \implies r = n^{0.575}.$$

The overall runtime becomes $O(n^{2.575})$. □

The below corollary should be contrasted with the fact that APMWT in *edge-weighted* graphs is as hard to compute as the distance product for which no truly subcubic algorithm is known.

Corollary 4.4.2 *All Pairs Maximum Weight Triangles in a node-weighted graph can be computed in $O(n^{2.575})$ time.*

4.4.3 The Czumaj and Lingas approach and applications

Czumaj and Lingas [30] found a way to justify the intuition that finding one maximum node-weighted triangle should be easier than computing the maximum triangle for all pairs of nodes. They present a clever idea of reducing the problem of finding a maximum weight triangle to that of determining whether there is a triangle in the graph. Their algorithm is roughly as follows:

Let $T(n)$ be the runtime of determining the existence of a triangle in an unweighted n-node graph. Let $G = (V, E, w)$ be a given node-weighted graph for which we want to find the maximum weight triangle. We begin by creating a sorted list of the nodes (in nondecreasing order of $w(\cdot)$).

Our algorithm will be given 3 lists A, B, C of nodes from V in sorted order by their weights. The edges between nodes are induced from E. We will be guaranteed that there is a triangle with one node in each of A, B and C. Initially, $A = B = C = V$ where V is the vertex set in sorted order of w. We have checked in $T(n)$ time that there is a triangle in G.

We want to find a maximum weight triangle with endpoints in A, B, C. Let g be a (constant) parameter. If A, B and C have $\leq g$ nodes in them, then we can find a triangle with endpoints in A, B, C in $O(1)$ time. Otherwise, we begin by bucketting the nodes of A into g roughly equal consecutive lists $A_1, \ldots, A_g$. We do the same to B and C obtaining $\{B_1, \ldots, B_g\}$ and $\{C_1, \ldots, C_g\}$. Then, for all triples $\{A_i, B_j, C_k\}$ we check in $O(g^3 T(3n/g))$ time whether there is a triangle with endpoints in A_i, B_j, and C_k. We now have a list of all triples $\{A_i, B_j, C_k\}$ for which there is a triangle. If for some triple $\{A_i, B_j, C_k\}$ there exists another triple $\{A_{i'}, B_{j'}, C_{k'}\}$ in the list with $i' \geq i$, $j' \geq j$ and $k' \geq k$, then we remove $\{A_i, B_j, C_k\}$ from the list. Czumaj and Lingas show that there will be $\leq O(g^2)$ triples left in the list after this processing. We can then do $O(g^2)$ recursive steps.

The running time recurrence becomes

$$F(n) \leq g^3 T(3n/g) + g^2 F(n/g).$$

Suppose $T(n) = n^c$ for some $2 \leq c < 2.376$. Then the recurrence can be viewed as

$$F(n) \leq g^{3-c} n^c + g^2 F(n/g).$$

Czumaj and Lingas show that one can set $g = O(1)$ so that if $c > 2$, then $F(n) = O(n^c)$ and if $c = 2$, $F(n) = O(n^2 \log n)$. In particular, a maximum node-weight triangle can be found in $O(n^\omega) = O(n^{2.376})$ time.

We show that their approach can be extended to find for all pairs of nodes u, v in a node-weighted graph a triangle (if it exists) of weight sum $\geq K$, for a given global parameter K. This shows that the approach is powerful enough to handle some all-pairs problems, yet it is unclear whether it can be used to compute the maximum witness product.

Theorem 4.4.4 *Given a node-weighted graph $G = (V, E, w)$ and a parameter K, in $O(n^\omega)$ one can compute for all nodes $u, v \in V$ the number of triangles with weight sum $\geq K$ going through u and v.*

Proof. For proof simplicity we will omit the big-O notation. Just as in the maximum triangle algorithm, suppose we are given three lists of nodes A, B and C, each of size n. We are also given K, and we will return an $n \times n$ matrix which for every $a_i \in A$, $c_i \in C$ gives the number of $b_i \in B$ such that there is a triangle $\{a_i, b_i, c_i\}$ of weight at least K.

As before, bucket A, B, C each into g buckets of size $\lceil n/g \rceil$, $\{A_1, \ldots, A_g\}, \{B_1, \ldots, B_g\}, \{C_1, \ldots, C_g\}$. Create an all zero result matrix R.

For all triples $A_i = (a'', \ldots, a'''), B_j = (b'', \ldots, b'''), C_k = (c'', \ldots, c''')$ of buckets do:

- if either $w(a'') = w(a''')$ or $w(c'') = w(c''')$ (W.L.O.G. assume the former; the latter is analogous)
  - create two $\lceil n/g \rceil \times \lceil n/g \rceil$ matrices X and Y. X is the adjacency matrix of $A \to B$, *i.e.* for all $a \in A$, $b \in B$,
    $$X[a, b] = \begin{cases} 1 & \text{if } (a, b) \in E \\ 0 & \text{otherwise.} \end{cases}$$
    For all $b \in B$, $c \in C$,
    $$Y[b, c] = \begin{cases} 1 & \text{if } (b, c) \in E \text{ and } w(b) + w(c) \geq K - a'' \\ 0 & \text{otherwise.} \end{cases}$$
    Multiply X and Y in $O((n/g)^\omega)$ time. After this, add all edges which are determined to be parts of a triangle, entry by entry, to the corresponding entries in R; we are done with the triple.
- otherwise, create the adjacency matrices X and Y of $A \to B$ and $B \to C$ respectively. Compute the product D of X and Y in $O((n/g)^\omega)$ time.
  - if $w(b'') = w(b''')$, for all $(i, j) \in E$ such that $w(i) + w(j) \geq K - b''$, add $D[i, j]$ to $R[i, j]$; we are done with the triple.

- Now consider all remaining triplies $A_i = (a'', \ldots, a'''), B_j = (b'', \ldots, b'''), C_k = (c'', \ldots, c''')$ with $w(b'') \neq w(b'''), w(a'') \neq w(a'''), w(c'') \neq w(c''')$. No two of these triples share the same weights in any of the A, B or C node components. By Czumaj and Lingas' argument, the number of such triples for which $a'' + b'' + c'' \leq K < a''' + b''' + c'''$ is at most $O(g^2)$. Recurse on these triples in $O(g^2 F(n/g))$ time to obtain g^2 matrices which can be added to R. For triples with $a'' + b'' + c'' \geq K$ we do not need to recurse. We can just add the corresponding entries from D to R because any triangle in such a triple would be of weight at least K. For triples with $a''' + b''' + c''' < K$ we do nothing since they cannot contain relevant triangles. After all triples are processed, we return R.

The runtime is recurrence is

$$F(n) \leq g^3(n/g)^\omega + g^2 F(n/g) \implies F(n) = O(n^2 \log n + n^\omega) \text{ for some } g = O(1).$$

□

Chapter 5

Bottleneck Paths

While we are still unable to give a bona fide sub-cubic algorithm for APSP, we do present such an algorithm for an intimately related problem: computing *all-pairs bottleneck paths* (APBP) in general graphs. In this problem, one is given a directed graph with (arbitrary) capacities on its edges, and the problem is to report, for all pairs of vertices s, t, the maximum amount of flow that can be routed from s to t along any single path. (This amount is given by the smallest capacity edge on the path, a.k.a. the *bottleneck* edge.)

Definition 5.0.2 *Given a graph $G = (V, E, w)$ with $w : E \to \mathbb{R}$, the* bottleneck edge *of a path between vertices u and v is the edge on that path of smallest weight. A* maximum bottleneck path *between u and v is a path whose bottleneck edge weight is maximum over the bottleneck edge weights of all paths from u to v.*

The APBP problem has been studied alongside APSP in several contexts over the years. Pollack [74] introduced APBP (calling it the *maximum capacity route problem*), and showed how the cubic APSP algorithms of that time could be modified to solve it. Hu [52] proved that in *undirected* graphs, maximum capacity paths can be obtained by simply taking the paths in a maximum spanning tree. Therefore the problem on undirected graphs on n nodes can actually be solved in $O(n^2)$ time. The directed case of the problem has remained open until now, and recently appeared as an explicit goal in [83]. Prior to our work, the best known algorithm for the general case of APBP had been $O(mn + n^2 \log n)$ time (obtained by using Fredman and Tarjan's implementation of Dijkstra [43]).

Subramanian [87] proved that on random (Erdös-Renyi) graphs, both APBP and APSP can be solved in $O(n^2 \log n)$ time. Very recently, Shapira, Yuster, and Zwick [83] have given algorithms for APBP in the special case where the *vertices* have weights, but not the edges. Their algorithms use rectangular matrix multiplication, running in $O(n^{2.58})$ time. Note that the vertex-weight case can be easily reduced to the edge-weight case, by setting the weight of an edge to be the minimum weight of its two endpoints. Shapira, Yuster, and Zwick's algorithm relies on the linearity of the number of weights. As the number of weights in the vertex-weight case is only n, but the number in the edge-weight case can be $\Omega(n^2)$, their techniques do not seem to apply to the latter case.

We also consider the single source bottleneck paths problem (SSBP). In this version, we are given a source vertex s and we are asked to find maximum bottleneck paths from s to all other nodes in the graph. As in undirected graphs a maximum spanning tree solves the all pairs version of the problem, in particular SSBP can be solved in $O(m\alpha(n))$ time by using Chazelle's minimum

spanning tree algorithm [19]. Here, $\alpha(\cdot)$ is the inverse-Ackermann function. If randomization is allowed, SSBP in undirected graphs can be computed in *linear* time by Karger, Klein and Tarjan's linear time minimum spanning tree algorithm [55]. Dijkstra's algorithm [32] can also be adapted to solve SSBP, and with Fredman and Tarjan's Fibonacci Heaps [43] this gives an $O(m + n \log n)$ algorithm for SSBP. This is currently the fastest algorithm for general SSBP.

5.1 All Pairs Bottleneck Paths

We begin by investigating the all pairs bottleneck path problem on general graphs. We first note how to compute just the weights of the bottlenecks for all pairs of vertices, from now on referred to as *all pairs bottleneck distances*. We will then devote most of the rest of the chapter on how to obtain actual paths.

Our method for APBP is based on the $O(n^{2+\omega/3})$ algorithm (from Chapter 3) for computing the MaxMin product of two $n \times n$ matrices with arbitrary entries from $\mathbb{R} \cup \{\infty, -\infty\}$. Since $(\mathbb{R}, \max, \min, -\infty, +\infty)$ is a closed semiring, we immediately obtain the following corollary of Theorem 2.4.1 from Chapter 2.

Corollary 5.1.1 *If the MaxMin product of two arbitrary real $n \times n$ matrices is computable in $M(n)$ time, then all pairs bottleneck distances of an n vertex graph is computable in $O(M(n))$ time.*

Therefore, by Corollary 5.1.1 it follows that all pairs bottleneck distances can be computed in $O(n^{2.792})$ time by setting the adjacency matrix of the graph (as in Chapter 2) to be

$$A[u,v] := \begin{cases} \infty & \text{if } u = v \\ -\infty & \text{if } (u,v) \notin E \\ w(u,v) & \text{otherwise,} \end{cases}$$

and then computing its transitive closure under the $(\mathbb{R}, \max, \min, -\infty, +\infty)$ closed semiring.

5.1.1 Computing explicit maximum bottleneck paths

By Corollary 5.1.1 we can obtain a matrix representing all pairs bottleneck paths in an edge weighted directed graph in $O(n^{2+\frac{\omega}{3}})$ time. To compute the actual paths, a bit more work is necessary. We take an approach analogous to that used by Zwick [106] in computing all pairs shortest paths. First, we compute APBP by repeatedly squaring the original adjacency matrix via MaxMin product, instead of the approach in Aho et al. [1]. We also record, for every pair of vertices i, j, the last iteration $T[i,j]$ of the repeated squaring phase in which the bottleneck edge weight was changed, together with a witness vertex w_{ij} on a path from i to j, provided by the MaxMin product computation in that iteration.

Given an iteration matrix T and a witness matrix w_{ij} (derived from a shortest path computation), Zwick [106] gives a procedure which computes a matrix of successors in $O(n^2)$ time, and another procedure that, given a matrix of successors and a pair of vertices, returns a simple shortest path between the vertices. Applying his procedures to our setting, we get simple maximum bottleneck paths. The major difference here is that our iteration values are obtained by repeated squaring, whereas Zwick's iteration values come from his random sampling algorithm for finding witnesses. We review Zwick's algorithm in Figure 5.1.1.

```
algorithm wit-to-suc(W, T):
S ← 0
for ℓ = 0 to log n do T_ℓ = {(i, j) | T[i, j] = ℓ}
for every (i, j) ∈ T_0 do S[i, j] = j
for ℓ = 1 to log n do
    for each (i, j) ∈ T_ℓ do
        k = w_ij
        while S[i, j] = 0 do
            S[i, j] ← S[i, k], i ← S[i, j]
return S
```

Figure 5.1: Algorithm `wit-to-suc` converts the witness matrix into a successor matrix.

Theorem 5.1.1 *The all pairs bottleneck paths problem can be computed in $O(n^{2+\frac{\omega}{3}})$ time. Furthermore, in $O(n^{2+\frac{\omega}{3}} \cdot \log n)$ time algorithm* `wit-to-suc` *computes a successor matrix from which for any i, j a simple maximum bottleneck path between i and j can be recovered in $O(\ell)$ time, where ℓ is the length of the returned path.*

Proof. Let w_{ij} and $T[i, j]$ for all vertex pairs i, j be provided by repeated squaring of the adjacency matrix using MaxMin product.

Consider algorithm `wit-to-suc`. Let S be the matrix of successors that the algorithm computes. The algorithm processes vertex pairs (i, j) in increasing order of their iteration numbers $T[i, j]$. The idea is that if k is a witness for (i, j), then $T[i, k]$ is an earlier iteration of the squaring than $T[i, j]$, and hence $S[i, k]$ would be set before $S[i, j]$ is processed.

We claim by induction that after a value $S[i, j]$ is set, matrix S stores a simple maximum bottleneck path from i to j which can be recovered by following successors one by one. Our argument is similar to that of Zwick [106].

At iteration 0 of the algorithm, all pairs whose maximum bottleneck path is an edge are fixed. Suppose that at the iteration in which vertex pair (i, j) is processed, the claim holds for all vertex pairs (k, ℓ) that have been processed before (i, j) (and hence which have a nonzero $S[k, \ell]$ value). Now consider the iteration in which (i, j) is processed. Let $k = w_{ij}$. Since $S[i, k]$ is set, we can use its successor value to set $S[i, j]$ since we know that a maximum bottleneck path goes through k. We then take $S[i, j]$ and if its successor on the path to j has not been set. we set it to match $S[S[i, j], k]$. We continue processing consecutive successors similarly, until we encounter some i_0 for which $S[i_0, j]$ is set (i_0 exists as k is such a vertex). Since it is set, and the path from i to k is simple (by induction), $S[i_0, j]$ must have been set before (i, j) is processed. Hence by induction, the path from i_0 to j is simple and all successors for vertices on that path to j are set. But since no successors for vertices between i and i_0 were set, then the paths i to i_0 and i_0 to j are simple and nonoverlapping, and the overall path is simple and a maximum bottleneck path. Furthermore, now the successors of all vertices on the simple path are set in the S matrix.

The algorithm for obtaining successors from witnesses takes $O(n^2)$ time. Given a matrix of successors, obtaining the actual path from i to j is straightforward: find $S[i, j]$ and then recursively obtain the path from $S[i, j]$ to j. This clearly takes time linear in the length of the path. □

5.2 All Pairs Bottleneck Shortest Paths

Consider a scenario in which we want to get from location u to location v in as few hops as possible, and subject to this, we wish to maximize the flow the we can route from u to v. In other words, we want to compute for each pair of vertices, the shortest (unweighted) distance $d(u, v)$ and the maximum bottleneck weight $b(u, v)$ of a path of length $d(u, v)$ from u to v. This is the all pairs bottleneck shortest paths problem (APBSP). This problem was considered by Shapira, Yuster and Zwick [83] who gave a truly subcubic algorithm in the node-weighted graph case.

5.2.1 The "Short Paths, Long Paths" method

Shapira, Yuster and Zwick used the well-known *short path-long path method* [106, 49, 18] which we will use again in the next chapter. The method proceeds to first design a single source algorithm for the path problem, running in $O(T(n))$ time for some $T(n)$. Then one picks a parameter $\ell < n$. One iterates a matrix product on the adjacency matrix ℓ times to obtain best paths between $\leq n^2$ pairs of vertices. This takes, say, $O(M(n)\ell)$ time, where $M(n)$ is the time to compute the matrix product.

Then one uses the following lemma to obtain in $O(n^2\ell)$ time a set of $\frac{n \log n}{\ell}$ vertices hitting all shortest paths between vertices at distance ℓ.

Lemma 5.2.1 ([106, 49, 18]) *Given a collection of N subsets of $\{1, \ldots, n\}$, each of size ℓ, one can find in $O(N\ell)$ time a set of $\frac{n \log n}{\ell}$ elements of $\{1, \ldots, n\}$ hitting every one of the subsets.*

One way to prove the lemma is by random sampling, another is by the greedy approximation to hitting set.

After obtaining the hitting set, one argues that for any pair of vertices some best path of length $\geq \ell$ (if one exists) must contain a node from the hitting set. Then one runs the single source algorithm from all nodes in the hitting set in $O(\frac{nT(n) \log n}{\ell})$ time. Finally, one combines the results in $O(\frac{n^3 \log n}{\ell})$ time by considering every pair of nodes and every possible midpoint from the hitting set. Suppose $T(n) = O(n^2)$ as is with most problems for which Dijkstra's algorithm applies. Then the overall running time is minimized when

$$M(n)\ell = \frac{n^3 \log n}{\ell}, \ \ell = \sqrt{\frac{n^3}{M(n)} \log n},$$

and the runtime becomes

$$O\left(n^{1.5}\sqrt{M(n) \log n}\right).$$

We will apply the method once in the following section and once in the next chapter when discussing all pairs nondecreasing paths.

5.2.2 APBSP

Following the short path-long path approach just as in [83] we can obtain the following.

Theorem 5.2.1 *APSBP on an n node graph can be found in $O(n^{2.896})$ time.*

We first give a single source algorithm for bottleneck shortest paths.

Lemma 5.2.2 *SSSBP on a graph with m edges and n nodes can be found in $O(m+n)$ time.*

Proof. The algorithm is an adaptation of breadth first search. Let $G = (V, E)$ be the given directed graph, $w : E \to \mathbb{R}$ and s be the source node. We will maintain a set Q_i which at each stage i which will contain nodes at (unweighted) distance i from s. The set needs to support insert, pop an element, go through the elements one by one. A linked list suffices to support all of these operations in $O(1)$ time.

Every node v in the graph has a bit $visited(v)$ which is set if and only if an edge from an in-neighbor of v to v has been traversed. Node v has values $d(v)$ and $b(v)$ associated with it. Value $d(v)$ will be the (unweighted) shortest distance from s to v and $b(v)$ will be the maximum bottleneck edge on a shortest path from s to the v. Originally, $d(v) = \infty$ for $v \neq s$ and $d(s) = 0$, $b(v) = -\infty$ for all $v \neq s$ and $b(s) = \infty$, $visited(v) = 0$ for all $v \neq s$, $visited(s) = 1$.

We begin by inserting each out-neighbor v of s into Q_1. We set $d(v) = 1$, $b(v) = w(s, v)$ and $visited(v) = 1$. We then process Q_1.

To process Q_i, repeat: pop a node v from Q_i; for all out-neighbors u of v:

- if u is not visited, insert u into Q_{i+1}, set $d(u) = i + 1$, $b(u) = \max\{b(u), \min\{b(v), w(v, u)\}\}$, and $visited(u) = 1$.
- else if u is visited and if $d(u) = i + 1$, set $b(u) = \max\{b(u), \min\{b(v), w(v, u)\}\}$.

When Q_i is empty, process Q_{i+1} if Q_{i+1} is nonempty.

Correctness follows by induction: if the bottlenecks for nodes in Q_{i-1} are correct, then since every path of length i from s to u must be of the form P_{sv} followed by (v, u) where P_{sv} is a path of length $i - 1$ and $(v, u) \in E$, going through all nodes $v \in Q_{i-1}$ and setting $b(u) = \max\{b(u), \min\{b(v), w(v, u)\}\}$ computes the correct bottleneck weight.

The running time of the algorithm is $O(m+n)$ since we go through each edge at most once and a node is only accessed via an incoming edge or when popping it from a set Q_i. □

Now we can apply the short path-long path method to prove Theorem 5.2.1.

Proof.[Theorem 5.2.1] In our application of the method, the matrix product we use is the MaxMin product, taking $M(n) = O(n^{2+\frac{\omega}{3}})$ per iteration. Recall that the adjacency matrix A is defined as follows for $j, k \in [n]$:

$$A[j,k] = \begin{cases} \infty & \text{if } j = k \\ w(j,k) & \text{if } (j,k) \in E \\ -\infty & \text{otherwise.} \end{cases}$$

For a parameter ℓ, we iterate the MaxMin product on the adjacency matrix ℓ times as follows: for $i = 2, \ldots, \ell$, compute $C^i = A^{i-1} \odot A$ (where $\odot$ is the MaxMin product). Here for $j, k \in [n]$:

$$A^1[j,k] = \begin{cases} w(j,k) & \text{if } (j,k) \in E \\ -\infty & \text{otherwise.} \end{cases}$$

At iteration 1 we have an $n \times n$ matrix D with $D[i, i] = 1$ for all $i \in [n]$, and for $i \neq j$:

$$D[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise.} \end{cases}$$

Matrix D contains the unweighted distance between any two nodes at distance at most ℓ. After computing C^i, set for all $j, k \in [n]$

$$D[j,k] = \begin{cases} D[j,k] & \text{if } D[j,k] < \infty \\ i & \text{if } C^i[j,k] > -\infty \text{ and } D[j,k] = \infty \\ \infty & \text{otherwise.} \end{cases}$$

After D is updated, we create A^i by setting for all $j, k \in [n]$

$$A^i[j,k] = \begin{cases} C^i[j,k] & \text{if } D[j,k] = i \\ -\infty & \text{otherwise.} \end{cases}$$

$A^i[j,k]$ is the maximum bottleneck edge weight on a path from j to k of length i if i is the shortest distance between j and k. Over all $i = 1, \ldots, \ell$, computing all A^i and D takes $O(\ell n^{2+\frac{\omega}{3}})$ time.

The short path-long path method requires that we find a hitting set S in $O(\ell n^2)$ time and do SSBSP from all nodes in S. By Lemma 5.2.2 this takes $O(n^3 \log n/\ell)$ time. We obtain all distances and bottlenecks by combining the results in $O(n^3 \log n/\ell)$ as given by the method. We set

$$\ell = \sqrt{\frac{n^3}{M(n)} \log n} = n^{\frac{3-\omega}{6}},$$

and the runtime becomes

$$O\left(n^{1.5}\sqrt{M(n)\log n}\right) = O(n^{\frac{15+\omega}{6}} \log n) = O(n^{2.896}).$$

□

5.3 Single Source Bottleneck Paths

We investigate the single source bottleneck paths problem (SSBP). We first note that the problem is hard in constant degree graphs in the following sense.

Lemma 5.3.1 *If SSBP can be computed in $O(T(n))$ time in n-node graphs of $O(1)$ maximum degree, then SSBP can be computed in $O(m + n + T(3m))$ time in general graphs on n nodes and m edges.*

Lemma 5.3.1 is a simple corollary of the following general lemma on semiring path problems.

Lemma 5.3.2 *Let P be a path problem over a semiring $(R, \oplus, \odot, 0, 1)$. If P can be computed in $O(T(n))$ time in n-node graphs of $O(1)$ maximum degree, then P can be computed in $O(m + n + T(3m))$ time in general graphs on n nodes and m edges.*

Proof. We prove the lemma by a reducing P in a general graph with m edges and n nodes to P in an m-node m-edge graph with outdegree ≤ 2 and indegree ≤ 2.

Let $G = (V, E, w)$ be the input of the P problem such that $|E| = m$ and $|V| = n$. Create a graph $G' = (V', E', w')$ as follows.

Consider each node $v \in V$. Let $N_{in}(v)$ and $N_{out}(v)$ be the in-neighbors and out-neighbors of v respectively. For every node $u \in N_{in}(v)$ create a node $v_u^{in} \in V'$. Likewise, for every node

$u \in N_{out}(v)$ create a node $v_u^{out} \in V'$. Pick any ordering on the nodes in $N_{in}(v)$ and one on the nodes in $N_{out}(v)$, so that

$$N_{in}(v) = \{i_1, \ldots, i_{|N_{in}(v)|}\},$$

$$N_{out}(v) = \{o_1, \ldots, o_{|N_{out}(v)|}\}.$$

According to this ordering, add an edge $\left(v_{i_j}^{in}, v_{i_{j+1}}^{in}\right)$ to E' for every $j = 1, \ldots, |N_{in}(v)| - 1$, and an edge $\left(v_{o_j}^{out}, v_{o_{j+1}}^{in}\right)$ to E' for every $j = 1, \ldots, |N_{out}(v)| - 1$. If $N_{in}(v)$ and $N_{out}(v)$ are nonempty, also add edges $\left(v_{o_{|N_{out}(v)|}}^{out}, v_{i_1}^{in}\right)$ and $\left(v_{i_{|N_{out}(v)|}}^{in}, v_{o_1}^{out}\right)$. If $N_{in}(v)$ is empty, add an edge $\left(v_{o_{|N_{out}(v)|}}^{out}, v_{o_1}^{out}\right)$, and if $N_{out}(v)$ is empty, add an edge $\left(v_{i_{|N_{in}(v)|}}^{in}, v_{i_1}^{in}\right)$. The edges we have added form a cycle on the nodes corresponding to v. Let the weights of all the cycle edges be 1. Finally, for every edge $(u, v) \in E$, create an edge $e_{uv} = \left(u_v^{out}, v_u^{in}\right) \in E'$ with weight $w'(e_{uv}) = w(u, v)$. An example of this construction is shown in Figure 5.2.

Now we show that for any two nodes $u, v \in V$ there is a path of weight w in G if and only if there is a path of weight w between some node u_x^{out} and some node v_y^{in}. Let $u \to x_1 \to \ldots \to x_t \to v$ be a path with weight w in G. Then consider the following path in G'. Start with $u_{x_1}^{out} \to {x_1}_u^{in}$, then follow the path around the cycle of G' corresponding to x_1 to ${x_1}_{x_2}^{out}$, then follow edge $\left({x_1}_{x_2}^{out}, {x_2}_{x_1}^{in}\right)$. Inductively, follow the cycle corresponding to x_j from ${x_j}_{x_{j-1}}^{in}$ to ${x_j}_{x_{j+1}}^{out}$ and then the edge $\left({x_j}_{x_{j+1}}^{out}, {x_{j+1}}_{x_j}^{in}\right)$. The cycle edges have 1 weights and hence do not contribute to the path weight, as $\odot$ is associative and $1 \odot x = x \odot 1 = x$ for all $x \in R$. The rest of the edges are in one to one correspondence with the edges of the path from G and hence this path in G' has weight w. On the other hand, any path in G' is composed of paths within a cycle linked by edges between cycles. Moreover, any path which takes a path through the cycle of u_1, goes through some edge $\left({u_1}_{u_2}^{out}, {u_2}_{u_1}^{in}\right)$, then through the cycle of u_2 etc., corresponds to a path in G which starts from u_1, takes the edge (u_1, u_2) etc. The weights of the two paths must be equal since the only difference is in the weights of cycle edges which are 1.

We have thus reduced G to a graph G' on $2m$ nodes and $3m$ edges such that there is a mapping from paths in G to paths in G' such that weights are preserved. Our reduction took $O(m+n)$ time. Hence if the P problem in G' can be solved in $O(T(|V'|) = O(T(3m))$ time, then P in G can be solved in $O(n + m + T(3m))$ time. □

The above lemma implies that if one can solve SSBP in constant degree graphs in linear time, then SSBP is in linear time for general directed graphs (this of course holds for all single source semiring path problems, *e.g.* for SSSP). We do not know however of a faster algorithm for SSBP than the easy $O(m + n \log n)$ time modification of Dijkstra's algorithm. We hence consider the seemingly easier problem of finding the maximum bottleneck path between two given vertices in the graph. This is the so called *single source-single destination* bottleneck paths problem. It is also referred to as the *maximum capacity path* problem [74].

5.3.1 Single source-single destination bottleneck paths

The Single Source-Single Destination Bottleneck Paths, also referred to as the s-t maximum bottleneck path problem (s-t BP), is mainly studied in relation to the maximum flow problem. Edmonds and Karp [36] showed that if s-t BP in an m-edge graph can be computed in $O(T(m))$ time, then the maximum flow in any m-edge graph with integral edge weights can be computed in $O(mT(m) \log C)$

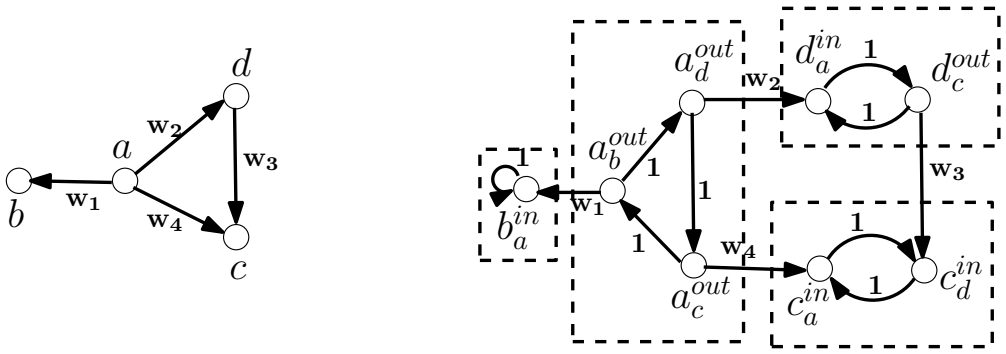

Figure 5.2: An example transformation of a graph with maximum outdegree 3 into one with maximum indegree and outdegree 2.

time where C is the maximum edge weight in the graph. They proceeded to show an $O(m^2 \log C)$ algorithm for maximum flow based on the above observation. Of course, s-t BP requires $\Omega(m)$ time just to read the input, and hence we cannot trivially beat the $O(m^2 \log C)$ algorithm for maximum flow by improving the known s-t BP algorithms. Nevertheless, efficient algorithms for s-t BP are still of interest, as bottleneck paths have other applications. Moreover, a really good algorithm for s-t BP may also give some intuition into the maximum flow problem.

s-t BP in Undirected Graphs. In the beginning of the chapter we noted that for undirected graphs one can even solve the all-pairs bottleneck paths problem in $O(m+n)$ randomized time. The best deterministic runtime for the single source version of the problem however is $O(m\alpha(n))$ [19] where $\alpha(n)$ is the inverse-Ackermann function. We give a simple *linear* time *deterministic* algorithm for the single source-single destination version of the problem. This solution turned out to be folklore. It was published by Camerini [15] in 1978 and later by Punnen [76] in 1991.

Theorem 5.3.1 *Let G be an undirected graph with real weights and s and t be two given vertices. Then the maximum bottleneck path between s and t can be found in $O(m+n)$ time, where m and n are the number of edges and the number of vertices in G respectively.*

We will do a binary search for the maximum bottleneck weight between s and t. We need two results:

Lemma 5.3.3 (Median Finding [11]) *Given m elements from a totally ordered set, their median can be found in $O(m)$ time.*

Lemma 5.3.4 (Connected Components [88]) *All connected components consisting of at least two nodes in a graph on m edges can be found in $O(m)$ time.*

Proof.[Theorem 5.3.1] We assume without loss of generality that the edge weights are distinct, otherwise, define a new weight $w(e)$ for each edge so that $w(e) = (w'(e), \ell(e))$ where $w'(e)$ is the old weight and $\ell(e)$ is a unique label for edge e from $1, \ldots, m$. The total order on these weights is defined as follows: $w(e) < w(e')$ if $w'(e) < w'(e')$ or if $w'(e) = w'(e')$ and $\ell(e) < \ell(e')$.

At each point we will work with an edge range $[e', e'']$ with $w(e') < w(e'')$, so that we initially have that e' is the edge of minimum weight and e'' is the edge of maximum weight in G. We will assume that the maximum bottleneck edge between s and t is in the edge range $[e', e'']$.

Let m' be the number of edges e with weights $w(e') \leq w(e) \leq w(e'')$. In $O(m')$ time find the median edge e_m of the range $[e', e'']$. Then, in $O(m')$ time find the connected components of the graph G' induced in G by the edges in $[e_m, e'']$. If s and t are in the same connected component of G', then any maximum bottleneck path between s and t in G is also in G'. We can recurse on range $[e_m, e'']$.

Otherwise, suppose that s and t are in different connected components of G'. Then we create a graph G'' in $O(m')$ time: we create a node v_C in G'' for each connected component C of G' (there are at most m' of these); we then go through the edges of G' and for any edge (u, v) with $u \in C$ and $v \in C'$ with $C \neq C'$ we add an edge $(v_C, v_{C'})$ in G''. Let C_s and C_t be the components of G' containing s and t respectively. Then we search in the same way as above for the maximum bottleneck path in G'' from v_{C_s} to v_{C_t}. The number of edges in G'' is $\leq m'/2$ as all edges in $[e_m, e'']$ were merged to form connected components.

The running time recurrence becomes

$$T[m] \leq c \cdot m + T[m/2],$$

for some constant c. Hence the algorithm runs in $O(m)$ time. □

5.3.2 Single source-single destination bottleneck paths in directed graphs

The binary search approach above which we used for undirected graphs does not work immediately for directed graphs. If we try to adapt the approach trivially, we could replace finding connected components by finding strongly connected components. However, in this way we cannot guarantee that we would reduce the number of edges a lot by shrinking components since by shrinking the components, instead of isolated vertices we would obtain a DAG which may still contain many edges.

We get around this problem by changing the algorithm. We begin a lemma which asserts that if we have sorted edge weights, the problem can be solved in linear time.

Lemma 5.3.5 *Let $G = (V, E)$ be a graph with edge weights $w : E \to \{1, \ldots, m\}$, then SSBP from source $s \in V$ in G can be solved in $O(m + n)$ time.*

Proof. Our algorithm is very similar to Dial's [31, 2] implementation of Dijkstra's algorithm.

For every edge weight $c \in \{1, \ldots, m\}$ we create a *bucket* B_c. Let $c_{m'}, c_{m'-1}, \ldots, c_1$ be the edge weights in decreasing order ($m' \leq m$). Then we maintain the buckets in a linked list with pointers from each c_i to c_{i-1}. Each bucket itself will be a linked list, initially empty. For each

We maintain a value $b[v]$ for each node which will be the maximum bottleneck weight for a path from s to v discovered so far. Initially, $b[s] = \infty$ and $b[v] = -\infty$ for all other nodes.

The algorithm begins by going through all edges out of s. For every such edge $e = (s, v)$, place v in $B_{w(e)}$ by putting it at the front of the linked list. We remove e from the graph. Assuming that given e we can access $B_{w(e)}$ in $O(1)$ time this assignment of the neighbors of s to buckets takes $O(d(s))$ where $d(s)$ is the outdegree of s.

After this, starting with $i = m'$ and decreasing i we find the largest i such that B_i is nonempty. We process each node $u \in B_i$ as follows. Set $w = \min\{d[u], w(u, v)\}$. place each outneighbor v of u

in B_w and remove edge (u, v). After finishing B_i we move on to the next nonempty bucket in the linked list of buckets etc. Since we remove each edge when it is accessed, each edge is accessed at most once and the runtime is $O(m + n)$.

The correctness of the algorithm follows from the correctness of Dijkstra's algorithm as the nodes are processed in the (reverse) sorted order of their maximum bottleneck distances.

The algorithm can be implemented in $O(m + n)$ time on a pointer machine provided the edges of G are provided in their (reverse) sorted order, say in a linked list. □

Single source-single destination. We now give an easy $O(m \log\log n)$ algorithm for single source-single destination maximum bottleneck problem: Let $G = (V, E, w)$ be given with two nodes s and t. W.L.O.G. assume that the each edge has distinct weight. We can ensure that by tacking on an $O(\log n)$ bit edge label to the end of the edge weight. Pick a parameter p. In $O(m \log p)$ time find p edges $e_1, \ldots, e_p$ so that there are at most $m/p + 1$ edges between e_i and e_{i+1} in the sorted order. This can be done by the linear time median finding algorithm of Blum *et al.* [11]. Then, for every edge e between e_i and e_{i+1} let $w'(e) = i$. Solve the maximum bottleneck problem from s to t in $G' = (V, E, w')$ in $O(m+n)$ time using Lemma 5.3.5. Let j be the weight of the maximum bottleneck edge between s and t in G'. Then the maximum bottleneck edge between s and t in G is between e_j and e_{j+1} in the sorted order of the edges.

We remove all edges smaller than e_j. The rest of the edges form a set E''. For all edges e after e_{j+1} in the sorted order, set $w''(e) = \infty$. Sort all edges between e_j and e_{j+1} in $O(m/p \log m)$ time setting their weights w'' as their rank in the sorted order. Then apply Lemma 5.3.5 again to solve the problem for $G'' = (V, E'', w'')$ in $O(m + n)$ time.

The overall runtime is $O(m/p \log m + m \log p)$ and is minimized for $p = \log m / \log\log m$, giving $O(m \log\log n)$ time.

We discovered the above algorithm and then noticed that it is known. In fact, Gabow and Tarjan [47] discovered it in 1988 also showing that using a clever recursive technique one can bring the running time down to $O(\min\{m \log^* n, m + n \log n\})$ where $\log^* n$ is the number of times we do $k := 2^k$ starting with $k = 2$ until we exceed n (their algorithm also works for bottleneck spanning tree). Punnen [75] gave the same runtime for the problem also noting that any bottleneck problem which can be solved in linear time when the input weights are given in sorted order, can be solved in $O(m \log^* n)$ time.

Theorem 5.3.2 (Gabow and Tarjan [47]) *Single source-single destination maximum bottleneck path can be computed in $O(m \log^* n)$ time.*

Proof. In recursive step i, set $p_i = 2^{p_{i-1}}$ with $p_1 = 2$. Let there be m_i edges in the graph $G_i = (V, E_i, w_i)$ of stage i with finite weights. Initially $m_1 = m$ and we will have $m_i = m_{i-1}/p_{i-1}$.

In $O(m_i \log p_i)$ time find p_i edges $e_1, \ldots, e_p$ so that there are at most $m/p_i + 1$ finite weight edges between e_i and e_{i+1} in the sorted order. Then, for every edge e between e_i and e_{i+1} let $w'(e) = i$, and for all infinite weight edges set $w'(e) = \infty$. Solve the maximum bottleneck problem from s to t in $G' = (V, E_i, w')$ in $O(m + n)$ time using Lemma 5.3.5. Let j be the weight of the maximum bottleneck edge between s and t in G'. Then the maximum bottleneck edge between s and t in G_i is between e_j and e_{j+1} in the sorted order of the edges.

We remove all edges smaller than e_j. The rest of the edges form the set E_{i+1}. For all infinite edges and all edges e after e_{j+1} in the sorted order, set $w_{i+1}(e) = \infty$. For all edges e between e_j

and e_{j+1} set $w_{i+1}(e) = w_i(e)$. Move on to stage $i+1$ with graph $G_{i+1} = (V, E_{i+1}, w_{i+1})$. Note that the number of finite weight edges in G_{i+1} is $m_{i+1} \leq m_i/p_i$ by construction.

The running time of stage i is $O(m_i \log p_i + m + n)$. As $m_i \log p_i = (m_{i-1}/p_{i-1}) \cdot p_{i-1} = m_{i-1}$ we have that

$$\sum_i (m_i \log p_i) \leq m_1 + \sum_{i \geq 1} m_i < m_1 + \sum_i (m_i/2^i) = O(m).$$

Hence the runtime is dominated by the linear time runs of the algorithm of Lemma 5.3.5. There are $O(\log^* m)$ such runs as we can stop the recursion when $m_i = 1$. The overall runtime is then $O(m \log^* n)$ (here we are assuming that $m \geq n-1$ as we only need to consider the weakly connected component containing s and t). □

One weakness of the above algorithm is that it always goes through the infinite edges. One could imagine that there is a way to store some information with each recursive call which would allow us to shortcut some of these infinite edges. Such a result would lead to a runtime improvement.

Chapter 6

Nondecreasing Paths

We consider the problem of planning a train trip so that we arrive at our destination as early as possible. The travel booking office has a timetable with departure and arrival times for all scheduled trains, together with their origins and destinations. From this data, we wish to extract the best choice of train connections possible, given our start city and preferred departure time. While this is an old problem, it is of course still interesting and relevant today. To our knowledge, the problem was first studied in the theory of algorithms in 1958 in a paper by Minty [63]. Minty reduced it to a problem on edge-weighted graphs as follows. The train stops are mapped to vertices. If there is a train scheduled from city A to city B leaving A at time t_1 and arriving at B at time t_2, one creates a vertex v for the scheduled train and adds an edge from A to v with weight t_1 and an edge from v to B with weight t_2. A train trip from a start S to a destination T is only valid if one can take each consecutive train on the trip without missing a connection. In particular, this means that for any two consecutive trains, the arrival time of the first one must be no later than the scheduled departure time of the second one. In the graph this corresponds to a *nondecreasing* path from S to T, *i.e.* a path, the weights on which form a nondecreasing sequence. In order to find the route which gets us to T at the earliest possible time, we must find a nondecreasing path minimizing the last edge weight on the path.

Minty [63] also defined a generalization of the problem: given an edge weighted graph $G = (V, E, w)$ and a source $s \in V$, report for every $t \in V$, the minimum last edge weight on a nondecreasing path from s to t (∞ if such a path does not exist). We call this the single source minimum nondecreasing paths problem (SSNP).

SSNP is a variant of the popular single source shortest paths problem (SSSP). In SSNP the sum operation from SSSP is replaced by an inequality ($\leq$) constraint. SSNP was studied alongside SSSP and many of the algorithms for the two problems are quite similar. Minty gave an algorithm for SSNP running in $O(mn)$ time, where $n = |V|$ and $m = |E|$. In the following year, Moore [66] gave a similar $O(mn)$ time algorithm for SSSP and showed how to transform it into one for SSNP. Incidentally, Moore's algorithm strikingly resembles the Bellman-Ford algorithm [8, 41].

Nowadays much faster solutions are known for both problems. Dijkstra's algorithm [32] using Fredman and Tarjan's implementation [43] solves SSSP with nonnegative weights in $O(m + n \log n)$ time. A folklore modification of that same algorithm also gives an $O(m + n \log n)$ time algorithm for SSNP in the comparison-based model of computation.

The restriction of SSNP for nonnegative weights has been studied in the algorithmic applications community under the name the *earliest arrival problem*. It seems that most of the research has been

devoted to analyzing different graph representations and various heuristics to optimally implement Dijkstra's algorithm (*e.g.* [12, 77, 81]). Yet no asymptotically better algorithm for the problem has been presented, to our knowledge.

In the word RAM (transdichotomous) model of computation, Thorup [90] gave a linear time algorithm for SSSP on *undirected* graphs with nonnegative weights. In this paper we how that in the word RAM model of computation there is a *linear* time algorithm for SSNP, even for *directed* graphs with *arbitrary* weights. In contrast, there is no known linear time algorithm for SSSP on weighted directed graphs. Our algorithm can be implemented to run in $O(m \log\log n)$ time in the comparison based model of computation. This is better than Dijkstra's algorithm for $m = o(n \log n / \log\log n)$.

We also consider an all pairs version of the minimum nondecreasing paths problem. In the all pairs nondecreasing paths problem (APNP) one is given an edge-weighted graph and one must return for all pairs of vertices s and t the minimum last weight on a nondecreasing path from s to t. We give a truly subcubic (in n) algorithm for APNP. Our runtime is $O(n^{2.896})$. Recall that although some truly subcubic algorithms (*i.e.* $O(n^{3-\delta})$ for constant $\delta > 0$) are known for certain interesting restrictions of the all pairs shortest paths (APSP) problem [82, 49, 84, 106, 18], there is no known truly subcubic algorithm for APSP. In the previous chapter we showed that the related all pairs maximum bottleneck paths problem (APBP) is solvable in truly subcubic time. In retrospect, one can say that the algorithm for APBP in [97] was based on solving a special case of APNP. In particular, one can show that APNP is at least as hard as APBP. APNP is also at least as hard as finding dominating pairs in $\mathbb{R}^n$, solvable in $O(n^{2.688})$ time by Matousek's algorithm. Any improvement to this running time would imply better algorithms for many problems such as computing the most significant bits of the distance product [95].

The results on SSNP and APNP appear in our paper in SODA08 [94]. In this chapter we also present an extension which did not appear in the original paper: an efficient algorithm for cheapest increasing paths.

6.1 All Pairs Nondecreasing Paths

In the all pairs version of the nondecreasing paths problem, we are interested in an algorithm with running time which is subcubic in terms of n, preferably $O(n^{3-\delta})$ for some constant $\delta > 0$. Observe that obtaining a cubic running time is easy - just run the SSNP algorithm from the next section on all vertices.

We begin with a motivating example. Consider the variant of APNP in which the weights are on the vertices instead of on the edges. In this case, the last weight on a path from s to t is defined as the weight of the node just before t on the path, or $-\infty$ if $s = t$. This is clearly a restriction of APNP since we can convert the node weights into edge weights by setting the weight of an edge (u, v) to the weight of u. A subcubic algorithm for vertex weighted APNP is as follows: first go through the graph and create a restricted adjacency matrix A such that $A[i, j] = 1$ if $w(i) \leq w(j)$ and $(i, j) \in E$, and $A[i, j] = 0$ otherwise. Compute the transitive closure of A in $O(n^\omega)$ time. This gives a matrix T such that $T[i, j] = 1$ if and only if there is a nondecreasing path from i to j with the property that the last weight on the path is $\leq w(j)$. Now, create the actual adjacency matrix B of the graph such that $B[i, j] = 1$ if $(i, j) \in E$ and $B[i, j] = 0$ otherwise. W.L.O.G. assume that the vertices are sorted by their weights so that $i \leq j$ iff $w(i) \leq w(j)$. Now use the minimum witness algorithm for boolean matrix multiplication [59] to find for every pair of vertices i, j for which $(TB)_{ij} = 1$ the minimum value $w(k)$ such that $T_{ik}B_{kj} = 1$. The overall algorithm

runs in $O(n^{2.575})$ time. Furthermore, vertex weighted APNP is at least as hard as the problem of finding minimum witnesses of boolean matrix multiplication, since the latter is just the restriction of vertex weighted APNP on tripartite graphs with $-\infty$ weights in the first partition. Hence, the above runtime cannot be improved unless there is a better algorithm for minimum witnesses.

Now we consider the more general edge weighted case. We will use Theorem 3.4.1. Recall, the theorem states that the $(\min, \leq^r)$-product $C = A \circledcirc B$ of two $n \times n$ matrices A and B can be computed in $O(n^{2+\frac{\omega}{3}}) = O(n^{2.792})$ time, so that in the same time, for each pair i, j, if $C[i, j] < \infty$, a witness k is returned. The $(\min, \leq^r)$-product allows us to solve all pairs minimum nondecreasing paths in three-layered graphs. We will use a slight variant of the short path-long path method given in the previous chapter. Because APNP is not a problem over a semiring, instead of giving a single source algorithm, we need to give a single midpoint algorithm for minimum nondecreasing paths:

Lemma 6.1.1 *Given an edge weighted directed graph $G = (V, E, w)$ and a vertex $s \in V$ there is an algorithm which in $O(n^2 \log n)$ time computes for all pairs of vertices i, j the minimum last edge weight of a nondecreasing path from i to j passing through s. Moreover, the algorithm can give predecessor/successor information for each node so that the best path can be recovered in time linear in its length.*

Proof. We first compute for every vertex $t \in V$ the minimum weight last edge $ml(t)$ on a nondecreasing path from t to s.

We initialize for all in-neighbors v of s, $ml(v) = w(v, s)$, and for all other v, $ml(v) = \infty$. We begin the actual algorithm by sorting all in-edges for each node v and inserting them in order of their weights in a balanced binary search tree S_v (splay tree, treap etc.). This takes $O(m \log n)$ time. We then go through the in-edges of s in nondecreasing order of their weights, calling procedure Process(e) on each of them. For $e = (v, x)$ procedure Process(e) accesses using S_v all edges into v with weights $\leq w(e)$. For every such edge (u, v) it removes (u, v) from the graph and S_v, sets $ml(u) = \min\{ml(u), ml(v)\}$ and calls Process$((u, v))$. All accesses of edges in binary search trees take $O(m \log n)$ time as each edge is deleted once it is accessed. At the end of the algorithm, $ml(v)$ is the minimum last weight of a nondecreasing path from v to s.

Once all weights $ml(u)$ are computed, for every vertex u we create a balanced binary tree T_u on $n-1$ leaves as follows. The leaves from left to right contain the edges out of s sorted by their weight (for nodes which are not out-neighbors of s the weights are ∞). Each node in the tree contains two numbers. The first number $q_1(t)$ of a tree node t represents the minimum over all weights of edges stored in the subtree rooted at t. This number can be filled in at the creation of the tree which takes $O(n \log n)$ for each vertex u. The second number $q_2(t)$ of a tree node t represents the minimum weight of a last edge on a nondecreasing path from s to u beginning at an edge stored in the subtree rooted at t. The second number is initialized as ∞ at creation.

We start another search for nondecreasing paths as follows. We first insert for every node v its out-edges in a balanced binary search tree S'_v. Then, we process the edges (s, v) out of s in *nonincreasing* sorted order of their weights by calling Process$((s, v), (s, v))$. The processing step is now as follows. For $e = (x, v)$, Process$(e, (s, j))$ uses S'_v to access all edges (v, u) with weights $\geq w(e)$. For every such edge (v, u), we remove (v, u) from S'_v and then access the leaf ℓ of T_u containing (s, j). If the current stored minimum end weight $q_2(\ell)$ is greater than $w(v, u)$, update $q_2(\ell) = w(v, u)$. Go up T_u from ℓ to the root updating minimum weights $q_2(\cdot)$ if necessary.

At the end of this process, for each node u, take the minimum last edge weight $w = ml(u)$ over all nondecreasing paths from u to s (computed in the first part of the algorithm). For each node v,

go down the binary search tree T_v from the root to the leaf holding the first edge with weight w. Look at the set R_v of all right children of the nodes on this path together with the leaf ending the path. Find the minimum out of $q_2(x)$ over all $x \in R_v$. This finds the minimum last edge weight of a nondecreasing path from u to v going through s. If this weight corresponds to some edge (j, v), one can also set the predecessor of v on the path from u to be j. The overall running time of the algorithm is $O(n^2 \log n)$. □

We again use the hitting set Lemma 5.2.1 proven for example by [106, 49, 18]. We combine Theorem 3.4.1 and Lemmas 6.1.1 and 5.2.1 into an algorithm for APNP. The running time for this algorithm is the same as our running time for APBSP (Theorem 5.2.1) because we use the same short path-long path approach and our running times of computing MaxMin and $(\min, \leq^r)$-products are the same.

Theorem 6.1.1 *There is a strongly polynomial algorithm computing APNP in* $O(n^{\frac{15+\omega}{6}} \log n) = O(n^{2.896})$ *time.*

Proof. Let ℓ be a parameter to be set later. Let A be the matrix for which $A[i,i] = -\infty$, and if $i \neq j$, $A[i,j] = w(i,j)$ for $(i,j) \in E$ and $A[i,j] = \infty$ otherwise. First set $B = A$ and then repeat $B = B \circledcirc A$ for $\ell - 1$ times. This gives for each i, j the minimum weight of a last edge on a nondecreasing path from i to j of length at most ℓ. The running time for this portion of the algorithm is $O(\ell n^{2+\frac{\omega}{3}})$.

When running the above, keep for each pair of vertices i, j, an actual minimum nondecreasing path of length $\leq \ell$ found between i and j. A subset L of these paths are of length exactly ℓ (otherwise we have found all minimum nondecreasing paths). Using Lemma 5.2.1, in $O(n^2\ell)$ time obtain a set S of $O(\frac{n \log n}{\ell})$ vertices hitting each of the $\leq n^2$ paths in L. For each $s \in S$ run the algorithm from Lemma 6.1.1 to obtain for every pair i, j the minimum last edge on a nondecreasing path from i to j going through S. This part of the algorithm takes $O(n^2\ell + \frac{n^3 \log^2 n}{\ell})$ time.

Taking the minimum of the above two path values for each pair gives the result. The running time is minimized when $\ell = n^{\frac{3-\omega}{6}} \log n$ and is $O(n^{\frac{15+\omega}{6}} \log n)$. We note that our algorithm above is strongly polynomial because the only operations we perform on the weights are comparisons. □

6.2 Single Source Nondecreasing Paths

We now consider the single source version of nondecreasing paths, SSNP. The best known algorithm for SSNP is the following folklore modification of Dijkstra's algorithm.

Algorithm `A`: Initialize $d[s] = -\infty$ and $d[v] = \infty$ for all $v \neq s$. Extract the vertex u minimizing $d[u]$, setting it as completed. Then for all uncompleted neighbors v of u, if $w(u,v) \geq d[u]$, set $d[v] = \min\{d[v], w(u,v)\}$. Recurse on the remaining uncompleted vertices until there are no more.

It is not hard to verify that at the completion of the algorithm $d[v]$ contains the minimum last weight edge on a nondecreasing path from s to v. Using Fibonacci heaps this algorithm runs in $O(m + n \log n)$.

We mention one more algorithm which resembles DFS. It takes a node v and a weight wt corresponding to an edge into v. Initially, $(v, wt) = (s, -\infty)$. For each vertex v, a value $d[v]$ is stored as before. Initially, $d[v] = \infty$ for $v \neq s$ and $d[s] = -\infty$.

Algorithm `B`(v, wt): Go through the edges (v,u) going out of v, and if $w(v,u) \geq wt$, remove (v,u) from the graph, set $d[u] = \min\{d[u], w(v,u)\}$ and call `B`$(u, w(v,u))$.

It can be verified that algorithm `B` also computes SSNP correctly. Its running time is $O(mn)$, but this can be improved by using good data structures. The bottleneck of the algorithm is going through the outgoing edges of v every time v is accessed. Yet we only need to access those edges whose weight is greater than the incoming weight. If we first store a successor search data structure at every vertex to maintain the outgoing edge weights, we could directly access only the necessary edges at each access of v. For instance, if before running the algorithm, we insert the outgoing edges of each vertex in a binary search tree achieving $O(\log n)$ time per operation, we can reduce the running time to $O(m \log n)$. [1]

Our idea is to employ the right data structures so that interleaving algorithms `A` and `B` yields a linear time algorithm for SSNP. In particular we prove the following.

Theorem 6.2.1 *In the word RAM model of computation there is a linear time algorithm for SSNP on directed weighted graphs.*

Q-Heaps. The data structure which we use to store the weights of the edges going out of each node is a successor search data structure based on the Q-Heaps of Fredman and Willard [44]. As pointed out by Thorup [90], the Q-Heaps result can be summarized as follows.

Lemma 6.2.1 *([44]) Given $O(n)$ preprocessing time and space for the construction of tables, there are data structures called Q-Heaps maintaining a family $\{S_i\}$ of word-sized integers multisets, each of size at most $O(\log^{1/4} n)$, so that each of the following operations can be done in constant time: insert x in S_i, delete x from S_i, and find the rank of x in S_i, returning the number of elements of S_i that are strictly smaller than x.*

Using a 4-level indirection (bucketing elements similarly to the AF-heaps of Fredman and Willard) and a small modification of the above we can obtain a data structure for successor queries which can dynamically maintain $O(\log n)$ elements with $O(1)$ amortized time per operation.

Corollary 6.2.1 *Given $O(n)$ preprocessing time and space for the construction of tables, there are data structures maintaining a family (U_i) of disjoint sets, each of size at most $O(\log n)$, so that each of the following operations can be done in constant amortized time: insert x in U_i, delete x from U_i, and given some integer y (not necessarily in $\bigcup_i U_i$) find $x \in U_i$ such that $x \geq y$ and x is closest to y from all elements of U_i.*

We call the data structures presented in the corollary FW-heaps (for Fredman and Willard).

Linear Time Algorithm for SSNP. This section is devoted to proving Theorem 6.2.1. Let N be the smallest power of 2 greater than n. For our linear time algorithm we first transform every edge weight $w(u, v)$ into a weight $w'(u, v) = N^2 w(u, v) + Nv + u$, where As noted in the preliminaries vertex v is identified with a unique integer from 1 to n. These new weights allow us to distinguish between edges with the same weight. The size of the integers in consideration increases only by $O(\log n)$, as in the binary representation we simply concatenate $w(u, v)$ and v. From now on we consider both w' and w weights where the w weights are assumed to have $2\lceil \log n \rceil$ zero bits concatenated at the end. The w' weights are used solely by the FW-heaps data structure.

[1] Alternatively, one can presort the outgoing edges in nonincreasing order and then access this list when a node is processed.

The idea of the algorithm is to handle *low* and *high* outdegree nodes differently. Define a *low* degree node to be one with outdegree at most $\log n$. A *high* degree node has outdegree at least $\log n$. Note that there are at most $m/\log n$ high degree nodes. We put all high degree nodes into a Fibonacci heap F with weights ∞. We insert s in F with weight $-\infty$. The source s is treated as a high degree node. Its key in F will never be decreased and it will be extracted from F first. We create FW-heaps maintaining the sets of outgoing edge weights for each low degree vertex. In particular, for a low degree vertex v we store each edge (v, u) with key $w'(v, u)$. In this way the sets stored are disjoint. The FW-heaps data structure allows us, when given any incoming edge, to extract the successor edge in the sorted order of the outgoing edges in (amortized) constant time. Since insertions also take constant amortized time, preparing these FW-heaps takes $O(m)$ time overall.

The nondecreasing paths algorithm works in stages. First an algorithm `AlgoLow` just like algorithm `B` is run on the low degree nodes starting from the neighbors of the source until no more low degree nodes can be reached via nondecreasing paths. Algorithm `AlgoLow` can visit nodes more than once, but no edge is accessed more than once since edges are removed from the graph once they are visited. During the run of the algorithm, every reached high degree node is updated in the Fibonacci heap F just like in algorithm `A`, *i.e.* if the edge taken to reach it has a smaller weight than the value stored in F, the value is replaced with this weight.

When no more low degree nodes can be reached, just as in algorithm `A`, the minimum weight high degree node is extracted from the Fibonacci heap and a new run of algorithm `AlgoLow` is started from it. We call this part of our algorithm `AlgoHigh`. This process alternates between `AlgoLow` and `AlgoHigh` until we have either gone through all edges, or the Fibonacci heap is empty.

Let M be the set of visited nodes, initially $\emptyset$. Let for a low degree node v, N_v be the subset maintained in the FW-heaps data structure storing its incident out-edges. For each $t \in V$ and weight w, $N_t.successor(w)$ returns the lexicographically smallest neighbor v of t for which $w(t, v) \geq w$ and there is no other neighbor u for which $w(t, v) > w(t, u) \geq w$. Let for any vertex v, $d[v]$ represent the current minimum weight of an edge used to get to v. For the current unvisited high degree nodes, $d[v]$ is stored in the Fibonacci heap F. Recall that at the start, $d[s] = -\infty$ and $d[v] = \infty$ for $v \neq s$. Let for every vertex $v \neq s$, $\pi[v]$ be the predecessor of v on the current best nondecreasing path from s to v. The initial value is $\pi[v] =$`null` for every v; $\pi[v]$ remains `null` if v is not reachable from s via a nondecreasing path. Pseudocode for `AlgoLow`, `AlgoHigh` and the final algorithm for SSNP is given in Figure 6.1.

Running time. We first handle the getMin calls on the Fibonacci heap, because they are the most expensive unit operations. The number of elements in F is at most $m/\log n$ so there can be at most $m/\log n$ getMin calls. Their overall cost is thus $O(\log n \times m/\log n) = O(m)$. Other than that, each edge is examined at most once in constant amortized time since FW-heaps allow each edge to be found quickly. When an edge is looked at, a constant number of other $O(1)$ time operations are performed. Thus the overall time is linear: $O(m)$. We note that if instead of FW-heaps one uses binary search trees, then the algorithm would run in $O(m \log\log n)$ time in the comparison model.

Correctness. We need to show that at the end of the algorithm, a vertex v is in M if and only if there is a nondecreasing path from s to v, and that furthermore $d[v]$ is the minimum weight of the last edge on a nondecreasing path from s to v. We prove three claims which imply the final proof of correctness.

```
AlgoLow(t,w):
   Add t to M
   If t is a low degree node:
     u_0<- N_t.successor(w)
     do until u_0 = null:
       remove (t,u_0) from the graph
       remove u_0 from N_t
       if w(t,u_0)<d[u_0] then
         d[u_0] = w(t,u_0)
         pi[u_0]=t
         AlgoLow(u_0, w(t,u_0))
       u_0<- N_t.successor(w(t,u_0))
   Else t is a high degree node:
     If w<d[t], then
       F.decreaseKey(t, w).
```

```
AlgoHigh():
   z<- F.getMin()
   for all neighbors u of z:
     remove (z,u) from graph
     if w(z,u)>=d[z], then
       if w(z,u)<d[u] then
         d[u] = w(z,u)
         pi[u]=z
         AlgoLow(u, w(z,u))

findSSNP(s):
   M={}; d[s]=-infty
   d[v]= infty for all v!=s
   PrepareDataStructures()
   While F and E are nonempty:
     AlgoHigh()
```

Figure 6.1: The `AlgoLow` and `AlgoHigh` algorithms handle low and high degree nodes respectively. The final algorithm FindSSNP finds the minimum nondecreasing paths from s.

Claim 1 *If a node v is in M, then it can be reached by a nondecreasing path, and $d[v] = w(\pi(v), v)$ is the weight of a last edge of some path from s to v.*

Proof. Follows by induction and because the value of a node in F gives a weight of an edge which is the end of a nondecreasing path from s to the node. □

The last two claims show that if a node $z \neq s$ can be reached by a nondecreasing path from s, then it is in M. For any nondecreasing path from s to z with last edge weight w, $d[z] \leq w$. Our algorithm maintains an invariant that for any $v \neq s$, $d[v] = w(\pi(v), v)$. Hence our proofs disregard $\pi(v)$. We note that using the π values one can recover actual minimum nondecreasing paths from s to any vertex t in time linear in their lengths.

Claim 2 *Consider a run of algorithm* `AlgoLow` *starting from a high degree vertex z with weight $d[z]$ and finishing when no more vertices can be reached via a nondecreasing path using only low degree vertices. If v is reachable from z via a path of only low degree vertices, then at the end of this run $d[v]$ is at most the minimum last edge weight on any nondecreasing path from z to v with first edge at least $d[z]$.*

Proof. Let us first prove the claim for low degree vertices. This setting is almost the same as running algorithm `B` with z as a source and ignoring edges out of z with weights smaller than $d[z]$. The only real difference is that the algorithm might never recurse on some low degree vertices v reachable from z via such a nondecreasing path because their $d[v]$ values are already smaller or equal to the last edge weight of any nondecreasing path from z to v with first weight $\geq d[z]$. However, this can only happen if there is a path from s to v which is at least as good as any nondecreasing path from z to v with first weight $\geq d[z]$. Hence v need not be considered in the search. Furthermore, any low degree paths going through such a v would have been considered when the best path to v

was traversed. Hence the claim holds for all low degree vertices. Because the $d[v]$ values are also updated when a high degree vertex can be reached from z, the claim also holds for high degree vertices reachable from z via a low degree nondecreasing path with first weight at least $d[z]$. □

The remainder of the argument bears similarities to the proof of correctness for Dijkstra's SSSP algorithm.

Claim 3 *At any point of our algorithm let S contain all high degree vertices which have already been extracted from F. For each high degree node $v \in S$ the value of $d[v]$ when v is extracted from F is equal to the minimum last edge weight of a nondecreasing path from s to v. Furthermore, for all high degree nodes $v \notin S$, $d[v]$ is the minimum last edge weight of a nondecreasing path using only vertices in S and low degree vertices in M.*

Proof. We do induction on the size of S. The base case is when $|S| = 1$. Then S consists only of s, and $d[s]$ is $-\infty$. Inductive step: suppose the statement holds for $|S| < k$. After the $k-1$st high degree node z is added to S, `AlgoLow` is called on all of z's neighbors whose edge weight is greater than $d[z]$. Until the kth high degree node is to be added, all low degree nodes reachable from z by a nondecreasing path with edge weights greater than $d[z]$ are added to M. Their $d[v]$ value is at most the the best value for paths first using vertices in S and low degree nodes in M, then going through z and finally using only low degree nodes. Moreover, for every high degree $v \notin S$, if there is a nondecreasing path from s to v using only nodes from S and low degree nodes from M, going through z and using only low degree nodes, then the ending edge weight of that path is at least $d[v]$.

Suppose now that there is a nondecreasing path P from s to v using vertices in S, low degree vertices in M and z such that there is some high degree node (from S) between z and v on P. Let $y \in S$ be the last such high degree node. By the induction hypothesis when y was extracted from F it had the minimum value Y of a last edge on any nondecreasing path from s to a high degree node. After y was extracted, all low degree nodes reachable from it with nondecreasing paths with edge weights $\geq Y$ were explored (either by `AlgoLow` on y or by a previous run of `AlgoLow`). In particular, if v has a low degree predecessor in P, then `AlgoLow`(v, wt) was run, where wt is at most the minimum end weight of a nondecreasing path from y with weights $\geq Y$. Otherwise, y is v's predecessor in P. Either way, there is a path from s to v using only high degree nodes in $S \setminus \{z\}$ and low degree nodes in M of last edge weight at most that of P. And hence P does not need to be considered and $d[v]$ is indeed the minimum last edge weight of a nondecreasing path using only vertices in S and low degree vertices in M.

Now we need to show that when the kth high degree node v is added to S, $d[v]$ is equal to the minimum last edge weight of a nondecreasing path from s to v. Suppose not and let $Q := s \to y_1 \to \ldots \to y_{k-1} \to v$ be a minimum nondecreasing path from s to v that contains high degree vertices not in S. Let y_i be the first high degree vertex on Q which is not in S. $d[y_i]$ is the minimum last edge weight of a nondecreasing path using high degree vertices only from S. Because y_i appears before v in Q, we must have that $d[y_i] \leq w(y_{i-1}, y_i) \leq w(y_{k-1}, v)$. But since v is to be extracted from F before y_i, $d[v] \leq d[y_i]$ and hence $d[v] \leq w(y_{k-1}, v)$ and there is a path Q' from s to v using high degree vertices only from S such that the last edge weight of Q' is at most that of Q, making $d[v]$ equal to the minimum value of the last edge weight of a nondecreasing path from s to v. The induction step is completed. □

Claim 3 ensures that the values of the high degree nodes are minimized. By Claim 2 all values of low degree nodes reachable by nondecreasing paths are also minimized. Inductively, at the end of the algorithm all $d[\cdot]$ and $\pi[\cdot]$ values are correct.

6.3 Cheapest Nondecreasing Paths

Consider a traveler who wants to obtain a valid itinerary from one city to another but their major objective is to minimize their expenses. The travel agency has information about all flights – departure and arrival times and a price for each segment. In the graph context, one is given a directed graph $G = (V, E)$ with two weight functions $t : E \to \mathbb{R} \cup \{-\infty, \infty\}$ and $p : E \to \mathbb{R}^+ \cup \{0, \infty\}$. One wishes to find a path from a source s to a destination t (or to all possible destinations t) which minimizes the sum of $p(e)$ weights of edges e on the path over all paths with nondecreasing $t(e)$ weights. More formally:

Definition 6.3.1 (Cheapest Nondecreasing Path) *Given a directed graph $G = (V, E, t, p)$ with weights $t : E \to \mathbb{R} \cup \{-\infty, \infty\}$ and $p : E \to \mathbb{R}^+ \cup \{0, \infty\}$, a path $P = v_1 \to v_2 \dots v_{k-1} \dots v_k$ is a* nondecreasing *path between nodes s and t if*

- *$(v_i, v_{i+1}) \in E$ for all $i < k$, $s = v_1$ and $t = v_k$*
- *$t(v_i, v_{i+1}) \leq t(v_{i+1}, v_{i+2})$ for all $i = 1, \dots, k-2$.*

P is a cheapest *nondecreasing path between s and t if*

- *P is a nondecreasing path from s to t and*
- *for all other nondecreasing paths $P' = v'_1 \to v'_2 \dots v'_{k'-1} \dots v'_{k'}$ from s to t,*

$$\sum_{i=1}^{k-1} p(v_i, v_{i+1}) \leq \sum_{i=1}^{k'-1} p(v'_i, v'_{i+1}).$$

We note that computing shortest paths in a graph $G = (V, E, w)$ can be easily reduced to computing cheapest nondecreasing paths in the graph $G' = (V, E, t, p)$: just set $p = w$ and $t(e) = 1$ for all $e \in E$.

Proposition 6.3.1 *If SSCNP in an n node graph can be computed in $O(T(n))$ time, then so can SSSP in an n node graph. Likewise, if APCNP in an n node graph can be computed in $O(T(n))$ time, then so can APSP in an n node graph.*

We will consider the single source cheapest nondecreasing paths problem (SSCNP): given a graph $G = (V, E, p, t)$ as above and a source node $s \in V$, compute for all vertices $t \in V$ a cheapest nondecreasing path from s to t, if one exists.

The following simple algorithm was communicated to us by Tarjan [89].

Theorem 6.3.1 ([89]) *SSCNP in an m-edge, n-node graph can be computed in $O(m \log n)$ time.*

Proof. Sort the edges by weight $t(\cdot)$ in $O(m \log n)$ time. The sorted list of edges consists of *buckets* of edges of the same weight. That is, the list looks like $B_1, \dots, B_{m'}$ with $m' \leq m$ such that each B_i is a list of edges of the same weight w_i and for $i < j$, $w_i < w_j$.

We note that for any nondecreasing path in the graph there is some way to reorder the edges within the buckets B_i so that the nondecreasing path is a subsequence of the sorted list. This suggests the following approach. We maintain a value $d[u]$ for every node u which is the current

minimum price sum of a nondecreasing path from s to u. Initially, $d[u] = \infty$ for all $u \neq s$ and $d[s] = 0$.

Beginning with $i = 1$, process bucket B_i as follows. For every edge $(u, v) \in B_i$, insert u into a priority queue Q_i, initially empty. The priority of u in B_i is $d[u]$. While Q_i is nonempty, extract the minimum priority node u from Q_i and set $d[v] = \min\{d[v], d[u] + p(u, v)\}$ for all edges $(u, v) \in B_i$ (if $v \in Q_i$, this also updates Q_i). When Q_i is emptied, move on to B_{i+1}, if any.

The running time of the algorithm is $O(m \log n)$ because of the original sorting and as a priority queue is only updated if an edge is accessed, and each edge is accessed at most once. □

The algorithm can be adapted to handle negative $p(\cdot)$ weights and to detect negative cycles.

Cheapest Increasing Paths in Low Degree Graphs. We consider a restriction of the cheapest nondecreasing paths problem, the cheapest increasing paths problem. Instead of minimizing the price sum over nondecreasing paths, one minimizes it over *increasing* paths. This restriction makes single source version of the problem, SSCIP, seemingly easier to solve than SSCNP. For instance Tarjan's approach from Theorem 6.3.1 becomes significantly simpler. In particular, one does not need to use Dijkstra's algorithm and sorting is practically all that is necessary to obtain an $O(m \log n)$ time algorithm. We show that in fact with some bookkeeping one can obtain a runtime which varies with the maximum degree of the graph. When the graph has constant degree the algorithm runs in linear time. Such a runtime is not known for SSSP and hence for SSCNP (by Proposition 6.3.1).

Theorem 6.3.2 *Given a directed graph $G = (V, E, p, t)$ with maximum degree Δ, n nodes and m edges, and a source $s \in V$, one can compute for all nodes $v \in V$ a cheapest increasing path from s to v in $O(m \log \Delta)$ time.*

Proof. First we do some preprocessing on the graph. We compute in $O(m \log \Delta)$ time single source increasing paths from s using a slight modification of algorithm `B` from the beginning of the chapter. Let the computed last edge weights be $le(v)$ for nodes $v \in V$. Then remove from the graph all edges (u, v) with $w(u, v) \leq le(u)$. After this removal all edges in the graph are part of some increasing path from s, and the edges that were removed were not part of any increasing path from s.

For every node $u \in V$, sort the edges incident on u in nondecreasing order of their weights. The sorted list of edges incident on u consists of alternating groups of edges, some from the incoming edges, and some from the outgoing edges, *i.e.* this ordered list N_u looks like

$$In_1(u), Out_1(u), In_2(u), Out_2(u), \ldots,$$

where $In_i(u)$ is a list of edges going into u and $Out_i(u)$ is a list of edges coming out of u. Up to some $K > 1$, for all $1 \leq i < K$, $Out_i(u)$ is nonempty, and for all $1 < i < K$, $In_i(u)$ is nonempty, and for all $j > K$, $In_j(u) = \emptyset$, and for all $j \leq K$, $Out_j(u) = \emptyset$.

When creating $In_i(u), Out_i(u)$ we break ties in the sorted order as follows: edges coming into u with weight w are considered to be larger than edges coming out of u of weight w. In this way if $(x, u) \in In_i(u)$ and $(u, y) \in Out_j(u)$ with $j \geq i$, then $x \to u \to y$ is an increasing path. Moreover, all increasing paths $x \to u \to y$ have the property that if $(x, u) \in In_i(u)$ and $(u, y) \in Out_j(u)$ then $j \geq i$.

For accounting purposes we maintain a few values for every $i = 1, \ldots, K$:

- a counter $Count_i(u)$ initially 0
  - $Count_i(u)$ will equal the number of processed edges (x, u) with $(x, u) \in In_i(u)$
- a bit $Done_i(u)$ initially 0; also for completeness, set $Done_0(u) = 1$.
  - $Done_i(u) = 1$ iff $Count_k(u) = |In_k(u)|$ for all $k \leq i$.
- $d_i(u)$ which is the cheapest price via a path with last edge in $In_k(u)$ with $k \leq i$; for completeness $d_i(s) = 0$ for all i.
  - if $Done_i(u) = 1$ then all edges in $Out_i(u)$ can be processed using $d_i(u)$ as the current price sum to u.

We now give some pseudocode for the edge processing algorithm:

Process_edge (v, u):
If (v, u) is $In_i(u)$ and $Out_t(v)$:
  $d_i(u) = \min(d_i(u), d_t(v) + p(v, u))$
  $Count_i(u) += 1$
  If $Count_i(u) = |In_i(u)|$ and $Done_{i-1}(u) = 1$
    Set $Done_i(u) = 1, d_i(u) = \min(d_{i-1}(u), d_i(u))$
    for all $e \in Out_i(u)$, **Process_edge**(e)
    for all $k > i$ while $Count_k(u) = |In_k(u)|$
      Set $Done_k(u) = 1, d_k(u) = \min(d_{k-1}(u), d_k(u))$
      for all $e \in Out_k(u)$, **Process_edge**(e)

To continue the algorithm for SSCIP after the edge sorting phase, go through the edges e going out of s in nondecreasing order and run **Process_edge**(e).

First we show that every edge in the graph is processed at most once. Let (u, v) be the first edge to be processed a second time. Then at least two calls **Process_edge**(x_1, u) and **Process_edge**(x_2, u) for $x_1 \neq x_2$ must have caused a call to **Process_edge**(u, v).

Suppose **Process_edge**(x_1, u) was called first. Let $(u, v) \in Out_j(u)$. In order for **Process_edge**(u, v) to have been called, $Count_i(u) = |In_i(u)|$ for all $i \leq j$. Since $Count_i(u)$ is only incremented when an edge from $In_i(u)$ is processed and since every edge smaller than (u, v) was processed at most once, this means that all edges from $In_i(u)$ for all $i \leq j$ must have been processed. Since (x_2, u) was processed after (x_1, u), then $(x_2, u) \in In_t(u)$ for $t > j$. This contradicts the assumption that **Process_edge**(x_2, u) caused a call to **Process_edge**(u, v).

Since every edge is processed at most once, the running time of this portion of the algorithm is $O(m)$. The major overhead in the overall algorithm is the original sorting of the edges which makes the running time $O(m \log \Delta)$.

Now we show that every edge in the graph is processed at least once. This is done again by contradiction. All edges (s, v) are processed. Let (u, v) for $u \neq s$ be the smallest edge which was not processed. Let $(u, v) \in Out_j(u)$. Then all edges $(x, u) \in In_i(u)$ with $i \leq j$ must have been processed since they are all smaller than (u, v). Also, there must be at least one such edge (x, u) since (u, v) is part of an increasing path from s to v. Let (x, u) be the largest edge smaller than

(u, v). Edge (x, u) has been processed. Whenever **Process_edge**(x, u) was called, $Done_j(u)$ must have been set and hence (u, v) was processed. Contradiction.

Lastly, we show that the algorithm computes the correct minimum price sums for all nodes. First note that any $d_i(u)$ which is ever computed is a price sum of some increasing path from s to u. This is by induction and since we only process an edge (u, v) if some edge into u smaller than (u, v) was processed. Consider node $v \in V$, and let $P = s \to u_1 \to \ldots \to u_k \to v$ be a cheapest increasing path from s to v. Let $u_x \in \{s = u_0, u_1, \ldots, u_k, v = u_{k+1}\}$ be the last node on the path such that when (u_x, u_{x+1}) is processed, $d_i(u_x)$ (for $(u_{x-1}, u_x) \in In_i(u_x)$) is at most the price sum of path $s \to u_1 \to u_x$. Such a node u_x exists as $d_i(s) = 0$ for all i. If $u_x = v$ we are done. Otherwise, when edge (u_x, u_{x+1}) is processed, the value of $d_j(u_{x+1})$ for (j such that $(u_x, u_{x+1}) \in In_j(u_{x+1})$) is set to $\min(d_j(u_{x+1}), d_t(u_x) + p(u_x, u_{x+1}))$, where t is such that $(u_x, u_{x+1}) \in Out_t(u_x)$. We have $d_t(u_x) \leq d_i(u_x)$ as $t \geq i$ and hence

$$d_j(u_{x+1}) \leq d_i(u_x) + p(u_x, u_{x+1}) \leq \sum_{y=0}^{x} p(u_y, u_{y+1}).$$

This contradicts the maximality of x. Hence $d_n(v)$ (the final price sum) is at most the price sum of path P, and thus equals the price sum of P and is the cheapest price of an increasing path from s to v. □

All Pairs Cheapest Increasing Paths and All Pairs Shortest Paths. We show that all pairs cheapest increasing Paths (APCIP) is at least as hard as APSP. In particular, computing the distance product of two $n \times n$ matrices A and B can be reduced to APCIP in an $3n$ node directed graph as follows. Create a 3-layered graph G on $3n$ nodes and node partitions I, J and K. For every $i \in I$, $j \in J$ set $p(i, j) = A[i, j]$ and $t(i, j) = 1$. For $j \in J$, $k \in K$, set $p(j, k) = B[j, k]$ and $t(j, k) = 2$. Even when G is undirected the only increasing paths in the graph are of the form $i \to j \to k$ with $i \in I, j \in J, k \in K$, and all such paths are increasing. Hence APCIP actually computes the distance product of A and B, as for every $i \in I$, $k \in K$, the cheapest increasing path between i and k has price sum $\min_j(A[i, j] + B[j, k])$. We obtain the following.

Theorem 6.3.3 *If APCIP in an undirected graph on n nodes can be computed in $O(T(n))$ time, then APSP in an arbitrary graph on n nodes can be computed in $O(T(n))$ time.*

Chapter 7

Combinatorial Algorithms for Path Problems in Sparse Graphs

A large collection of graph problems in the literature admit essentially two solutions: an *algebraic* approach and a *combinatorial* approach. Algebraic algorithms rely on the theoretical efficacy of fast matrix multiplication over a ring, and reduce the problem to a small number of matrix multiplications. These algorithms achieve unbelievably good theoretical guarantees, but can be impractical to implement given the large overhead of fast matrix multiplication. Combinatorial algorithms rely on the efficient preprocessing of small subproblems. Their theoretical guarantees are typically worse, but they usually lead to more practical improvements. Combinatorial approaches are also interesting in that they have the capability to tackle problems that seem to be currently out of the reach of fast matrix multiplication. For example, many sparse graph problems are not known be solvable quickly with fast matrix multiplication, but a combinatorial approach can give asymptotic improvements. (Examples are below.)

7.1 A Combinatorial Approach for Sparse Graphs

In this chapter we introduce a new combinatorial approach for path problems in sparse graphs. All results presented here, together with some others, appear in [10]. We first present a new combinatorial method for preprocessing an $n \times n$ dense Boolean matrix A in $O(n^{2+\varepsilon})$ time (for any $\varepsilon > 0$) so that sparse vector multiplications with A can be done faster, while matrix updates are not too expensive to handle. In particular,

- for a vector v with t nonzeros, $A \cdot v$ can be computed in $O(\frac{n}{\log n}(t/\kappa + n/\ell))$ time, where ℓ and κ are parameters satisfying $\binom{\ell}{\kappa} \leq n^{\varepsilon}$, and
- row and/or column updates to the matrix can be performed in $O(n^{1+\varepsilon})$ time.

The matrix-vector multiplication can actually return a vector w of *minimum witnesses*; that is, $w[i] = k$ iff k is the minimum index satisfying $A[i,k] \cdot v[k] \neq 0$. The data structure is simple, does not use devious "word tricks" or hide any large constants, and can be implemented on a pointer machine.[1] We apply our data structure to three fundamental graph problems: transitive closure,

[1]When implemented on the w-word RAM, the multiplication operation runs in $O(\frac{n}{w}(t/k + n/\ell))$. In fact, all of the combinatorial algorithms mentioned in this chapter can be implemented on a w-word RAM in $O(T(n)(\log n)/w)$

all pairs shortest paths, and all pairs maximum weight triangles (as in Chapter 4). All three are known to be solvable in $n^{3-\delta}$ time for some $\delta > 0$, but the algorithms are algebraic and do not exploit the potential sparsity of graphs. With the right settings of the parameters ℓ and κ, our data structure can be applied to all the above problems, giving the best runtime bounds for *sparse problems* to date.

Transitive Closure: We have a directed graph on n nodes and m edges, and wish to find all pairs of nodes (u, v) whether there is a path from u to v in the graph. Transitive closure has myriad applications and a long history of ideas. The best known theoretical algorithms use $O(M(n))$ time [39, 67] where $M(n)$ is the complexity of $n \times n$ Boolean matrix product, and $O(mn/\log n + n^2)$ time [21, 16]. Algebraic matrix multiplication implies an $O(n^\omega)$ algorithm, where $\omega < 2.376$ [28], and combinatorial matrix multiplication gives an $O(n^3/\log^2 n)$ runtime [6, 79, 100]. Our data structure can be used to implement transitive closure in $O(mn(\log(\frac{n^2}{m})/\log^2 n) + n^2)$ time. This constitutes the first combinatorial improvement on the bounds of $O(n^3/\log^2 n)$ and $O(mn/\log n + n^2)$ that follow from Four Russians preprocessing. In fact, it provides a smooth transition from the running time for dense graphs ($O(n^3/\log^2 n)$) to the running time for graphs with $m = O(n^{2-\varepsilon})$ for $\varepsilon > 0$ ($O(mn/\log n + n^2)$).

All Pairs Shortest Paths (APSP): We want to construct a representation of a given graph, so that for any pair of nodes, a shortest path between the pair can be efficiently obtained from the representation. The work on APSP is deep and vast; here we focus on the simplest case where the graph is *undirected* and *unweighted*. For this case, Galil and Margalit [50] and Seidel [82] gave $O(n^\omega)$ time algorithms. These algorithms do not improve on the simple $O(mn + n^2)$ algorithm (using BFS) when $m = o(n^{\omega-1})$. The first improvement over $O(mn)$ was given by Feder and Motwani [38] who gave an $O(mn\log(\frac{n^2}{m})/\log n)$ time algorithm. Recently, Chan [17] presented new algorithms that take $O(n^{2+\varepsilon} + mn/\log n)$ time for any $\varepsilon > 0$. We show that APSP on undirected unweighted graphs can be computed in $O(mn\log(\frac{n^2}{m})/\log^2 n)$ time, again providing a smooth transition from the known combinatorial runtimes of $O(n^3/\log^2 n)$ ([38, 17]) and $O(n^{2+\varepsilon} + mn/\log n)$ time. Our algorithm modifies Chan's $O(mn/\log n + n^2 \log n)$ time solution, implementing its most time-consuming procedure efficiently using our data structure.

All Pairs Weighted Triangles: As in Chapter 4, we consider the problem of finding maximum node-weighted triangles in a graph. Here we have a directed graph with an arbitrary weight function w on the nodes. We wish to compute, for all pairs of nodes v_1 and v_2, a node v_3 that such that (v_1, v_3, v_2, v_1) is a cycle and $\sum_i w(v_i)$ is minimized or maximized. The problem has applications in data mining and pattern matching. In Chapter chap:tria we uncovered interesting algebraic algorithms for this problem (also see [98]), the current best being $O(n^{2.575})$, but again it is somewhat impractical, relying on fast rectangular matrix multiplication. (We note that the problem of finding a *single* minimum or maximum weight triangle has been shown to be solvable in $n^{\omega+o(1)}$ time [30].) The proof of the result in Chapter 4 also implies an $O(mn/\log n)$ algorithm. Our data structure lets us solve the problem in $O(mn\log(\frac{n^2}{m})/\log^2 n)$ time.

time, where $T(n)$ is the runtime stated.

7.2 On the Optimality of our Algorithms

We have claimed that all the above problems can be solved in $O(mn \log(n^2/m)/\log^2 n)$ time. How does this new runtime expression compare to previous work? It is easier to see the impact when we let $m = n^2/s$ for a parameter s. Then

$$\frac{mn \log(n^2/m)}{\log^2 n} = \frac{n^3}{\log^2 n} \cdot \frac{\log s}{s} \tag{7.1}$$

Therefore, our algorithms yield asymptotic improvements on "medium density" graphs, where the number of edges is in the range $n^{2-o(1)} \leq m \leq o(n^2)$.

At first glance, such an improvement may appear small. We stress that our algorithms have essentially reached the best that one can do for these problems, without resorting to fast matrix multiplication or achieving a major breakthrough in combinatorial matrix algorithms. As all of the above problems can be used to simulate Boolean matrix multiplication, (7.1) implies that an $O(mn\frac{\log(n^2/m)}{f(n)\log^2 n})$ algorithm for any of the above problems and any unbounded function $f(n)$ would entail an asymptotic improvement on combinatorial Boolean matrix multiplication, a longstanding open problem. Note that an $o(mn/\log n)$ combinatorial algorithm does not imply such a breakthrough: let $m = n^2/s$ and observe $o(mn/\log n) = o(n^3/(s \log n))$; such an algorithm is still slow on sufficiently dense matrices.

7.3 Related Work

Closely related to our work is Feder and Motwani's ingenious method for compressing sparse graphs, introduced in STOC'91. In the language of Boolean matrices, their method runs in $\tilde{O}(mn^{1-\varepsilon})$ time and decomposes an $n \times n$ matrix A with m nonzeros into an expression of the form $A = (A_1 * A_2) \vee A_3$, where $*$ and $\vee$ are matrix product and pointwise-OR, A_1 is $n \times mn^{\varepsilon}$, A_2 is $mn^{\varepsilon} \times n$, A_1 and A_2 have $O(\frac{m \log(n^2/m)}{\log n})$ nonzeros, and A_3 has $O(n^{1+\varepsilon})$ nonzeros. Such a decomposition has many applications to speeding up algorithms.

While a $(\log n)/\log(n^2/m)$ factor of savings crops up in both approaches, our approach is markedly different from graph compression; in fact it seems orthogonal. From the above viewpoint, Feder-Motwani's graph compression technique can be seen as a method for preprocessing a sparse Boolean matrix so that multiplications of it with arbitrary vectors can be done in $O\left(\frac{m \log(n^2/m)}{\log n}\right)$ time. In contrast, our method preprocesses an *arbitrary matrix* so that its products with *sparse vectors* are faster, and updates to the matrix are not prohibitive. This sort of preprocessing leads to a Feder-Motwani style improvement for a new set of problems. It is especially applicable to problems in which an initially sparse graph is augmented and becomes dense over time, such as transitive closure.

Our data structure is related to one given previously by Williams [100], who showed how to preprocess a matrix over a constant-sized semiring in $O(n^{2+\varepsilon})$ time so that subsequent vector multiplications can be performed in $O(n^2/\log^2 n)$ time. Ours is a significant improvement over this data structure in two ways: (1) the runtime of multiplication now varies with the sparsity of the vector (and is never worse than $O(n^2/\log^2 n)$), and (2) our data structure also returns minimum witnesses for the multiplication. Both of these are non-trivial augmentations that lead to new applications.

7.4 Preliminaries and Notation

Define $H(x) = x\log_2(1/x) + (1-x)\log_2(1/(1-x))$. H is often called the binary entropy function. All logarithms are assumed to be base two. We define $\delta_G(s,v)$ to be the distance in graph G from node s to node v. We assume G is always weakly connected, so that $m \geq n-1$.

We denote the typical Boolean product of two matrices A and B by $A \cdot B$. For two Boolean vectors u and v, let $u \wedge v$ and $u \vee v$ denote the componentwise AND and OR of u and v respectively; let $\neg v$ be the componentwise NOT on v.

A *minimum witness* vector for the product of a Boolean matrix A with a Boolean vector v is a vector w such that $w[i] = 0$ if $\vee_j(A[i][j] \cdot v[j]) = 0$, and if $\vee_j(A[i][j] \cdot v[j]) = 1$, $w[i]$ is the minimum index j such that $A[i][j] \cdot v[j] = 1$.

7.5 Combinatorial Matrix Products With Sparse Vectors

We begin with a data structure which after preprocessing stores a matrix while allowing efficient matrix-vector product queries and column updates to the matrix. On a matrix-vector product query, the data structure not only returns the resulting product matrix, but also a minimum witness vector for the product.

Theorem 7.5.1 *Let B be a $d \times n$ Boolean matrix. Let $\ell \geq 1$ and $\kappa < \ell$ be parameters. Then one can create a data structure with $O(\frac{dn\kappa}{\ell} \cdot \sum_{b=1}^{\kappa} \binom{\ell}{b})$ preprocessing time so that the following operations are supported:*

- *given any $n \times 1$ binary vector r, output $B \cdot r$ and a $d \times 1$ vector w of* minimum witnesses *for $B \cdot r$ in $O(d\log^2 n + \frac{d}{\log n}\left(\frac{n}{\ell} + \frac{m_r}{\kappa}\right))$ time, where m_r is the number of nonzeros in r;*
- *replace any column of B by a new column in $O(d\kappa \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.*

The result can be made to work on a pointer machine as in [100]. Observe that the naïve algorithm for $B \cdot r$ that simulates $\Theta(\log n)$ word operations on a pointer machine would take $O(\frac{dm_r}{\log n})$ time, so the above runtime gives a factor of κ savings provided that ℓ is sufficiently large. The result can also be generalized to handle products over any fixed size semiring, similar to [100].

Proof of Theorem 7.5.1. Let $0 < \varepsilon < 1$ be a sufficiently small constant in the following. Set $d' = \lceil \frac{d}{\lceil \varepsilon \log n \rceil} \rceil$ and $n' = \lceil n/\ell \rceil$. To preprocess B, we divide it into block submatrices of at most $\lceil \varepsilon \log n \rceil$ rows and ℓ columns each, writing B_{ji} to denote the j,i submatrix, so

$$B = \begin{bmatrix} B_{11} & \dots & B_{1n'} \\ \vdots & \ddots & \vdots \\ B_{d'1} & \dots & B_{d'n'} \end{bmatrix}.$$

Note that entries from the kth column of B are contained in the submatrices $B_{j\lceil k/\ell \rceil}$ for $j = 1, \dots, d'$, and the entries from kth row are in $B_{\lceil k/\ell \rceil i}$ for $i = 1, \dots, n'$. For simplicity, from now on we omit the ceilings around $\varepsilon \log n$.

For every $j = 1, \dots, d'$, $i = 1, \dots, n'$ and every ℓ length vector v with at most κ nonzeros, precompute the product $B_{ji} \cdot v$, and a minimum witness vector w which is defined as: for all

$k = 1, \ldots \varepsilon \log n$,

$$w[k] = \begin{cases} 0 & \text{if } (B_{ji} \cdot v)[k] = 0 \\ (i-1)\ell + w' & \text{if } B_{ji}[k][w'] \cdot v[w'] = 1 \text{ and } \forall w'' < w', B_{ji}[k][w''] \cdot v[w''] = 0. \end{cases}$$

Store the results in a look-up table. Intuitively, w stores the minimum witnesses for $B_{ji} \cdot v$ with their indices in $[n]$, that is, as if B_{ji} is construed as an $n \times n$ matrix which is zero everywhere except in the (j, i) subblock which is equal to B_{ji}, and v is construed as a length n vector which is nonzero only in its ith block which equals v. This product and witness computation on B_{ji} and v takes $O(\kappa\varepsilon \log n)$ time. There are at most $\sum_{b=1}^{\kappa} \binom{\ell}{b}$ such vectors v, and hence this precomputation takes $O(\kappa \log n \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time for fixed B_{ji}. Over all j, i the preprocessing takes $O(\frac{dn}{\ell \log n} \cdot \kappa \log n \sum_{b=1}^{\kappa} \binom{\ell}{b}) = O(\frac{dn\kappa}{\ell} \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.

Suppose we want to modify column k of B in this representation. This requires recomputing $B_{j\lceil k/\ell \rceil} \cdot v$ and the witness vector for this product, for all $j = 1, \ldots, n'$ and for all length ℓ vectors v with at most κ nonzeros. Thus a column update takes only $O(d\kappa \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.

Now we describe how to compute $B \cdot r$ and its minimum witnesses. Let m_r be the number of nonzeros in r. We write $r = [r_1 \ \cdots \ r_{n'}]^T$ where each r_i is a vector of length ℓ. Let m_{ri} be the number of nonzeros in r_i.

For each $i = 1, \ldots, n'$, we decompose r_i into a union of at most $\lceil m_{ri}/\kappa \rceil$ disjoint vectors r_{ip} of length ℓ and at most κ nonzeros, so that r_{i1} contains the first κ nonzeros of r_i, r_{i2} the next κ, and so on, and $r_i = \vee_p r_{ip}$. Then, for each $p = 1, \ldots, \lceil m_{ri}/\kappa \rceil$, r_{ip} has nonzeros with larger indices than all $r_{ip'}$ with $p' < p$, *i.e.* if $r_{ip}[q] = 1$ for some q, then for all $p' < p$ and $q' \geq q$, $r_{ip'}[q'] = 0$.

For $j = 1, \ldots, d'$, let $B^j = [B_{j1} \ldots B_{jn'}]$. We shall compute $v_j = B^j \cdot r$ separately for each j and then combine the results as $v = [v_1, \ldots, v_{d'}]^T$. Fix $j \in [d']$. Initially, set v_j and w_j to be the all-zeros vector of length $\varepsilon \log n$. The vector w_j shall contain minimum witnesses for $B^j \cdot r$.

For each $i = 1, \ldots, n'$ in increasing order, consider r_i. In increasing order for each $p = 1, \ldots, \lceil m_{ri}/\kappa \rceil$, process r_{ip} as follows. Look up $v = B_{ji} \cdot r_{ip}$ and its witness vector w. Compute $y = v \wedge \neg v_j$ and then set $v_j = v \vee v_j$. This takes $O(1)$ time. Vector y has nonzeros in exactly those coordinates h for which the minimum witness of $(B^j \cdot r)[h]$ is a minimum witness of $(B_{ji} \cdot r_{ip})[h]$ but not of any $(B_{ji'} \cdot r_{i'p'})[h]$ with $i' < i$ or $i' = i$ and $p' < p$. In this situation we say that the minimum witness is in r_{ip}. In particular, if $y \neq 0$, r_{ip} contains some minimum witnesses for $B^j \cdot r$ and the witness vector w_j for $B^j \cdot r$ needs to be updated. Then, for each $q = 1, \ldots, \varepsilon \log n$, if $y[q] = 1$, set $w_j[q] = w[q]$. This ensures that after all i, p iterations, $v = \bigvee_{i,p} (B_{ji} \cdot r_{ip}) = B^j \cdot r$ and w_j is the product's minimum witness vector. Finally, we output $B \cdot r = [v_1 \cdots v_{d'}]^T$ and $w = [w_1 \ldots w_{d'}]$. Updating w_j can happen at most $\varepsilon \log n$ times, because each $w_j[q]$ is set at most once for $q = 1, \ldots, \varepsilon \log n$. Each individual update takes $O(\log^2 n)$ time. Hence, over all j, the minimum witness computation takes $O(d \log^2 n)$. Computing $v_j = B^j \cdot r$ for a fixed j takes $O(\sum_{i=1}^{n'} \lceil m_{ri}/\kappa \rceil) \leq O(\sum_{i=1}^{n'} (1 + m_{ri}/\kappa))$ time. Over all $j = 1, \ldots, d/(\varepsilon \log n)$, the computation of $B \cdot r$ takes asymptotically

$$\frac{d}{\log n} \sum_{i=1}^{n/\ell} \left(1 + \frac{m_{ri}}{\kappa}\right) \leq \frac{d}{\log n} \left(\frac{n}{\ell} + \frac{m_r}{\kappa}\right).$$

In total, the running time is $O(d \log^2 n + \frac{d}{\log n}(\frac{n}{\ell} + \frac{m_r}{\kappa}))$. □

Let us demonstrate what the data structure performance looks like with a particular setting of the parameters ℓ and k. From Jensen's inequality we have:

Fact 2 *$H(a/b) \leq 2a/b \log(b/a)$, for $b \geq 2a$.*

Corollary 7.5.1 *Given a parameter m and $0 < \varepsilon < 1$, any $d \times n$ Boolean matrix A can be preprocessed in $O(dn^{1+\varepsilon})$ time, so that every subsequent computation of AB can be determined in $O(de \log^2 n + \frac{md \log(ne/m)}{\log^2 n})$ time, for any $n \times e$ Boolean matrix B with at most m nonzeros.*

Proof. When $m \geq \frac{en}{2}$, the runtime in the theorem is $\Omega(end/\log^2 n)$, and can be achieved via Four Russians processing. Consider $m \leq \frac{en}{2}$. Apply Theorem 7.5.1 with $\ell = \varepsilon \frac{en}{m} \cdot (\log n)/\log(en/m)$, and $\kappa = \varepsilon(\log n)/\log(en/m)$. Then by Fact 2, we can show that $\binom{\ell}{\kappa} \leq n^{\varepsilon}$. Hence the preprocessing step takes $O(dn^{1+\varepsilon}m/(ne)) = O(dn^{1+\varepsilon})$ time. Matrix-vector multiplication with a vector of m_i nonzeros takes $O\left(d \log^2 n + \frac{d}{\log n}\left(\frac{m \log(en/m)}{e \log n} + \frac{m_i \log(en/m)}{\log n}\right)\right)$ time. If m_i is the number of nonzeros in the ith column of B, as $\sum_i m_i = m$, the full matrix product $A \cdot B$ can be done in $O\left(de \log^2 n + \frac{md \log(en/m)}{\log^2 n}\right)$ time. □

It follows that Boolean matrix multiplication for $n \times n$ matrices can be done in $O(n^{2+\varepsilon} + mn \log \frac{n^2}{m}/\log^2 n)$ time, provided that at least one of the matrices has only m nonzeros. We note that such a result could also be obtained by construing the sparse matrix as a bipartite graph, applying Feder-Motwani's graph compression to represent the sparse matrix as a product of two sparser matrices (plus a matrix with $n^{2-\delta}$ nonzeros), then using an $O(mn/\log n)$ Boolean matrix multiplication algorithm on the resulting matrices. However, given the complexity of this procedure (especially the graph compression, which is involved) we believe that our method is more practical. Theorem 7.5.1 also leads to two other new results almost immediately.

Minimum Witnesses for Matrix Multiplication. The *minimum witness* product of two $n \times n$ Boolean matrices A and B is the $n \times n$ matrix C defined as $C[i][j] = \min_{k=1}^{n}\{k \mid A[i][k] \cdot B[k][j] = 1\}$, for every $i, j \in [n]$. This product has applications to several graph algorithms. It has been used to compute all pairs least common ancestors in DAGs [59, 29], to find minimum weight triangles in node-weighted graphs [96], and to compute all pairs bottleneck paths in a node weighted graph [83]. The best known algorithm for the minimum witness product is by Czumaj, Kowaluk and Lingas [59, 29] and runs in $O(n^{2.575})$ time. However, when one of the matrices has at most $m = o(n^{1.575})$ nonzeros, nothing better than $O(mn)$ was known (in terms of m) for combinatorially computing the minimum witness product. As an immediate corollary of Theorem 7.5.1, we obtain the first asymptotic improvement for sparse matrices: an algorithm with $O(mn \log(n^2/m)/\log^2 n)$ running time.

Corollary 7.5.2 *Given $n \times n$ Boolean matrices A and B, where B has at most m nonzero entries, all pairs minimum witnesses of $A \cdot B$ can be computed in $O(n^2 + mn \log(n^2/m)/\log^2 n)$ time.*

Minimum Weighted Triangles. The problem of computing for all pairs of nodes in a node-weighted graph a triangle (if any) of minimum weight passing through both nodes can be reduced to finding minimum witnesses as follows ([96]). Sort the nodes in order of their weight in $O(n \log n)$ time. Create the adjacency matrix A of the graph so that the rows and columns are in the sorted order of the vertices. Compute the minimum witness product C of A. Then, for every edge $(i, j) \in G$, the minimum weight triangle passing through i and j is $(i, j, C[i][j])$. From this reduction and Corollary 7.5.2 we obtain the following.

Corollary 7.5.3 *Given a graph G with m edges and n nodes with arbitrary node weights, there is an algorithm which finds for all pairs of vertices i, j, a triangle of minimum weight sum going through i, j in $O(n^2 + mn\log(\frac{n^2}{m})/\log^2 n)$ time.*

7.6 Transitive Closure

The transitive closure matrix of an n node graph G is the $n \times n$ Boolean matrix A for which $A[i][j] = 1$ if and only if node i is reachable from node j in G. In terms of n, the complexity of computing the transitive closure of an n node graph is equivalent to that of multiplying two Boolean $n \times n$ matrices [1], thus the best known algorithm in terms of n is $O(n^\omega)$. However, when the sparsity of G is taken into account, it is unclear how to use an algorithm for sparse matrix multiplication to solve transitive closure in the same runtime. While Feder and Motwani's [38] graph compression implies an $O(mn\log(\frac{n^2}{m})/\log^2 n)$ algorithm for sparse matrix product, this result gives little insight on how to compute the transitive closure of a sparse graph—since the number of edges in the transitive closure is independent of m in general, maintaining a graph compression will not suffice. In contrast, the data structure of Theorem 7.5.1 is perfectly suited for use with a dynamic programming algorithm for transitive closure.

Theorem 7.6.1 *Transitive closure can be computed in $O(n^2 + mn\log(\frac{n^2}{m})/\log^2 n)$ time on graphs with n nodes and m edges.*

Proof. We first compute the strongly connected components of G in $O(m + n)$ time. We then contract them in linear time to obtain a DAG G'. Clearly, if we have the transitive closure matrix of G', we can recover the transitive closure matrix of G with an extra $O(n^2)$ additive overhead: for every edge (u, v) in the transitive closure graph of G', go through all pairs of vertices (i, j) such that i is in u and j is in v and add 1 to the transitive closure matrix of G. Hence it suffices for us to compute the transitive closure of a DAG G'.

First, we topologically sort G' in $O(m + n)$ time. Our transitive closure algorithm is based on a dynamic programming formulation of the problem given by Cheriyan and Mehlhorn [21], later also mentioned by Chan [16]. The algorithm proceeds in n iterations; after the kth iteration, we have computed the transitive closure of the last k nodes in the topological order. At every point, the current transitive closure is maintained in a Boolean matrix R, such that $R[u][v] = 1$ iff u is reachable from v. Let $R[\cdot][v]$ denote column v of R. Let p be the $(k + 1)$st node in reverse topological order. We wish to compute $R[\cdot][p]$, given all the vectors $R[\cdot][u]$ for nodes u after p in the topological order. To do this, we compute the componentwise OR of all vectors $R[\cdot][u]$ for the neighbors u of p.

Suppose R is a matrix containing columns $R[\cdot][u]$ for all u processed so far, and zeros otherwise. Let v_p be the *outgoing neighborhood* vector of p: $v_p[u] = 1$ iff there is an arc from p to u. Construing v_p as a column vector, we want to compute $R \cdot v_p$. Since all outgoing neighbors of p are after p in the topological order, and we process nodes in reverse order, correctness follows.

We now show how to implement the algorithm using the data structure of Theorem 7.5.1. Let R be an $n \times n$ matrix such that at the end of the algorithm R is the transitive closure of G'. We begin with R set to the all zero matrix. Let $\ell \geq 1$, and $\kappa < \ell$ be parameters to be set later. After $O((n^2\kappa/\ell)\sum_{b=1}^{\kappa}\binom{\ell}{b})$ preprocessing time, the data structure for R from Theorem 7.5.1 is complete.

Consider a fixed iteration k. Let p be the kth node in reverse topological order. As before, v_p is the neighborhood column vector of p, so that v_p has $outdeg(p)$ nonzero entries. Let $0 < \varepsilon < 1$ be

a constant. We use the data structure to compute $r_p = R \cdot v_p$ in $O(n^{1+\varepsilon} + n^2/(\ell \log n) + outdeg(p) \cdot n/(\kappa \log n))$ time. Then we set $r_p[p] = 1$, and $R[\cdot][p] := r_p$, by performing a column update on R in $O(n\kappa \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time. This completes iteration k. Since there are n iterations and since $\sum_p outdeg(p) = m$, the overall running time is asymptotic to

$$\frac{n^2\kappa}{\ell} \sum_{b=1}^{\kappa} \binom{\ell}{b} + n^{2+\varepsilon} + \frac{n^3}{\ell \log n} + \frac{mn}{\kappa \log n} + n^2\kappa \sum_{b=1}^{\kappa} \binom{\ell}{b}.$$

To minimize the running time we need to set ℓ and κ. Similar to Corollary 7.5.1, we set $\frac{n^3}{\ell \log n} = \frac{mn}{\kappa \log n}$, (implying $\ell = \kappa n^2/m$), and $n^\varepsilon = \sum_{b=1}^{\ell m/n^2} \binom{\ell}{b}$. For $m \leq n^2/2$, $\sum_{b=1}^{m\ell/n^2} \binom{\ell}{b} = O(2^{\ell H(m/n^2)}) = O(2^{\ell \frac{m}{n^2} \log \frac{n^2}{m}})$. Hence we want $\ell \frac{m}{n^2} \log \frac{n^2}{m} = \log n^\varepsilon$, and $\ell = \varepsilon' \frac{n^2 \log n}{m \log(n^2/m)}$, $\kappa = \varepsilon' \frac{\log n}{\log(n^2/m)}$ for $\varepsilon' < \varepsilon$ suffices for a runtime of $O(n^{2+\varepsilon} + mn \log(n^2/m)/\log^2 n)$. If $m = \Omega(n^{1+\varepsilon} \log^2 n)$, then the second term is $\Omega(n^{2+\varepsilon})$. On the other hand, if $m = O(n^{1+\varepsilon} \log^2 n)$, we can ignore the preprocessing step and execute the original algorithm, in $O(mn/\log n) \leq O(mn \log \frac{n^2}{m}/\log^2 n)$ time. Therefore the $n^{2+\varepsilon}$ overhead in running time can be ignored. □

7.7 APSP on Unweighted Undirected Graphs

Our data structure can also be applied to solve all pairs shortest paths (APSP) on unweighted undirected graphs, yielding the fastest algorithm for sparse graphs to date. Chan [17] gave two algorithms for this problem, which constituted the first major improvement over the $O(mn\frac{\log(n^2/m)}{\log n} + n^2)$ obtained by Feder and Motwani's graph compression [38]. The first algorithm is deterministic running in $O(n^2 \log n + mn/\log n)$ time, and the second one is Las Vegas running in $O(n^2 \log^2 \log n/\log n + mn/\log n)$ time on the RAM. To prove the following Theorem 7.7.1, we implement Chan's first algorithm, along with some modifications.

Theorem 7.7.1 *The All Pairs Shortest Paths problem in unweighted undirected graphs can be solved in $O(n^{2+\varepsilon} + mn \log(n^2/m)/\log^2 n)$ time, for any $\varepsilon > 0$.*

By running our algorithm when $m = \Omega(n^{1+\varepsilon} \log^2 n)$ for some $\varepsilon > 0$ and Chan's second algorithm otherwise, we obtain the following result.

Theorem 7.7.2 *On the probabilistic RAM, the APSP problem on unweighted undirected graphs can be solved in $O(n^2 \log^2 \log n/\log n + mn \log(n^2/m)/\log^2 n)$ time, with high probability.*

To be able to prove Theorem 7.7.1, let us focus on a particular part of Chan's first algorithm that produces the bottleneck in its runtime. The algorithm contains a subprocedure $P(d)$, parameterized by an integer $d < n$. The input to $P(d)$ is graph $G = (V, E)$, a collection of n/d vertices $\{s_1, \ldots, s_{n/d}\}$, and a collection of n/d *disjoint* vertex subsets $\{S^1, \ldots, S^{n/d}\}$, such that $\forall\, i \in [n/d]$, every $s \in S^i$ has distance at most 2 from s_i. $P(d)$ outputs the $|\cup_i S^i| \times n$ matrix D, such that for every $s \in \cup_i S^i$ and $v \in V$, $D[s][v] = \delta_G(s, v)$. Notice, $|\cup_i S^i| \leq n$. This procedure $P(d)$ is significant for the following reason:

Lemma 7.7.1 (Implicit in Chan [17]) *If $P(d)$ can be implemented in $O(T(P(d)))$ time, then APSP on unweighted undirected graphs is in $O(T(P(d)) + n^2 \cdot d)$ time.*

Setting $d = n^\varepsilon$, Theorem 7.7.1 is obtained from the following lemma.

Lemma 7.7.2 *$P(d)$ is implementable in $O(n^{2+\varepsilon} + \frac{mn}{d} + \frac{mn\log(n^2/m)}{\log^2 n})$ time, $\forall \varepsilon > 0$.*

Proof. First, we modify Chan's original algorithm slightly. As in Chan's algorithm, we first do breadth-first search from every s_i in $O(mn/d)$ time overall. When doing this, for each distance $\ell = 0, \ldots, n-1$, we compute the sets $A^i_\ell = \{v \in V \mid \delta_G(s_i, v) = \ell\}$. Suppose that the rows (columns) of a matrix M are in one-to-one correspondence with the elements of some set U. Then, for every $u \in U$, let $\bar{u}$ be the index of the row (column) of M corresponding to u.

Consider each (s_i, S^i) pair separately. Let $k = |S^i|$. For $\ell = 0, \ldots, n-1$, let $B_\ell = \cup_{j=\ell-2}^{\ell+2} A^i_\ell$, where $A^i_j = \{\}$ when $j < 0$, or $j > n-1$.

The algorithm proceeds in n iterations. Each iteration $\ell = 0, \ldots, n-1$ produces a $k \times |B_\ell|$ matrix D_ℓ, and a $k \times n$ matrix OLD (both Boolean), such that for all $s \in S^i$, $u \in B_\ell$ and $v \in V$,

$$D_\ell[\bar{s}][\bar{u}] = 1 \text{ iff } \delta_G(s,u) = \ell \text{ and } \mathrm{OLD}[\bar{s}][\bar{v}] = 1 \text{ iff } \delta_G(s,v) \leq \ell.$$

At the end of the last iteration, the matrices D_ℓ are combined in a $k \times n$ matrix D^i such that $D[\bar{s}][\bar{v}] = \ell$ iff $\delta_G(s,v) = \ell$, for every $s \in S^i$, $v \in V$. At the end of the algorithm, the matrices D^i are concatenated to create the output D.

In iteration 0, create a $k \times |B_0|$ matrix D_0, so that for every $s \in S^i$, $D_0[\bar{s}][\bar{s}] = 1$, and all 0 otherwise. Let OLD be the $k \times n$ matrix with $\mathrm{OLD}[\bar{s}][\bar{v}] = 1$ iff $v = s$.

Consider a fixed iteration ℓ. We use the output ($D_{\ell-1}$, OLD) of iteration $\ell - 1$. Create a $|B_{\ell-1}| \times |B_\ell|$ matrix N_ℓ, such that $\forall u \in B_{\ell-1}, v \in B_\ell$, $N_\ell[\bar{u}][\bar{v}] = 1$ iff v is a neighbor of u. Let N_ℓ have m_ℓ nonzeros. If there are b_ℓ edges going out of B_ℓ, one can create N_ℓ in $O(b_\ell)$ time: Reuse an $n \times n$ zero matrix N, so that N_ℓ will begin at the top left corner. For each $v \in B_\ell$ and edge (v,u), if $u \in B_{\ell-1}$, add 1 to $N[\bar{u}][\bar{v}]$. When iteration ℓ ends, zero out each $N[\bar{u}][\bar{v}]$.

Multiply $D_{\ell-1}$ by N_ℓ in $O(k \cdot |B_{\ell-1}|^{1+\varepsilon} + m_\ell k \log(\frac{n^2}{m})/\log^2 n)$ time (applying Corollary 7.5.1). This gives a $k \times |B_\ell|$ matrix A such that for all $s \in S^i$ and $v \in B_\ell$, $A[\bar{s}][\bar{v}] = 1$ iff there is a length-ℓ path between s and v. Compute D_ℓ by replacing $A[\bar{s}][\bar{v}]$ by 0 iff $\mathrm{OLD}[\bar{s}][\bar{v}] = 1$, for each $s \in S^i, v \in B_\ell$. If $D_\ell[\bar{s}][\bar{v}] = 1$, set $\mathrm{OLD}[\bar{s}][\bar{v}] = 1$.

Computing the distances from all $s \in S^i$ to all $v \in V$ can be done in

$$O(m + \sum_{\ell=1}^{n-1} \left(k \cdot |B_{\ell-1}|^{1+\varepsilon} + m_\ell k \log(n^2/m)/\log^2 n + b_\ell\right))$$

time. However, since every node appears in at most $O(1)$ sets B_ℓ, $\sum_{\ell=0}^{n-1} |B_\ell| \leq O(n)$, $\sum_\ell b_\ell \leq O(m)$ and $\sum_\ell m_\ell \leq O(m)$. The runtime becomes $O(kn^{1+\varepsilon} + mk\log(n^2/m)/\log^2 n + m)$. When we sum up the runtimes for each (s_i, S^i) pair, since the sets S^i are disjoint, we obtain asymptotically

$$\sum_{i=1}^{n/d} \left(m + |S^i| \cdot \left(n^{1+\varepsilon} + \frac{m\log(n^2/m)}{\log^2 n}\right)\right) = \frac{mn}{d} + n^{2+\varepsilon} + \frac{mn\log(n^2/m)}{\log^2 n}.$$

As the data structure returns witnesses, we can also return predecessors. □

Chapter 8

Open Problems

This thesis introduced novel techniques for designing algorithms for path problems in weighted graphs. The most pressing open question of this line of research is obtaining a truly subcubic algorithm for the distance product of two real matrices. Even though we know how to obtain such an algorithm for the case in which the maximum number of bits in any entry of the distance product is $< C \log n$ for a small constant C, the solution to the general problem has evaded us. Another, seemingly related open question is the maximum *edge-weighted* triangle problem. Our best approach for this problem has been to find all-pairs maximum edge-weighted triangles and then to take the maximum out of these. The all-pairs problem however is computationally equivalent to computing the distance product. Finding just one maximum weighted triangle might be an easier problem. Yet we do not even know an algorithm for finding a triangle in an unweighted graph faster than by using the matrix multiplication method. A slightly more modest open problem is to improve on the known results on dominance product. This would imply improved algorithms for APBP and APNP.

Many of our results on all-pairs path problems rely on good algorithms for Boolean matrix multiplication. Any improvements on the known bounds (algebraic or combinatorial) would imply improved algorithms for many problems in this thesis. The major open problem on matrix multiplication is to determine the value of ω. If $\omega = 2$ most of our algorithms would run in $\tilde{O}(n^{2.5})$. In the combinatorial algorithms setting, we would like to be able to find an $O(n^3/\log^{2+\varepsilon} n)$ algorithm for Boolean matrix multiplication for some $\varepsilon > 0$. This seems to be a modest goal, yet it has been open for many years.

There are many open problems related to our work on single source path algorithms. One of the most major open problems is determining the existence of a *linear* time algorithm for SSSP. We also do not know whether there is a linear time algorithm for single source bottleneck paths in directed graphs, or a deterministic linear time algorithm in undirected graphs. Another interesting question is whether undirected single source bottleneck paths requires the same time as the minimum spanning tree problem.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. The design and analysis of computer algorithms. *Addison-Wesley Longman Publishing Co., Boston, MA*, 1974.

[2] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice–Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.

[4] N. Alon, R. Yuster, and U. Zwick. Color-coding. *JACM*, 42(4):844–856, 1995.

[5] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[6] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[7] J. Basch, S. Khanna, and R. Motwani. On diameter verification and boolean matrix multiplication. *Report No. STAN-CS-95-1544, Department of Computer Science, Stanford University (1995)*, 1995.

[8] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[9] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Inf. Process. Lett.*, 8(5):234–235, 1979.

[10] G. Blelloch, V. Vassilevska, and R. Williams. A new combinatorial approach to sparse graph problems. In *Proc. ICALP*, 2008.

[11] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Linear time bounds for median computations. In *Proc. STOC*, volume 4, pages 119–124, 1972.

[12] G. Brodal and R. Jacob. Time-dependent networks as models to achieve fast exact time-table queries. In *Proc. ATMOS Workshop, Electronic Notes in Theoretical Computer Science*, volume 92, 2004.

[13] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*. Springer-Verlag New York, Inc., 1997.

[14] S. Butenko. Maximum independent set and related problems, with applications. *U. of Florida Doctoral Dissertation*, 2003.

[15] P. Camerini. The min-max spanning tree problem and some extensions. *Inform. Process. Lett.*, 7:10–14, 1978.

[16] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. In *Proc. WADS*, volume 3608, pages 318–324, 2005.

[17] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. In *Proc. SODA*, pages 514–523, 2006.

[18] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. STOC*, pages 590–598, 2007.

[19] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[20] J. Chen, X. Huang, I. A. Kanj, and G. Xia. Linear FPT reductions and computational lower bounds. In *Proc. STOC*, pages 212–221, 2004.

[21] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[22] N. Chiba and L. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14:210–223, 1985.

[23] M. Chudnovsky, N. Robertson, P. Seymour, and R.Thomas. The strong perfect graph theorem. *Ann. Math.*, 164:51–229, 2006.

[24] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Proc. FOCS*, volume 46, pages 379–388, 2005.

[25] H. Cohn and C. Umans. A group-theoretic approach to fast matrix multiplication. In *Proc. FOCS*, volume 44, pages 438–449, 2003.

[26] D. Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13:42–49, 1997.

[27] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM J. Comput.*, 11(3):472–492, 1982.

[28] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251–280, 1990.

[29] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theor. Comput. Sci.*, 380(1–2):37–46, 2007.

[30] A. Czumaj and A. Lingas. Finding a heaviest triangle is not harder than matrix multiplication. In *Proc. SODA*, pages 986–994, 2007.

[31] R. Dial. Algorithm 360: shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[32] E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, pages 269–271, 1959.

[33] R. Downey and M. Fellows. Fixed-parameter tractability and completeness II. On completeness for $W[1]$. *Theoretical Computer Science*, 141(1-2):109–131, 1995.

[34] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness i: Basic results. *SIAM J. Comput.*, 24(4):873–921, 1995.

[35] D. Dubois and H. Prade. Fuzzy sets and systems: Theory and applications. *Academic Press*, 1980.

[36] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[37] F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326(1-3):57–67, 2004.

[38] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Proc. STOC*, pages 123–133, 1991.

[39] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *Proc. FOCS*, pages 129–131, 1971.

[40] R. W. Floyd. Algorithm 97: shortest path. *Comm. ACM*, 5:345, 1962.

[41] L. R. Ford. Network flow theory. *Technical Report Paper P-923, The Rand Corporation*, 1956.

[42] M. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5:49–60, 1976.

[43] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.

[44] M. L. Fredman and D. E. Willard. Transdichotomous algorithms for minimum spanning trees and shortest paths. *JCSS*, 48:533–551, 1994.

[45] M. E. Furman. Applications of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Dokl. Akad. Nauk SSSR (in Russian)*, 194:524, 1970.

[46] M. E. Furman. Applications of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Math. Dokl. (in English)*, 11(5):1252, 1970.

[47] H. Gabow and R. Tarjan. Algorithms for two bottleneck optimization problems. *Journal of Algorithms*, 9(3):411–417, 1988.

[48] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *Amer. Math. Monthly*, 69:9–15, 1962.

[49] Z. Galil and O. Margalit. Witnesses for boolean matrix multiplication and for transitive closure. *Journal of Complexity*, 9(2):201–221, 1993.

[50] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *JCSS*, 54:243–254, 1997.

[51] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105–142, 1998.

[52] T. C. Hu. The maximum capacity route problem. *Operations Research*, 9(6):898–900, 1961.

[53] X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *J. of Complexity*, 14(2):257–299, 1998.

[54] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7(4):413–423, 1978.

[55] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[56] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[57] L. R. Kerr. The effect of algebraic structure on the computational complexity of matrix multiplications. *Ph.D. Thesis, Cornell University, Ithaca, NY*, 1970.

[58] T. Kloks, D. Kratsch, and H. Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3-4):115–121, 2000.

[59] M. Kowaluk and A. Lingas. LCA queries in directed acyclic graphs. In *Proc. ICALP*, volume 3580, pages 241–248, 2005.

[60] T. Lengauer and D. Theune. Unstructured path problems and the making of semirings. In *Proc. WADS*, pages 189–200, 1991.

[61] B. Mahr. A birds eye view to path problems. In *WG 1980*, pages 335–353, 1980.

[62] J. Matousek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

[63] G. J. Minty. A variant on the shortest-route problem. *Operations Research*, 6(6):882–883, 1958.

[64] Miscellaneous Authors. Queries and problems. *SIGACT News*, 16(3):38–47, 1984.

[65] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[66] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching Part II*, pages 285–292, 1959.

[67] J. I. Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971.

[68] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Comment. Math. Univ. Carolin.*, 26(2):415–419, 1985.

[69] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

[70] V. Pan. How can we speed up matrix multiplication? *SIAM Rev.*, 26(3):393–415, 1984.

[71] V. Y. Pan. Strassen's algorithm is not optimal. In *Proc. FOCS*, volume 19, pages 166–176, 1978.

[72] V. Y. Pan. New combinations of methods for the acceleration of matrix multiplications. *Comput. Math. Appl.*, 7:73–125, 1981.

[73] C. Papadimitriou and M. Yannakakis. The clique problem for planar graphs. *Information Processing Letters*, 13:131–133, 1981.

[74] M. Pollack. The maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.

[75] A. Punnen. A fast algorithm for a class of bottleneck problems. *Computing*, 56(4):397–401, 1996.

[76] A. P. Punnen. A linear time algorithm for the maximum capacity path problem. *European Journal of Operational Research*, 53:402–404, 1991.

[77] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *J. Exp. Algorithmics*, 12, 2007.

[78] F. Romani. Some properties of disjoint sums of tensors related to matrix multiplication. *SIAM J. Comput.*, 11(2):263–267, 1982.

[79] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1–3):12–22, 1985.

[80] A. Schönhage. Partial and total matrix multiplication. *SIAM J. Comput.*, 10(3):434–455, 1981.

[81] F. Schulz, D. Wagner, and C. D. Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Revised Papers From the 4th international Workshop on Algorithm Engineering and Experiments, LNCS*, volume 2409, pages 43–59, 2002.

[82] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *JCSS*, 51:400–403, 1995.

[83] A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *Proc. SODA*, pages 978–985, 2007.

[84] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. FOCS*, pages 605–614, 1999.

[85] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[86] V. Strassen. Relative bilinear complexity and matrix multiplication. *J. Reine Angew. Math.*, 375/376:406–443, 1987.

[87] C. R. Subramanian. A generalization of janson inequalities and its application to finding shortest paths. In *Proc. SODA*, pages 795–804, 1999.

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

[89] R. E. Tarjan. Private communication.

[90] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *JACM*, 46(3):362–394, 1999.

[91] A. Tucker. Coloring perfect $(K_4 - e)$-free graphs. *J. Comb. Theory Ser. B*, 42(3):313–318, 1987.

[92] P. M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *Proc. FOCS*, volume 30, pages 332–337, 1989.

[93] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315, 1975.

[94] V. Vassilevska. Nondecreasing paths in weighted graphs, or: how to optimally read a train schedule. In *Proc. SODA*, pages 465–472, 2008.

[95] V. Vassilevska and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *Proc. STOC*, pages 225–231, 2006.

[96] V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H-subgraph in real weighted graphs and related problems. In *Proc. ICALP*, volume 4051, pages 262–273, 2006.

[97] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proc. STOC*, pages 585–589, 2007.

[98] V. Vassilevska, R. Williams, and R. Yuster. Finding heaviest H-subgraphs in real weighted graphs, with applications. *ACM Trans. on Algorithms*, 4(3), 2008.

[99] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[100] R. Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *Proc. SODA*, pages 995–1001, 2007.

[101] G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. In *Proc. IWPEC, LNCS 3162*, pages 281–290, 2004.

[102] R. Yuster and U. Zwick. Finding even cycles even faster. In *Proc. ICALP*, pages 532–543, 1994.

[103] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proc. SODA*, pages 247–253, 2004.

[104] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. on Algorithms*, 1(1):2–13, 2005.

[105] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. *Inf. Proc. Letters*, 4:155–156, 1976.

[106] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *JACM*, 49(3):289–317, 2002.