UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

DOCTORAL THESIS
(DOKTORSKA DISERTACIJA)

OPTIMIZATION AND GRAPHS: EFFICIENCY OF SOME ALGORITHMS IN THEORY AND PRACTICE

(OPTIMIZACIJA IN GRAFI: UČINKOVITOST NEKATERIH ALGORITMOV V TEORIJI IN PRAKSI)

MARKO GRGUROVIČ

KOPER, 2025

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

DOCTORAL THESIS
(DOKTORSKA DISERTACIJA)

OPTIMIZATION AND GRAPHS: EFFICIENCY OF SOME ALGORITHMS IN THEORY AND PRACTICE

(OPTIMIZACIJA IN GRAFI: UČINKOVITOST NEKATERIH ALGORITMOV V TEORIJI IN PRAKSI)

MARKO GRGUROVIČ

KOPER, 2025

MENTOR: PROF. ANDREJ BRODNIK, PHD
CO-MENTOR: ASSIST. PROF. ROK POŽAR, PHD

Acknowledgement

I'd like to thank my advisor Prof. Andrej Brodnik for the countless hours of discussion and advice during my graduate (and undergraduate) studies. Likewise I'd like to express my gratitude to my co-advisor Prof. Rok Požar, who has contributed a lot to the discussions and to the papers published during the formation of this thesis. I would also like to express gratitude to members of the thesis committee: Gerth Stølting Brodal, Sergio Cabello, and Martin Milanič who have meticulously combed through this thesis, pointing out issues and suggesting improvements. Thanks to the former and current guys at DIST – Tine, Jernej, Aleksandar, and others for all the discussions and beers (especially the beers). I'd like to thank my mother Janja for the support during the initial years of my studies, and later for the constant reminders that I still have a thesis to write (to be fair, it took me a while!). Last but not least, I'd like to thank my girlfriend Simona, who has been very supportive of this whole ordeal and who has helped push me to the finish line.

Abstract

OPTIMIZATION AND GRAPHS: EFFICIENCY OF SOME ALGORITHMS IN THEORY AND PRACTICE

This dissertation focuses on algorithms and data structures used to solve combinatorial optimization problems. The results presented are considered both from the point of view of theory, as well as that of practical significance.

The first combinatorial optimization problem we consider involves finding shortest paths between all vertex pairs in a graph. We give an algorithm that efficiently solves the all-pairs shortest path problem on non-negatively weighted graphs using a single-source shortest path algorithm ψ as a black box. Its running time is $O(m \lg n + nT_\psi(m^* + n, n + 1))$ where $T_\psi(m, n)$ is the time required by algorithm ψ on a graph with m arcs and n vertices, and m^* is the number of arcs belonging to shortest paths in the graph. Furthermore, we provide another algorithm based on the well known Floyd-Warshall algorithm, which solves the same problem and runs in $O(n^2 \log^2 n)$ expected time for the class of complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0, 1]$. In both cases, the experimental evaluation confirms the theoretical results.

The original Floyd-Warshall solution to the shortest path problem is formulated as a dynamic programming algorithm. We briefly study the minimum equation which comes up frequently in the dynamic programming formulation of problems in bioinformatics, geology, and speech recognition. We obtain an algorithm that extends the previous special-case results to a wider family of inputs, resulting in a general-case solution with running time that is adaptive to the input.

The last group of combinatorial optimization problems we study includes problems that are intractable. To get a good enough solution in this case, we study a parallelized ant system algorithm solving the traveling salesman problem on n cities. Following prior results for the graphics processing unit model, we show that they translate to the parallel random access machine model and introduce further improvements. These lead us to new asymptotic bounds for the parallel ant system with step complexities $O(n \lg \lg n)$ on CRCW PRAM and $O(n \lg n)$ on CREW PRAM, using n^2 processors in both cases. As before, we conclude with an experimental evaluation of our solution.

Math. Subj. Class (2020): 05C85, 68Q25, 68W10, 90C27, 90C39

Key words: dynamic programming, combinatorial optimization, ant colony optimization, shortest path problem, bottleneck path problem, traveling salesman problem, parallel algorithms, asymptotic analysis, expected-case analysis.

Izvleček

OPTIMIZACIJA IN GRAFI: UČINKOVITOST NEKATERIH ALGORITMOV V TEORIJI IN PRAKSI

Disertacija se osredotoča na področje algoritmov in podatkovnih struktur, ki jih uporabljamo pri reševanju problemov kombinatorične optimizacije. Predstavljeni rezultati so analizirani tako iz zornega kota teoretične analize kot tudi iz zornega kota praktičnega ovrednotenja.

Prvi problem kombinatorične optimizacije, s katerim se soočimo, je iskanje najkrajših poti med vsemi pari vozlišč v grafu. Predstavimo algoritem, ki učinkovito poišče najkrajše poti med vsemi pari vozlišč na nenegativno uteženih grafih z uporabo algoritma ψ kot črne škatle. Pri tem je ψ algoritem, ki najde najkrajše poti iz enega izvora do vseh ostalih vozlišč. Asimptotična časovna zahtevnost algoritma je $O(m \lg n + nT_\psi(m^* + n, n + 1))$, kjer je $T_\psi(m, n)$ časovna zahtevnost algoritma ψ na grafu z m povezavami in n vozlišči, m^* pa predstavlja število povezav, ki so vsebovane v najkrajših poteh v grafu. Poleg tega predstavimo še algoritem, zasnovan na osnovi Floyd-Warshallovega algoritma. Predstavljeni algoritem rešuje isti problem in ima pričakovano časovno zahtevnost $O(n^2 \log^2 n)$ za razred polnih usmerjenih grafov na n vozliščih z naključno in neodvisno izbranimi utežmi povezav, porazdeljenimi enakomerno na intervalu $[0, 1]$. Teoretične rezultate potrdimo s praktičnim ovrednotenjem pri obeh predstavljenih algoritmih.

Floyd-Warshallov algoritem uporablja metodo dinamičnega programiranja, zato se posvetimo preučevanju minimizacijske kriterijske funkcije, ki je sestavni del tehnike. Pogosto se pojavlja pri rešitvah problemov, ki jih srečamo v bioinformatiki, geologiji in razpoznavanju govora. Pri tem pristop, ki je definiran za posebno obliko vhodnih podatkov, razširimo na večjo družino le-teh, kar nam da splošno rešitev s časovno zahtevnostjo, odvisno od oblike vhodnih podatkov.

Zadnji razred problemov, s katerimi se spopademo, vključuje probleme, ki v praksi niso rešljivi. Tokrat iščemo le dovolj dobro rešitev, in v ta namen preučimo vzporedno izvedbo sistema mravelj, ki rešuje problem trgovskega potnika na n mestih. Pri tem obstoječe rešitve za grafično procesorsko enoto (GPE) najprej prevedemo na model stroja z naključnim dostopom PRAM, kjer jih izboljšamo in nadgrajene uporabimo neposredno v kodiranju rešitve za GPE. Tako dobljene rešitve imajo koračno zahtevnost $O(n \lg \lg n)$ na CRCW PRAM in $O(n \lg n)$ na CREW PRAM, z uporabo n^2 procesorjev v obeh modelih računanja. Ponovno zaključimo s praktičnim ovrednotenjem naše rešitve.

Math. Subj. Class (2020): 05C85, 68Q25, 68W10, 90C27, 90C39

Ključne besede: dinamično programiranje, kombinatorična optimizacija, optimizacija s kolonijo mravelj, problem najkrajših poti, problem najširših poti, problem trgovskega potnika, vzporedni algoritmi, asimptotična analiza, analiza pričakovanega časa.

Contents

List of Figures

Chapter 1

Introduction

This dissertation focuses on algorithms and data structures used to solve combinatorial optimization problems. The results presented are studied both from the point of view of theory, as well as that of practical significance. Combinatorial optimization, an area of applied mathematics and theoretical computer science, deals with finding the best solutions to certain discrete problems. In contrast to continuous optimization problems, where the parameters that define the solution are continuous in nature, the problems of combinatorial optimization have parameters that are discrete. The various methods and approaches to solving combinatorial optimization problems include, among others, the greedy method and dynamic programming, both of which are featured prominently in this work. Problem solutions are often in the form of a subset, graph, integer, or other similar discrete structures. The practical utility of combinatorial optimization is evident in the wide array of real-world problems that it tackles: optimizing schedules, finding the most economical paths, computing minimal spanning trees, knapsack problems, etc.

Our focus will be primarily on graph-related combinatorial optimization problems. Indeed, graphs are one of the most common structures used to model the complex nature of the world. Therefore it is not surprising that a wide range of problems have been defined on them. Although these problems have been around since even before the inception of the computer science field, obtaining faster algorithms or nontrivial lower bounds has proven challenging for many of them. This thesis is primarily concerned with classical problems, but likewise concedes to some degree that performance might not necessarily be best described in the worst-case scenario.

Path optimization problems. The graphs considered in this thesis are directed. The length (also weight) of a path in a weighted graph is the sum of all weights of the arcs along the path. A shortest path from a vertex u to a vertex v is any path between u and v having minimal length. The length of a shortest path from u to v is called the *distance* from u to v. The problem of finding shortest paths in graphs typically has two variants. In the *single source shortest path* variant, we are looking for the shortest paths from a given source vertex to all other vertices in the graph. In the *all-pairs shortest path* variant, we are looking for the shortest paths between all vertex pairs. Finding shortest paths in graphs is one of the most iconic problems and the study of all-pairs shortest path algorithms comprises the bulk of the thesis, and concerns itself with two entirely different approaches: the Dijkstra approach [22] and the Floyd-Warshall approach [26, 83]. Both of these methods date into the 1960's, with

Dijkstra's approach seeing some improvements over the years such as the introduction of Fibonacci heaps [30] and derivations such as the Hidden Paths [48] and Uniform Paths [21] algorithms. On the other hand, there has been virtually no progress on the Floyd-Warshall algorithm.

In the case of Dijkstra's approach, we show that the all-pairs shortest path problem can be solved by an algorithm that uses a black-box single-source shortest path solver. This allows us to effectively attain average-case speedups on any single-source algorithm when used in this setting, and even lays bare some interesting connections between sorting and Dijkstra's approach to all-pairs shortest path. Given a graph with n vertices and m arcs, it is trivially known due to its reliance on priority queues, that Dijkstra's approach to single-source shortest path is necessarily lower-bounded by sorting, and thus no $o(m + n \lg n)$ algorithm is possible under the comparison-addition model. However, the single-source problem might not require sorting at all, so that bound does not say much about the problem itself. Interestingly, the connection we uncover is between a sorted version of the all-pairs problem and the single-source problem itself, indicating that a sorting bound on the sorted all-pairs problem can directly translate to a lower-bound on the (unsorted) single-source problem.

On the other hand, we have the Floyd-Warshall approach [26, 83], a completely different way of tackling the all-pairs problem that runs in time $O(n^3)$. Formulated as a dynamic programming algorithm and immediately recognizable due to its simplicity, it consists of a mere three nested *for* loops and a single relaxation operation. Our improvement has been motivated by observing the relaxation operation and considering how many times this operation fails to decrease the value already stored. As it turns out, the answer is that it fails most of the time. By extending the algorithm with a few more steps instead of a straightforward *for* loop and exploiting the structure of shortest paths, we have devised some improvements that have resulted in excellent performance in practice. Motivated by empirical results and preliminary calculations, we set out to rigorously prove the expected-case time complexity. The obtained $O(n^2 \log^2 n)$ expected-case bound is still worse than the $O(n^2 \log n)$ expected case bound achieved by the Hidden Paths algorithm [48], an algorithm that would fall under Dijkstra's approach. However, it is nonetheless surprising for such a small modification of Floyd-Warshall, an algorithm that has resisted improvement for so long. Due to its simplicity, it performs remarkably well in practice.

The bottleneck paths problem [43,68] is closely related to the shortest path problem, except that instead of the distance of a path, we define the width of a path as the minimum weight arc on the path. In other words, we change the aggregation operation from sum to minimum. The bottleneck path (also called the widest path) from u to v is then any path between u and v with maximum width. The bottleneck paths problem also comes in both single-source and all-pairs variety: in the case of single-source, we are looking for bottleneck paths from a given source vertex to all other vertices in the graph; and in the case of the all-pairs variant, we are asked to find bottleneck paths for all vertex pairs. In this work we investigate how to solve the all-pairs variant of this problem efficiently, and also point out a connection to the dynamic transitive closure problem on graphs.

As mentioned above, the original Floyd-Warshall solution to the shortest path problem is formulated as a dynamic programming algorithm. Many dynamic programming formulations contain a subproblem of finding the minimal sum of two elements drawn from one or more sets of such elements. We study this subproblem on its own, which

can be found in solutions to a wide variety of problems in bioinformatics [62], geology [77] and speech recognition [71]. We study this subproblem and show how a certain special-case approach can be generalized to achieve a solution that works in all cases and improves the performance of some dynamic programming algorithms.

Intractable problems. Although the problems we have mentioned thus far can be solved in polynomial time, there are many NP-hard problems on graphs which take prohibitively long to solve optimally. One such problem is the traveling salesman problem [49], where we are given a graph with vertices representing cities and arcs representing connections between cities. In the usual formulation each city is connected to every other city and the arcs are weighted so that the weight represents the distance between the cities it connects. The traveling salesman problem asks for the shortest Hamiltonian cycle.

Metaheuristic optimization algorithms are often used when solving NP-hard problems such as the traveling salesman problem, eschewing optimal solutions in favor of solutions that are good enough. Another unrelated approach is to run parts of the algorithm in parallel on several processors simultaneously. Combining these approaches together brings us to the idea of parallel metaheuristics. It is interesting that many metaheuristic algorithms are inspired by nature in one way or another, leading to algorithms such as genetic algorithms, differential evolution, simulated annealing, harmony search, particle swarm optimization, ant colony optimization, and many more. Since this is such a wide and varied field, we focus on the Ant system algorithm and study its behavior in a parallel environment solving the traveling salesman problem. The ant system algorithm works by placing artificial ants on a graph, and then tasks the ants with the goal to find the shortest Hamiltonian cycle. The algorithm mimics the techniques used by real ants to find paths from their nest to a source of food and back. Ants progressively build a solution by moving around on the graph. Movement is done probabilistically, based on the pheromone model, which is simply a set of parameters tied to graph arcs, that the ants can change during execution. Most studies on metaheuristic optimization are primarily empirical both in terms of running time as well as solution quality. Since our interest is both theoretical and empirical, we apply theoretical analysis when it comes to running time and analyze the algorithms in the well-understood parallel random access machine model.

Structure of the thesis. The rest of this thesis is structured in the following way. Chapter 2 introduces common notation, concepts and terminology used throughout the thesis. In Chapter 3 we study a subproblem that commonly occurs in the dynamic programming formulation of a wide variety of problems in bioinformatics, geology and in speech recognition.

Chapter 4 focuses on shortest path problems on graphs. It includes improvements to both the Dijkstra and Floyd-Warshall approach to solving the classic all-pairs shortest path problem. It presents a new algorithm for the all-pairs bottleneck path problem, a study of properties of certain paths in random graphs and finally, empirical comparisons of actual implementations of both novel and existing algorithms.

In Chapter 5 we take a different direction and study the parallel Ant system algorithm. We investigate how existing implementations for the graphics processing unit might be analyzed more rigorously on the parallel random access machine with the goal of finding areas where further improvement could be made. In addition, we also

compare existing and novel implementations empirically.

We conclude the thesis with Chapter 6 which offers an overview of the results and presents some open problems.

Some of the results of this thesis are published in the following articles:

- [8] A. Brodnik and M. Grgurovič. Speeding up shortest path algorithms. In K. Chao, T. Hsu, and D. Lee, editors, *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, volume 7676 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2012. (Presented in Section 4.1.)

- [9] A. Brodnik and M. Grgurovič. Solving all-pairs shortest path by single-source computations: Theory and practice. *Discrete Applied Mathematics*, 231(Supplement C):119 – 130, 2017. Algorithmic Graph Theory on the Adriatic Coast. (Presented in Sections 4.1 and 4.4.)

- [7] A. Brodnik and M. Grgurovič. Practical algorithms for the all-pairs shortest path problem. In A. Adamatzky, editor, *Shortest Path Solvers. From Software to Wetware*, pages 163–180. Springer International Publishing, Cham, 2018. (Presented in Sections 4.3, 4.4, and 4.5.)

- [10] A. Brodnik and M. Grgurovič. Parallelization of ant system for GPU under the PRAM model. *Comput. Informatics*, 37(1):229–243, 2018. (Presented in Chapter 5.)

- [6] A. Brodnik, M. Grgurovič, and R. Požar. Modifications of the Floyd-Warshall algorithm with nearly quadratic expected-time. *Ars Math. Contemp.*, 22(1):1–22, 2022. (Presented in Sections 4.2, 4.3, and 4.4.)

Chapter 2

Basics

Throughout the thesis we will use consistent concepts, terminology and notation which we establish in this chapter. Consequently, the reader is instructed to refer back to this chapter while reading the thesis. All logarithms written as $\log x$ are base e unless explicitly stated otherwise, and all logarithms written as $\lg x$ are base 2. For convenience, we define $[n] = \{1, 2, \ldots, n\}$.

2.1 Models of computation

Algorithms are analyzed based on idealized theoretical models. Two such models are outlined in Sections 2.1.1 and 2.1.2. However, in spite of an optimistic theoretical analysis of an algorithm, it is important to keep in mind that such models, and therefore results, do not always translate well into practice. Through implementation and subsequent empirical testing, we can determine how such algorithms perform in practice and compare them with other existing solutions.

Proving the correctness of an algorithm is a key step in its analysis, as it guarantees that the obtained result is a valid solution to the problem. More in-depth formal analysis of an algorithm also includes the asymptotic time and space complexities. On the basis of asymptotic complexity, we can determine the efficiency of an algorithm, and compare two algorithms with each other on a completely theoretical level. There are several types of asymptotic analyses, but for the purposes of the thesis, we focus on analyses of worst-case and expected-case asymptotic complexity.

2.1.1 Random access machine

The RAM [16] (*random access machine*) model is a classical model of computation in theoretical computer science. It is used in the analysis of sequential algorithms, and consists of a memory that contains an unbounded number of registers, which can store integers. Access to the registers is through direct access in constant time. Often, we are faced with a more powerful model called word RAM [31] which allows the execution of certain arithmetic and logical operations on registers in constant time. However, the word RAM also puts a limit to the size of a register, usually to $O(\lg n)$ where n is the problem size, which ensures we are able to fit the numerical representation of the size of the problem into a single register.

2.1.2 Parallel random access machine

The PRAM [28] (*parallel random access machine*) model is an extension of the RAM model that permits analysis of parallel algorithms. It consists of an unbounded number of processors and memory that is shared across all processors, which is a type of memory architecture that is often called UMA [44] (*unified memory access*). Processors in a PRAM execute instructions in parallel and synchronously. During analysis of time complexity, for a problem of size n, we distinguish between step complexity $S(n)$, which represents the maximum classical time complexity over all processors, and work complexity $W(n)$, which represents the sum of classical time complexity over all processors. Under the Flynn taxonomy [27] we would place the PRAM model somewhere between the SIMD (*single instruction, multiple data*) and MIMD (*multiple instruction, multiple data*) models, since under PRAM in the case of a branch, different processors can execute different instructions.

We denote the number of processors by p. In this thesis, we deal with two types of synchronous PRAM: concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW). The CREW variant allows processors to read from any memory location at any time, but does not allow two processors to write to the same memory location at the same time. In contrast, the CRCW variant has no such write restriction. Since under CRCW all processors can write to the same location at once, it is typical to parametrize the CRCW variant by how the competing writes are handled. In this thesis we consider two standard ways of doing that:

- COMMON: All processors must write the same value.
- COMBINING: All values being concurrently written are combined using some operator (e.g., addition, maximum, etc).

We focus on CREW, CRCW, and COMBINING CRCW algorithms, where by CRCW we mean algorithms that run under the COMMON variant. An important parallel operation which we will make extensive use of is finding the largest element in an array of n elements. It is important to note that finding the maximum among n numbers can be performed in $S(n) = O(\lg\lg n)$ steps under CRCW [73] with $p = \frac{n}{\lg\lg n}$. However, it is only possible in $S(n) = O(\lg n)$ steps under CREW with $p = \frac{n}{\lg n}$. The work complexity is $W(n) = O(n)$ in both cases. Under COMBINING CRCW, finding the maximum can be performed in $S(n) = O(1)$ and $W(n) = O(n)$ by making use of the combining mechanism in a trivial way (i.e., setting it to be the maximum operation).

2.2 Graphs

A *digraph* (or directed graph) G is a pair (V, A), where V is a non-empty finite set of elements called vertices and $A \subseteq V \times V$ a set of ordered pairs called arcs. We assume $V = \{v_1, v_2, \ldots, v_n\}$ for some n, and define $m = |A|$ for convenience. The two vertices joined by an arc are the arc's endvertices. The outdegree of $v \in V$, is the number of arcs in A that have origin v. The maximum outdegree in G is denoted by $\Delta(G)$.

A digraph $G' = (V', A')$ is a *subdigraph* of the digraph $G = (V, A)$ if $V' \subseteq V$ and $A' \subseteq A$. The (vertex-)induced subdigraph with the vertex set $S \subseteq V$, denoted by $G[S]$, is the subgraph (S, C) of G, where C contains all arcs $a \in A$ that have both endvertices in S, that is, $C = A \cap (S \times S)$. The (arc-)induced subdigraph with the arc set $B \subseteq A$,

denoted by $G[B]$, is the subgraph (U, B) of G, where U is the set of all those vertices in V that are endvertices of at least one arc in B.

A path P in G from $v_{P,0}$ to $v_{P,r}$ is a finite sequence $P = v_{P,0}, v_{P,1}, \ldots, v_{P,r}$ of pairwise distinct vertices such that $(v_{P,i}, v_{P,i+1})$ is an arc of G, for $i = 0, 1, \ldots, r-1$. The length of a path P, denoted by $|P|$, is the number of vertices occurring on P. Any vertex of P other than $v_{P,0}$ or $v_{P,r}$ is an intermediate vertex. Given a graph G with vertex set $V = \{v_1, \ldots, v_n\}$ and an integer $k \in \{0, 1, \ldots, n\}$, a *k-path* in G (with respect to the vertex ordering $v_1, \ldots, v_n$) is a path in G such that all internal vertices belong to the set $\{v_1, \ldots, v_k\}$. Obviously, a 0-path has no internal vertices.

A weighted digraph is a digraph $G = (V, A)$ together with a weight function $w \colon A \to \mathbb{R}$ that assigns each arc $a \in A$ a weight $w(a)$. A weight function w can be extended to path P by $w(P) = \sum_{i=0}^{r-1} w(v_{P,i}, v_{P,i+1})$. A shortest path from u to v, denoted by $u \rightsquigarrow v$, is a path in G whose weight is minimum among all paths from u to v. The distance from vertex u to vertex v, denoted by $D_G(u, v)$, is the weight of a shortest path $u \rightsquigarrow v$ in G. Given a subset $S \subseteq V$, the distance between S and a vertex v in G, denoted by $D_G(S, v)$, is $D_G(S, v) = \min_{u \in S} D_G(u, v)$. A shortest k-path from u to v is denoted by $u \overset{k}{\rightsquigarrow} v$. Further, we denote the set of all arcs that are part of some shortest k-path in G by $A^{(k)}$ and the subdigraph $G[A^{(k)}]$ by $G^{(k)}$. Note that the digraph $G^{(n)}$ is known as the essential subdigraph [58] of G.

2.3 Combinatorial optimization

Combinatorial optimization is a branch of mathematics and computer science that focuses on finding the optimal solution from a finite set of possible solutions. It involves exploring all possible combinations of a set of objects or variables to determine the best combination that satisfies a set of constraints or minimizes a specific objective function. Combinatorial optimization is used to solve a wide range of real-world problems, including logistics, scheduling, resource allocation, network design, and many others. The key challenge in combinatorial optimization is to efficiently search through the large space of possible solutions to find the optimal one. This field has developed various techniques and algorithms, such as linear programming, dynamic programming, and branch and bound, to efficiently solve combinatorial optimization problems. For a comprehensive overview of the field we refer the reader to [72].

2.4 Metaheuristic optimization

Metaheuristic optimization algorithms are optimization algorithms or general strategies that can be used to solve optimization problems, but without guaranteeing the optimality (or even an approximation) of the obtained solutions. For simple problems, where we might find the optimal solution relatively quickly, metaheuristics are a poor choice. However, the metaheuristic algorithms prove to be useful in practice in cases when the time required to find an optimal solution is prohibitively long. Among such problems are, for example, NP-hard problems.

Many metaheuristics mimic or are inspired by the behavior of natural systems, such as genetic algorithms, differential evolution, simulated annealing, harmony search, particle swarm optimization, ant colony optimization, and many more [5].

Chapter 3

Dynamic programming

In contrast to the *divide and conquer* approach, where we split a problem into subproblems and solve them independently, dynamic programming solves problems that are broken into subproblems which overlap and are not entirely independent. Dynamic programming [18] is thus a method which splits a problem into smaller subproblems, solves them once, and to improve performance memorizes the solutions in order to reuse them whenever the same subproblems resurface. This avoids redundantly solving the same subproblem multiple times. Furthermore, the solution to the problem is constructed from solutions to subproblems, usually by computing the minimum or maximum over a set of values. Algorithms that leverage techniques of dynamic programming are used in solving optimization problems such as finding shortest paths [26, 83], finding the shortest edit distance [71], etc.

3.1 Computing the minimum

We consider the following problem: given a vector $X = [X_0, \ldots, X_{n-1}]$ of real values and a function $g(i)$ for $1 \leq i \leq n$, compute for all $1 \leq i \leq n$:

$$Y_i = \min_{0 \leq k < i} \{X_k + g(i-k)\}. \tag{3.1}$$

It can be shown that certain dynamic programming algorithms used in bioinformatics [62], geology [77], and in speech recognition [71], can be reduced to this problem. The naïve algorithm for solving Eq. (3.1) takes $O(n)$ time for each i, which amounts to a total of $O(n^2)$ time over all i.

Focusing on specific cases of the function g (usually referred to as the *gap function*) allows us to solve this problem faster. We will focus on the algorithm from [34] for the case when g is nondecreasing and *convex*, that is, a function that grows at an increasing rate; more formally, given $a < b$ the following holds:

$$g(b+c) - g(a+c) \leq g(b+c') - g(a+c') \quad \text{for} \quad 0 \leq c \leq c', \tag{3.2}$$

but the same result can also be obtained with minor modifications when g is concave, as shown in [34].

Previous algorithms for this problem focused on either the concave case [34, 41, 84] or the convex case [34]. In this section, we develop an algorithm that is a combination of the special-case algorithms described in [34] but works for all inputs, and its running time depends on the number of inflection points of the function g, which are points where convexity changes into concavity or vice-versa.

3.1.1 The convex and concave case

We first describe the algorithm from [34] for the case when g is convex. Consider the sequence of integers $S = [1, 2, \ldots, n]$. We will assign to each element i in S the value Y_i, i.e., the combination that achieves the minimum for a given i in Eq. (3.1). For each i, we would like to find a k that achieves the minimum for that i.

Lemma 3.1. *Let g be a convex function and let k achieve the minimum for i. Then, there exists an index j achieving the minimum for $i+1$ such that $j \geq k$.*

Proof. Note that it is enough to prove that for all $0 \leq j' < k$ we have

$$X_{j'} + g(i+1-j') \geq X_k + g(i+1-k).$$

Let j' be an arbitrary index such that $0 \leq j' < k$. Since k achieves the minimum for i, we have

$$X_{j'} + g(i-j') \geq X_k + g(i-k).$$

Rearranging the last inequality we get

$$X_k - X_{j'} \leq g(i-j') - g(i-k).$$

Take $a = i-k$, $b = i-j'$, $c = 0$ and $c' = 1$. Since $a < b$, $0 \leq c \leq c'$ and g is convex, we can plug a, b, c and c' into Eq. (3.2) and obtain

$$g(i-j') - g(i-k) \leq g(i+1-j') - g(i+1-k).$$

Hence,

$$X_k - X_{j'} \leq g(i-j') - g(i-k) \leq g(i+1-j') - g(i+1-k),$$

as desired. □

Lemma 3.2. *Let g be a convex function and let k achieve the minimum for i, but not for $i+1$. Then, k does not achieve the minimum for any $i' > i$.*

Proof. By Lemma 3.1, there exists an index j achieving the minimum for $i+1$ such that $j \geq k$. Since k does not achieve the minimum for $i+1$, we have $j > k$ and

$$X_k + g(i+1-k) > X_j + g(i+1-j),$$

or equivalently,

$$X_j - X_k < g(i+1-k) - g(i+1-j),$$

We now prove that, for any $d \geq 1$ we have

$$X_j - X_k < g(i+d-k) - g(i+d-j).$$

Let $d \geq 1$ be an arbitrary index. Take $a = i-j$, $b = i-k$, $c = 1$, and $c' = d$. Since $a < b$, $0 \leq c \leq c'$ and g is convex, we can plug a, b, c and c' into Eq. (3.2) and obtain

$$g(i+1-k) - g(i+1-j) \leq g(i+d-k) - g(i+d-j).$$

Hence,

$$X_j - X_k < g(i+1-k) - g(i+1-j) \leq g(i+d-k) - g(i+d-j),$$

which completes the proof. □

What follows from Lemma 3.2 is that each element k will achieve the minimum for precisely a contiguous (possibly empty) subsequence of S.

In order to find the subsequence for each k, we would like to know, given two elements j and k with $j < k$, for which $i > k$ we have

$$X_k + g(i-k) \leq X_j + g(i-j).$$

The latter translates to the following: find the maximal index c in $\{k, k+1, \ldots, n\}$ such that

$$X_k - X_j \leq g(c-j) - g(c-k)$$

holds, which can be done in $O(\lg n)$ using binary search.

Now we are ready to describe the ConvexCaseMin algorithm from [34]. We will assign to each $0 \leq k < n$ a contiguous subsequence of S for which it achieves the minimum, and these subsequences will not overlap with each other. We start by assigning 0 the subsequence consisting of the entire sequence S. Then, the algorithm works by traversing the list of candidates $1, \ldots, n-1$. Let us denote the current candidate by u. We compare u with whichever element v is assigned to the rightmost subsequence of S. One can now determine for which values of k the new candidate achieves a lower value according to Eq. (3.1), and the subsequence can be updated by being split into two if necessary. In the case that the new candidate is better than the previous one for the entire subsequence, we repeat the process on the subsequence to the left of the current subsequence. Once we are done, we merge and/or update the boundaries of any neighboring subsequences that we updated.

Lemma 3.3. *When g is a convex function, the ConvexCaseMin algorithm correctly computes the solution to Eq.* (3.1)*.*

Proof. The correctness of the algorithm follows from Lemma 3.2 and the discussion above. □

Lemma 3.4. *The ConvexCaseMin algorithm has a time complexity of $O(n \lg n)$.*

Proof. In order to analyze the time required by the algorithm, we go through each candidate $0 \leq u < n$ and count how many binary searches are performed, each taking $O(\lg n)$ time. Assume that there are currently $0 < d < n$ subsequences, and u proves to be the better candidate than the candidate assigned to $0 \leq c \leq d$ of those subsequences. This means $c+1$ binary searches will be performed. If $c = 0$, this will cost us $O(\lg n)$. Otherwise if $c > 0$, note that after we are done with u and move onto the next candidate from S, the number of subsequences will be $d' = d - c + 1$ since we will reduce the number of subsequences due to concatenating the c subsequences into a single one. Thus, the number of such additional searches is $O(n)$. Therefore, the algorithm takes $O(n \lg n)$ time. □

We do not explicitly describe the algorithm in the concave case from [34], since it is similar to the convex case.

3.1.2 General case

We are now ready to state our contribution. Observe that the function g only takes on integral inputs. Therefore, we can consider the sequence of values:

$$g(2) - g(1), g(3) - g(2), \ldots, g(n) - g(n-1).$$

In a preprocessing step, we can traverse this sequence to discover non-overlapping, contiguous subsequences (which we will call *blocks*) that contain elements in ascending (resp. descending) order. These are precisely the regions of the function which are convex (resp. concave). We build a list L which, for each block, contains an element $(start, end, asc)$ that stores information about the block boundary ($start$, end) and whether the block is in ascending ($asc = 1$) or descending ($asc = 0$) order. Let us denote the number of such blocks $b_1, b_2, \ldots, b_B$ by B, and let $|b_i|$ denote the number of elements in block b_i. Observe that $1 \leq B \leq n/2$, since two elements form either an ascending or descending block, and in the case $B = 1$ the function is either convex or concave. This step takes $O(n)$ time, and allows us to partition g into disjoint intervals where the property of sorted residues holds.

The algorithm works by going through the list L and for each block $(start, end, asc)$, it finds the minimum combinations of $Y_i = X_k + g(i - k)$ for all Y_i whenever $i - k$ is inside the interval $(start, end)$. Once these combinations have been found, they are stored as Y_i if they are lower than the current value and the algorithm moves onto the next block, where the same process is repeated. Observe that these subproblems can be solved by a slight modification of the algorithms from Subsection 3.1.1, we use the convex algorithm if $asc = 1$ and the concave algorithm otherwise. Once we solve the subproblem in a block, we will not revisit it and so the space can be reused. The algorithm requires only $O(n)$ extra space.

Lemma 3.5. *The general-case algorithm correctly computes the solution to Eq.* (3.1).

Proof. From Lemma 3.2 and Lemma 3.3 we already know that inside a sorted contiguous interval of g, we can use binary search to find the combinations that achieve the minimum Y_i. However, since g does not consist of a single sorted contiguous interval and may indeed be broken up into several such intervals, it remains to show that the algorithm finds the combinations that achieve the minimum over all such intervals. This is accomplished by the algorithm by exhaustively computing the solutions for each interval separately and maintaining the minimal values Y_i at all times. □

Lemma 3.6. *The general-case algorithm has a time complexity of* $O(Bn \lg(\frac{n}{B}))$.

Proof. The time required to solve the subproblem on block b_i comes from the algorithm described in Section 3.1.1. In our case, the binary search is performed on b_i, so the time is $O(n \lg |b_i|)$. Over all subproblems the time becomes:

$$O\left(n \sum_{i=1}^{B} \lg |b_i|\right).$$

Turning the sum of logarithms into the logarithm of the product we get:

$$O\left(n \lg \left(\prod_{i=1}^{B} |b_i|\right)\right).$$

Assume B is fixed. Since the logarithm is a monotonically increasing function, the time is maximized when $\prod_{i=1}^{B} |b_i|$ is maximized. Recall that the geometric mean is less than or equal to the arithmetic mean. Hence, we have:

$$\left(\prod_{i=1}^{B} |b_i|\right)^{1/B} \leq n/B$$

$$\prod_{i=1}^{B} |b_i| \leq (n/B)^B.$$

Thus, we can upper bound the time by $O(Bn \lg(\frac{n}{B}))$. □

Regardless of g, the time is never worse than the straightforward $O(n^2)$ algorithm, and achieves the $O(n \lg n)$ bound when $B = O(1)$. Furthermore, B can be upper bounded by the number of inflection points of g. Recall for example, that a polynomial of degree d has at most $d - 2$ inflection points. Therefore, if g is a polynomial, we can upper bound the running time by $O(dn \lg n)$.

Chapter 4

Shortest paths in graphs

Finding shortest paths in graphs is a classical problem in algorithmic graph theory. The problem pops up frequently also in practice in areas like bioinformatics, logistics, and VLSI design (for a more comprehensive list of applications see, e.g., [2]). Two of the most common variants of the problem are the single-source shortest path (SSSP) problem and the all-pairs shortest path problem (APSP). In the SSSP variant, we are searching for paths with the least total weight from a fixed source vertex to every other vertex in the graph. Similarly, the APSP problem asks for a shortest path between every pair of vertices. In this chapter we focus on the all-pairs variant of the problem.

The asymptotically fastest APSP algorithm for dense graphs to date has a running time of $n^3/2^{\Omega(\sqrt{\log n})}$ [14]. For non-negative arc weights and for sparse graphs, there exist asymptotically fast algorithms for worst case inputs [64, 66, 79], and algorithms which are efficient expected-case modifications of Dijkstra's algorithm [21, 48, 63].

For the analysis of the expected-case running-time of shortest-path algorithms, input instances are generated according to a probability model on the set of complete directed graphs with arc weights. In the uniform model, arc weights are drawn at random, independently of each other, according to a common probability distribution. A more general model is the endpoint-independent model [4,74], where, for each vertex v, a sequence of $n-1$ non-negative arc weights is generated by a deterministic or stochastic process and then randomly permuted and assigned to the outgoing arcs of v. In the vertex potential model [15, 17], arc weights can be both positive and negative. This is a probability model with arbitrary real arc weights, but without negative cycles.

In the uniform model with arc weights drawn from the uniform distribution on $[0, 1]$, Hassin and Zemel [40] and Frieze and Grimmett [32] presented algorithms that solve the APSP problem in $O(n^2 \log n)$ expected-case time. Peres *et al.* [63] improved the expected-case running time to $O(n^2)$, which is optimal. In the endpoint-independent model, Spira [74] proved an expected-case time bound of $O(n^2 \log^2 n)$, which was improved by several authors. Takaoka and Moffat [78] improved the running time to $O(n^2 \log n \log\log n)$. Bloniarz [4] described an algorithm with expected-case running time $O(n^2 \log n \log^* n)$. Finally, Moffat and Takaoka [60] and Mehlhorn and Priebe [59] improved the running time to $O(n^2 \log n)$. In the vertex potential model, Cooper *et al.* [17] gave an algorithm with an expected-case running time $O(n^2 \log n)$. All the above algorithms use a Dijkstra-like approach which inherently requires the use of a priority queue. Table 4.1 lists the state of the art algorithms along with their bounds.

In Section 4.1 we deal with the non-negative arc weights and consider the following problem: what is the best way to make use of an SSSP algorithm when solving APSP?

Table 4.1: State of the art all-pairs shortest path algorithms

Worst Case			
	Weights	**Directed**	**Bound**
Chan-Williams [14]	Real	Yes	$n^3/2^{\Omega(\sqrt{\log n})}$
Pettie [64]	Real	Yes	$O(mn + n^2 \log\log n)$
Pettie-Ramachandran [66]	Real	No	$O(mn \log \alpha(m,n))$
Thorup [79]	Integer	No	$O(mn + n^2)$
Expected Case			
	Model	**Bound**	
Peres et al. [63]	Uniform	$O(n^2)$	
Mehlhorn-Priebe [59]	End-point ind.	$O(n^2 \log n)$	
Cooper et al. [17]	Vertex potential	$O(n^2 \log n)$	

There exists some prior work on a very similar subject in the form of an algorithm named the Hidden Paths Algorithm [48]. The Hidden Paths Algorithm is essentially a modification of Dijkstra's algorithm [22] to make it more efficient when solving APSP. Solving the APSP problem by repeated calls to Dijkstra's algorithm requires $O(mn + n^2 \lg n)$ time using Fibonacci heaps [30], with a single run requiring $O(m+n \lg n)$ time. The Hidden Paths Algorithm then reduces the running time to $O(m^* n + n^2 \lg n)$. The quantity m^* represents the number of arcs (u, v) such that (u, v) is included in at least one shortest path. In the Hidden Paths Algorithm this is accomplished by modifying Dijkstra's algorithm, so that it essentially runs in parallel from all vertex sources in a graph, and then reusing the computations performed by other vertices. The idea is simple: we can delay the inclusion of an arc (u, v) as a candidate for forming shortest paths until vertex u has found (u, v) to be the shortest path to v.

As pointed out in [48]: it is known [32, 40, 55] that $m^* = O(n \log n)$ with high probability when the input graph is the complete graph with edge weights chosen independently from any of a large class of probability distributions, including the uniform distribution on the real interval $[0, 1]$ or the uniform distribution on the range $\{1, ..., n^2\}$. It should be also noted that, in unit weight graphs, $m^* = m$ since every arc forms a shortest path.

However, the speedup technique employed by the Hidden Paths Algorithm is only applicable to Dijkstra's algorithm, since it explicitly sorts the shortest path lists by path weights, through the use of a priority queue. As a related algorithm, we also point out that a different measure related to the number of so-called uniform paths (also called locally shortest paths), has also been exploited to yield faster algorithms [21]. The main result of Section 4.1 is a speedup technique similar to the Hidden Paths Algorithm, but without relying explicitly on Dijkstra's algorithm, thus effectively bringing this speedup to any SSSP algorithm when it is used to solve APSP.

In Section 4.2 we analyze certain properties of shortest k-paths in complete graphs, which will be later used to prove expected-case time complexity bounds. In Section 4.3 we remove the condition of non-negative arc weights and study Floyd-Warshall [26,83], a simple dynamic programming algorithm that is frequently used to solve APSP. There exist many optimizations for the Floyd-Warshall algorithm, ranging from better cache performance [82], optimized program-generated code [38], to parallel variants for the GPU [39, 50]. One can also approach APSP through Min-plus matrix multiplication, and practical improvements have been devised to this end through the use of

sorting [57]. In spite of intensive research on efficient implementations of the Floyd-Warshall algorithm, there has not been much focus devoted to improvement of the number of path combinations examined by the algorithm.

In Section 4.4 we perform an empirical evaluation of all mentioned shortest path algorithms, both existing and novel. Finally, in Section 4.5, we study the case of a closely related problem called the all-pairs bottleneck paths problem and propose an algorithm as well as point out a connection to the dynamic transitive closure problem.

Throughout this chapter, the model of computation used in algorithm design and analysis is the comparison-addition model, where the only allowed operations on arc weights are comparisons and additions. Beside operations on weights, we assume a RAM model.

4.1 The Propagation algorithm

It is obvious that the APSP problem can be solved by n calls to an SSSP algorithm. Let us denote the SSSP algorithm as ψ. We can quantify the asymptotic time bound of such an APSP algorithm as $O(nT_\psi(m,n))$ and the asymptotic space bound as $O(S_\psi(m,n))$, where $T_\psi(m,n)$ is the time required by algorithm ψ on graphs with n vertices and m arcs and $S_\psi(m,n)$ is the space requirement of the same algorithm. We assume that the time and space bounds can be written as functions of m and n only. Note that if we are required to store the computed distance matrix, then we need at least $\Theta(n^2)$ additional space. If we account for this, then the space bound becomes $O(S_\psi(m,n) + n^2)$. Throughout this section, we assume that the shortest path weights are unique, although in practice we can break ties by making sure the path with fewer vertices on it is deemed shorter, and in case even that is the same, falling back to some other arbitrary criterion such as the index of the last vertex.

Let $G = (V, A)$ denote a weighted digraph with a non-negative arc weight function $w\colon A \to \mathbb{R}^+$ and $V = \{v_1, v_2, \ldots, v_n\}$. Without loss of generality, we assume that G is strongly connected. Similar to how shortest paths are discovered in Dijkstra's algorithm, we rank shortest paths in nondecreasing order of their weights. We call a path π the k-th shortest path if it is at position k in the length-sorted shortest path list. The list of paths is typically taken to be from a single source to variable target vertices. In contrast, we output paths from variable sources to a single target. (Note that if the desired output were lists of shortest paths out of a vertex, this could be easily accomplished by a preprocessing step where we flip all the arc directions in the graph before running the algorithm.)

Suppose we have an SSSP algorithm ψ. We denote a call of this algorithm by $\psi(V, A, w, s)$ where V and A correspond to the vertex and arc sets, respectively, w is the arc weight function of the graph, and s corresponds to the source vertex. The method we propose works in the fundamental comparison-addition model and does not assume a specific kind of arc weight function, except the requirement that it is non-negative. However, the algorithm ψ that is invoked can be arbitrary, so if ψ requires a different model or a specific weight function, then implicitly by using ψ, our algorithm does as well.

First we give a simpler variant of the algorithm, resulting in bounds $O(mn + nT_\psi(m^* + n, n + 1))$. We limit our interaction with ψ only to execution and reading its output. In Section 4.1.3 we show how to improve the running time by constructing a weighted digraph $G' = (V', A')$ with weight function w' on which we run ψ. There are

two processes involved: the method for solving APSP which runs on G, and the SSSP algorithm ψ which runs on G'. There are $n-1$ phases of the main algorithm, each composed of three steps: (1) prepare the graph G'; (2) run ψ on G'; and (3) interpret the results of ψ. The goal of each phase is to discover exactly one new shortest path per vertex, for a total of n new shortest paths.

Although the proposed algorithm effectively works on $n-1$ new graphs, these graphs are similar to one another. Thus, we can consider the algorithm to work only on a single graph G' with a representation that allows modification of arc weights and introduction of new arcs into G' in $O(1)$ time. The vertex set $V' = V \cup \{\star\}$, where $\star$ is a new vertex unrelated to the graph G, remains fixed throughout the execution of the algorithm. At phase k the set A' consists of all the arcs that are contained in some i-th shortest path for $i < k$, and n new arcs $(\star, v_j)$ for $j \in [n]$. The weights of the new arcs are assigned by the algorithm. We proceed with four definitions which will be motivated later.

Definition 4.1. *The* sorted shortest distance list S_j *is a list of* $2 \leq k \leq n+1$ *pairwise distinct elements, where the first element is always* $(v_j, 0)$ *and the last element is always* $(null, \infty)$*. The remaining elements are of the form* (v_a, δ) *where* $v_a \in V$ *and* $\delta = D_G(v_a, v_j)$*, and are sorted by* δ *in non-decreasing order.*

Definition 4.2. *Given sorted shortest distance lists* $S_1, \ldots, S_n$*, a vertex* $v_a \in V$ *is* a viable vertex for vertex $v_j \in V$ *if for each* $(v_{a'}, \delta') \in S_j$ *we have* $v_a \neq v_{a'}$*.*

Definition 4.3. *Given sorted shortest distance lists* $S_1, \ldots, S_n$*, a pair* $(v_a, \delta) \in S_i$ *is* a viable pair for vertex $v_j \in V$ *if* $(v_i, v_j) \in A$ *and if either* $v_a = null$*, or if* v_a *is a viable vertex for* v_j*.*

Definition 4.4. *Given sorted shortest distance lists* $S_1, ..., S_n$*, a viable pair* $(v_a, \delta) \in S_i$ *for vertex* v_j *is* the currently best pair for vertex v_j *if for every other viable pair* $(v_{a'}, \delta') \in S_p$ *for* v_j *the following holds* $\delta' + w(v_p, v_j) \geq \delta + w(v_i, v_j)$*.*

Next we describe the data structures. Each vertex $v_j \in V$ keeps its sorted shortest distance list S_j, which initially contains only two pairs $(v_j, 0)$ and $(null, \infty)$. It is worth pointing out that the algorithm will never change or remove pairs from S_j, it will only insert new pairs in front of the element $(null, \infty)$. Vertex v_j keeps a pointer $p[i, j]$ for each incoming neighbor $v_i \in V$ in G, which points to an element in the sorted shortest distance list S_i. Initially, each such pointer $p[i, j]$ is set to point to the first element of S_i. In order to traverse these lists, we use $p[i, j].next()$ in pseudocode, to get the next element in S_i.

The first step in each phase of the algorithm is preparation of the graph G'. In this step, each vertex v_j finds a currently best pair (v_a, δ). To do this, vertex v_j for each $v_i \in V$ such that $(v_i, v_j) \in A$ inspects S_i starting at position $p[i, j]$ until it finds a viable pair. Finally, it selects the minimum (as per Definition 4.4) among the found viable pairs. We call this process *reloading*.

Once reloaded we modify the arcs in the graph G'. Let $(v_a, \delta_a) \in S_i$ be the currently best pair for vertex v_j. Then we set $w'(\star, v_j) \leftarrow \delta_a + w(v_i, v_j)$ and call $\psi(V', A', w', \star)$. We assume ψ returns an array Π of length n such that each element $\Pi[j]$ is a pair (v_c, δ) where δ (in pseudocode $\Pi[j].\delta$) is the weight of a shortest path from $\star$ to v_j, and v_c is the second (the first being $\star$) vertex of this path (in pseudocode $\Pi[j].c$). The inclusion of the second encountered vertex is merely a convenience, and can otherwise

easily be obtained by examining the shortest path tree returned by the algorithm. For each vertex $v_j \in V$, let $v_r = \Pi[j].c$ and let $(v_{r'}, \delta_{r'})$ be the currently best pair for vertex v_r found in the reloading step. We append the pair $(v_{r'}, \Pi[j].\delta)$ to S_j. Note that the arcs $(\star, v_j) \in A'$ are essentially shorthands for paths in G. Thus, $v_{r'}$ represents the source of a path in G. We call this process *propagation*.

After propagation, we modify the graph G' as follows. For each vertex $v_j \in V$ such that $\Pi[j].c = v_j$, we check whether the currently best pair $(v_a, \delta_a) \in S_i$ that was selected during the reloading phase is the first element of the list S_i. If it is the first element, then we add the arc (v_i, v_j) into the set A'. This last step simply grows the arc set A' over time to contain all arcs in A that take part in the shortest path computations (which number m^* in total). This concludes the description of the algorithm. Figure 4.1 shows an example of the execution of the algorithm during phase 2 just before the propagation step, and Figure 4.2 shows the graph G' at the same point in time.

We formalize the procedure in pseudocode and obtain Algorithm 1. For a pair $p = (v_k, \delta_k)$ we use the notation $p.v$ to access the vertex and $p.\delta$ to access the distance. Besides the described variables, the algorithm maintains the matrix *viable* which allows to quickly check if a pair is viable. Algorithm 1 produces as output the array of sorted shortest distance lists containing S_j for every vertex v_j. To see why the algorithm correctly computes the shortest paths, we prove the following two lemmata.

Lemma 4.5. *Let $0 < k < n$ be a phase of the algorithm and let the k-th shortest path of vertex $v_j \in V$ consist of the concatenation of the edge (v_i, v_j) and a path from the sorted shortest distance list S_i contained at position $b < k$. Then this path was found during the reloading step of phase k of the algorithm and $\psi(V', A', w', \star)$ finds the arc $(\star, v_j)$ to be the shortest path into v_j.*

Proof. The weight of the arc $(\star, v_j)$ is set to the δ component of the best viable pair found during the reloading step. However, the best viable pair in this case is exactly the k-th shortest path, so the edge $(\star, v_j)$ has the same weight as the k-th shortest path. Now consider the case that some path, other than the arc $(\star, v_j)$ itself, would be found to be a shorter path to v_j by ψ. Since each of the outgoing arcs of $\star$ represent a path in G, this would mean that taking this path and adding the remaining arcs used to reach v_j would constitute a shorter path than the k-th shortest path of v_j. Let us denote the path obtained by this construction as P'. Clearly this is a contradiction unless P' is not the k-th shortest path, i.e., a shorter path connecting the two vertices is already known.

Without loss of generality, assume that $P' = \{(\star, v_i), (v_i, v_j)\}$. However, $w'(P')$ can only be smaller than $w'(\star, v_j)$ if v_j could not find a viable (non-*null*) pair in the list S_i, since otherwise a shorter path would have been chosen in the reloading phase. This means that all vertex sources (the first component of a pair) contained in the list S_i are also contained in the list S_j. Therefore a viable vertex for v_i must also be a viable vertex for v_j. This concludes the proof by contradiction, since the path obtained is indeed the shortest path between the two vertices. □

Lemma 4.6. *For $0 < k < n$, the main for loop of Algorithm 1 (line 10 of pseudocode) at iteration k correctly computes the k-th shortest paths to v_j for all vertices $v_j \in V$.*

Proof. Lemma 4.5 covers the case when the k-th shortest path to vertex v_j consists of the concatenation of a d-th shortest path for $d < k$ of a neighbor vertex v_i of v_j and

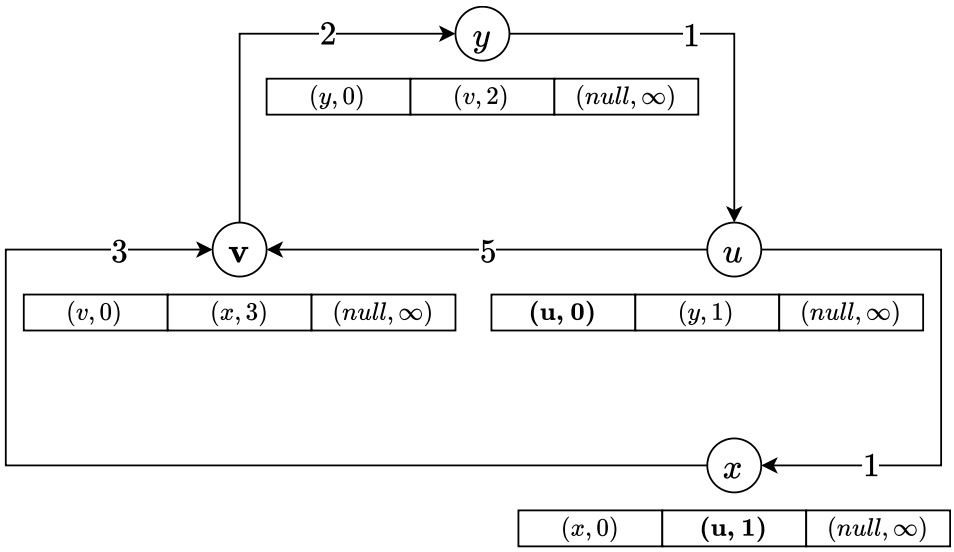


Figure 4.1: The Propagation algorithm at phase 2 just after reloading and before propagation on graph G, which contains vertices u, v, x, and y connected according to the arcs and their weights in the figure. Underneath each vertex is the sorted shortest distance list S belonging to that vertex. Bold pairs represent currently best pairs for v.

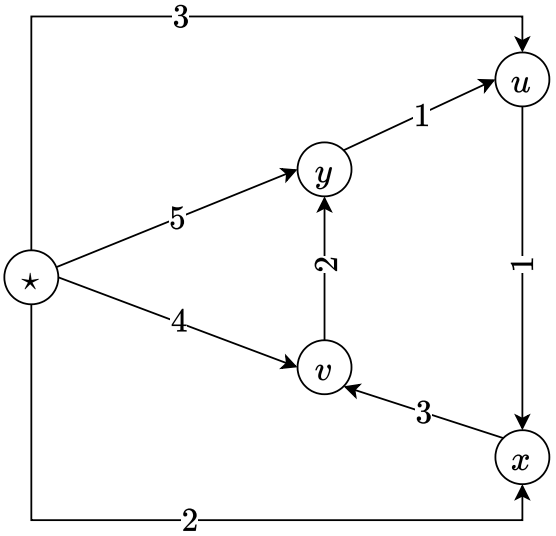


Figure 4.2: The graph G' for the Propagation algorithm at phase 2 after the reloading step, for the graph G illustrated in Figure 4.1.

Algorithm 1 Propagation Algorithm

1: **function** APSP(V, A, ψ)
2: $V' := V \cup \{\star\}$
3: $A' := \{(\star, v_j) : v_j \in V\}$
4: **for all** $v_i \in V$ **do**
5: $S_i := [(v_i, 0), (null, \infty)]$
6: Initialize $n \times n$ boolean matrix *viable* as *true* for each entry
7: **for all** $v_j \in V$ **do**
8: **for all** incoming neighbors v_i of v_j **do**
9: Set $p[i, j]$ to point to first element of S_i
10: **for** $k := 1$ to $n - 1$ **do**
11: **for all** $v_j \in V$ **do** ▷ Reloading
12: $best[j] := (null, \infty)$
13: **for all** incoming neighbors v_i of v_j **do**
14: **while** $\neg viable[j, p[i, j].v]$ and $p[i, j].next().v \neq null$ **do**
15: $p[i, j] := p[i, j].next()$
16: **if** $viable[j, p[i, j].v]$ and $p[i, j].\delta + w(v_i, v_j) < best[j].\delta$ **then**
17: $best[j].v := p[i, j].v$
18: $best[j].\delta := p[i, j].\delta + w(v_i, v_j)$
19: $w'(\star, v_j) := best[j].\delta$
20: $\Pi := \psi(V', A', w', \star)$ ▷ Run SSSP algorithm
21: **for all** $v_j \in V$ **do** ▷ Propagation
22: $S_j.add(\ (best[\Pi[j].c].v,\ \Pi[j].\delta)\ )$
23: $viable[j, best[\Pi[j].c].v] := false$
24: **if** $\Pi[j].c = j$ and $(best[j].v, v_j) \in A$ and $w(best[j].v, v_j) = best[j].\delta$ **then**
25: $A' := A' \cup \{(best[j].v, v_j)\}$
26: $w'(best[j].v, v_j) := w(best[j].v, v_j)$
27: **return** $[S_1, S_2, ..., S_n]$

the edge (v_i, v_j). This leaves two remaining cases, either the k-th path depends on a neighbor's k-th path, or it somehow depends on some neighbor's path at position $e > k$. The latter case is impossible by a straightforward counting argument: a neighbor v_i of v_j contains k shortest paths into v_i with different starting vertices that we can combine with the edge (v_i, v_j) to obtain paths into v_j. Any additional paths, e.g., a $(k+1)$-th path, would be longer because the paths are sorted by their weight. We now consider the case when the k-th path depends on a neighbor's k-th path.

If the k-th path of vertex v_j consists of the concatenation of the arc (v_i, v_j) and the k-th path from the list of v_i, then we can recursively apply the same argument regarding the dependency of the k-th path of vertex v_i on its neighbor's sorted shortest path list. Thus, the path becomes shorter after each such dependency, eventually becoming dependent on a path included at position $0 < b < k$ in a neighbor's sorted shortest path list or simply an empty path designated by the special element at the first position in the list. This path has already been found during the reloading step and is preserved as the shortest path due to Lemma 4.5, thus concluding the proof. □

The following corollary follows directly from a successive application of Lemma 4.6.

Corollary 4.7. *At the conclusion of Algorithm 1, for each vertex $v_j \in V$, the i^{th} entry in the sorted shortest distance list S_j corresponds to the i^{th} shortest path into v_j.*

4.1.1 Time and space complexity

First, we look at the time complexity. The main loop of Algorithm 1 (lines 10–26) performs $n-1$ iterations. The reloading loop (lines 11–19) considers each arc $(v_i, v_j) \in A$ which takes m steps. This amounts to $O(mn)$. Since each sorted shortest distance list is of length $n+1$, each pointer is moved to the next element n times over the execution of the algorithm. There are m pointers, so this amounts to $O(mn)$. Algorithm ψ is executed $n-1$ times, and the graph G' that ψ operates on consists of at most $n+1$ vertices and at most $m^* + n$ arcs. In total, the running time of Algorithm 1 is $O(mn + nT_\psi(m^* + n, n+1))$.

To get the space complexity of Algorithm 1 observe that each vertex keeps track of its sorted shortest distance list, which is of size $n+1$ and amounts to $\Theta(n^2)$ space over all vertices. Since there are exactly m pointers in total, the space needed for them is simply $O(m)$. On top of the costs mentioned, we require as much space as is required by algorithm ψ. In total, the combined space complexity for Algorithm 1 is $O(n^2 + S_\psi(m^* + n, n+1))$.

4.1.2 Implications

We will show how to further improve the time complexity of the algorithm in Section 4.1.3, but already at its current stage, the algorithm reveals an interesting relationship between the complexity of non-negative SSSP and a stricter variant of APSP called *sorted all-pairs shortest path* (SAPSP). The problem $SAPSP(m, n)$ is that of finding shortest paths between all pairs of vertices in a graph with m non-negative weight arcs and n vertices and producing the output in the form of n sorted shortest distance lists, one for each vertex.

Theorem 4.8. *Let T_{SSSP} denote the complexity of the single-source shortest path problem on graphs with m non-negative weight arcs and n vertices. Then the complexity of SAPSP is at most $O(nT_{SSSP})$.*

Proof. Given an algorithm ψ which solves SSSP, we can construct a solution to SAPSP in time $O(nT_\psi(m+n, n+1))$ by using Algorithm 1. We know from Corollary 4.7 that the lists S_j found by the algorithm are ordered by increasing distance from the source. □

What Theorem 4.8 says is that when solving APSP, either we can follow in the footsteps of Dijkstra and visit vertices in increasing distance from the source without worrying about a sorting bottleneck, or that if such a sorting bottleneck exists, then it proves a non-trivial lower bound for the single-source case.

4.1.3 Improving the time bound

The algorithm presented in the previous section has a running time $O(mn + nT_\psi(m^* + n, n+1))$. We show how to bring this down to $O(m \lg n + nT_\psi(m^* + n, n+1))$. We sort each set A_j of incoming arcs of vertex v_j by arc weights in non-decreasing order. By using any off-the-shelf sorting algorithm, this takes $O(m \lg n)$ time.

We only keep pointers $p[i, j]$ for the arcs which are shortest paths between v_i and v_j, and up to one additional arc per vertex for which we do not know whether it is part of a shortest path. This means we skip those neighbors in line 13 of Algorithm 1 for which we do not currently have pointers to. Since arcs are sorted by their weights, a vertex v_j can ignore an arc at position t in the sorted list A_j until the arc at position $t-1$ is either found to be the shortest path between the vertices it connects, or found *not* to be the shortest path. For some arc (v_i, v_j) the former case simply corresponds to using the first element, i.e., v_i, provided by $p[i, j]$ as a shortest path. The latter case on the other hand, is *not* using the first element offered by $p[i, j]$, i.e., finding it is not viable during the reloading phase. Whenever one of these two conditions is met, we include the next arc in the sorted list, and either throw away the previous arc if it was found not to be a shortest path, or keep it otherwise. This means that the total amount of pointers is at most $m^* + n$ at any given time, which is $O(m^*)$, since m^* is at least n. The total amount of time spent by the algorithm then becomes $O(m \lg n + nT_\psi(m^* + n, n+1))$.

Theorem 4.9. *Let ψ be an algorithm that solves the single-source shortest path problem on graphs with non-negative arc weights. Then, the all-pairs shortest path problem on such graphs can be solved in time $O(m \lg n + nT_\psi(m^* + n, n+1))$ and space $O(n^2 + S_\psi(m^* + n, n+1))$ where $T_\psi(m, n)$ is the time required by algorithm ψ on a graph with m arcs and n vertices and $S_\psi(m, n)$ is the space required by algorithm ψ on the same graph.*

Proof. See the discussion above and the initial algorithm and discussion in Section 4.1. □

4.1.4 Directed acyclic graphs with arbitrary weights

A combination of a few techniques yields an $O(m^* n + m \lg n)$ APSP algorithm for directed acyclic graphs (DAGs) with arbitrary arc weights. The first step is to transform the original graph (possibly containing negative-weight arcs) into a graph containing

only non-negative weight arcs through Johnson's [47] reweighting technique. Instead of using Bellman-Ford in the Johnson step, we visit vertices in their topological order, thus obtaining the reweighted graph in $O(m)$ time. Next, we use the improved time bound algorithm as presented in Subsection 4.1.3. For the SSSP algorithm, we again visit vertices according to their topological order. Note that if the graph G is a DAG then G' is also a DAG. The reasoning is simple: the only new arcs introduced in G' are those from $\star$ to each vertex $v_j \in V$. But since $\star$ has no incoming arcs, the acyclicity of the graph is preserved. The time bounds become $O(m)$ for Johnson's step and $O(m \lg n + nT_\psi(m^*+n, n+1))$ for the APSP algorithm where $T_\psi(m^*+n, n+1) = O(m^*)$. Thus, the combined asymptotic running time is $O(m^*n + m \lg n)$. The asymptotic space bound is simply $\Theta(n^2)$.

Theorem 4.10. *All-pairs shortest path on directed acyclic graphs can be solved in time* $O(m^*n + m \lg n)$ *and* $\Theta(n^2)$ *space.*

Proof. See the discussion above. □

4.1.5 Practical optimizations

We outline optimizations that can be performed to speed up the algorithm's running time, but which are not known to improve the asymptotic bound. First, Algorithm 1 is modified slightly, by reconstructing the graph G' at each iteration of the main *for* loop. By doing this, we can use the optimizations described below to reduce the size of G'. This makes sense when ψ has super-linear running time, as the construction only takes $O(m^* + n)$ time per loop iteration. The algorithm's asymptotic bounds remain the same. We denote by *phase* the state of the algorithm just before line 20 in Algorithm 1. It is easy to see that there are exactly $n - 1$ phases of the algorithm.

Size reduction

First, we will show how to reduce the number of arcs contained in G'. A key concept is the concept of a *starving arc.*

Definition 4.11. (Starving arc) *An arc* (v_i, v_j) *is said to be starving at phase* k *if for each* $(v_a, \delta) \in S_i$ *there exists some* $(v_a, \delta') \in S_j$ *at phase* k*.*

The following lemma shows that non-starving arcs can be omitted from G'.

Lemma 4.12. *If an arc* (v_i, v_j) *is not starving at phase* k*, its omission in* G' *does not influence the outcome of the SSSP computation during iteration* k*.*

Proof. Since the arc (v_i, v_j) is not starving at phase k, then by definition of a starving arc, S_i contains a viable pair (v_a, δ) for v_j contained at position $b < k$ in S_i. Recall that the arc $(\star, v_j)$ has weight equal to the weight of the best pair for v_j at phase k. Since (v_a, δ) is a viable pair for v_j, it serves as an upper bound to the best pair, i.e., let $(v_{a'}, \delta') \in S_q$ for some incoming neighbor v_q of v_j, be the best pair for v_j at phase k. Then $\delta' + w(v_q, v_j) \leq \delta + w(v_i, v_j)$. Let W_i be the weight of the shortest path from $\star$ to v_i in G' at iteration k. From Lemma 4.6, we know that W_i is also the weight of the k-th shortest path into v_i in G. The viable pair (v_a, δ) contains the weight of some

b-th shortest path into v_i for $b < k$, and since shortest path weights are monotonically increasing, $\delta \leq W_i$. Combining, we have:

$$\delta' + w(v_q, v_j) \leq \delta + w(v_i, v_j) \leq W_i + w(v_i, v_j).$$

Thus, the inclusion of the arc (v_i, v_j) in G' at iteration k does not influence the SSSP computation, since a shorter or equal-weight path from $\star$ to v_j exists in G'. □

Weight bounding

Even starving arcs can be omitted from G' under certain conditions.

Lemma 4.13. *For a starving arc (v_i, v_j) at phase k, let (v_a, δ) be the last pair in S_i before $(null, \infty)$. Further, let $(v_{a'}, \delta') \in S_q$ be the currently best pair for vertex v_j at phase k. If $\delta + w(v_i, v_j) \geq \delta' + w(v_q, v_j)$, then the omission of (v_i, v_j) in G' does not influence the outcome of the SSSP computation during iteration k.*

Proof. From the definition of $(\star, v_j)$ at phase k and from our lemma condition, we know that $w'(\star, v_j) \leq \delta' + w(v_q, v_j) \leq \delta + w(v_i, v_j)$. Let W_i be the weight of the shortest path from $\star$ to v_i in G' at iteration k. Observe that from the monotonically increasing property of shortest paths, and from the fact that W_i is also the weight of the k-th shortest path into v_i in G we have $\delta \leq W_i$. Thus, the weight of the shortest path from $\star$ to v_j in G' does not change if we remove (v_i, v_j) from G'. □

Further optimizations

It is worth noting that more aggressive strategies could check if the pruned graph conforms to a special case, such as a DAG (e.g., by finding strongly connected components and contracting them), and then use different algorithms to solve the SSSP problem.

4.2 Properties of shortest k-paths in complete graphs

With an aim of proving expected-case time complexity bounds for algorithms discussed later, we first analyze certain properties of shortest k-paths in complete graphs. Primarily we are interested in path length and distance, as well as the outdegree of a shortest path tree.

Before delving into specific properties of paths, we first consider the balls-into-bins process where M balls are thrown uniformly and independently into N bins. The maximum number of balls in any bin is called the maximum load. Let L_i denote the load of bin i, $i \in \{1, 2, \ldots, N\}$. The next lemma, used in Subsection 4.2.3, provides an upper bound on the probability that the maximum load exceeds a certain quantity. It is a simplified version of a standard result, cf. [69], tailored to our present needs. For completeness we provide a proof.

Lemma 4.14. *If M balls are thrown into N bins where each ball is thrown into a bin chosen uniformly at random, then $\mathbb{P}(\max_{1 \leq i \leq N} L_i \geq e^2(M/N + \log N)) = O(1/N)$.*

Proof. First, we have $\mu = \mathbb{E}(L_i) = M/N$, $i = 1, 2, \ldots, N$, and we can write each L_i as a sum $L_i = X_{i1} + X_{i2} + \cdots + X_{iM}$, where X_{ij} is a random variable taking value 1, if ball j is in bin i, and 0 otherwise. Next, since L_i is a sum of independent random

variables taking values in $\{0,1\}$, we can apply, for any particular bin i and for every $c>1$, the multiplicative Chernoff bound [37], which states that

$$\mathbb{P}(L_i \geq c\mu) \leq \left(\frac{e^{c-1}}{c^c}\right)^{\mu} \leq \left(\frac{e}{c}\right)^{c\mu}.$$

We consider two cases, depending on whether $\mu \geq \log N$ or not. Let $\mu \geq \log N$. Take $c = e^2$. Then,

$$\mathbb{P}(L_i \geq e^2\mu) \leq \left(\frac{1}{e}\right)^{e^2\mu} \leq \left(\frac{1}{e}\right)^{e^2 \log N} = \frac{1}{N^{e^2}} \leq \frac{1}{N^2}.$$

Consider now $\mu < \log N$. Take $c = e^2 \frac{N}{M} \log N$. Since $x^{-x} \leq \left(\frac{1}{e}\right)^x$ for all $x \geq e$, we have

$$\begin{aligned}\mathbb{P}\left(L_i \geq \mu e^2 \frac{N}{M} \log N\right) = \mathbb{P}(L_i \geq e^2 \log N) &\leq \left(\frac{e}{c}\right)^{c\mu} = \left(\left(\frac{c}{e}\right)^{-\frac{c}{e}}\right)^{e\mu} \\ &\leq \left(\left(\frac{1}{e}\right)^{\frac{c}{e}}\right)^{e\mu} = \left(\frac{1}{e}\right)^{e^2 \log N} \leq \frac{1}{N^2}.\end{aligned}$$

Putting everything together, we get that

$$\begin{aligned}\mathbb{P}(L_i \geq e^2(\mu + \log N)) &\leq \mathbb{P}(L_i \geq e^2\mu \mid \mu \geq \log N) + \mathbb{P}(L_i \geq e^2 \log N \mid \mu < \log N) \\ &\leq \frac{1}{N^2} + \frac{1}{N^2} = \frac{2}{N^2}.\end{aligned}$$

This, by the union bound, implies that

$$\mathbb{P}\left(\max_{1\leq i\leq N} L_i \geq e^2(\mu + \log N)\right) \leq \sum_{i=1}^{N} \mathbb{P}(L_i \geq e^2(\mu + \log N)) \leq N\frac{2}{N^2} = O(1/N).$$

□

4.2.1 Distances

Let K_n denote a complete digraph on the vertex set $V = \{v_1, v_2, \ldots, v_n\}$. We assume that arc weights of K_n are exponential random variables with mean 1 and that all $n(n-1)$ random arc weights are independent. Due to the memoryless property, it is easier to deal with exponentially distributed arc weights than directly with uniformly distributed arc weights. We define the diameter of a digraph $G = (V, A)$ as the largest shortest path in the digraph, that is, $\max_{u,v\in V} D_G(u, v)$. The aim of this subsection is to show that the diameter of $K_n^{(k)}$, the subdigraph of K_n consisting of all (weighted) arcs that are contained in shortest k-paths in K_n, is $O(\frac{\log n}{k})$ with very high probability. We note, however, that by the same argument as given in the beginning of Subsection 4.2.3, all results derived in this subsection for exponentially distributed arc weights also hold, asymptotically for $[0,1]$-uniformly distributed arc weights as soon as $k \geq \log^2 n$.

We start by considering for a fixed $u \in V$, the maximum distance in $K_n^{(k)}$ between u and other vertices in V. To this end, let $S = \{u, v_1, \ldots, v_k\} \subseteq V$, and let $\overline{S} = V \setminus S$. We clearly have

$$\max_{v\in V} D_{K_n^{(k)}}(u, v) \leq \max_{v\in S} D_{K_n[S]}(u, v) + \max_{v\in\overline{S}} D_{K_n[S\times\overline{S}]}(S, v), \tag{4.1}$$

that is, the maximum distance in $K_n^{(k)}$ between u and other vertices in V is bounded above by the sum of the maximum distance in $K_n[S]$ between u and other vertices in S, and by the maximum distance in $K_n[S \times \overline{S}]$ between S and vertices in $\overline{S}$. We note that $K_n[S]$ is a complete digraph with vertex set S and $K_n[S \times \overline{S}]$ is a complete bipartite digraph with bipartition $(S, \overline{S})$.

To provide an upper bound on $\max_{v \in S} D_{K_n[S]}(u, v)$, we use the following result, which follows from the equation (2.8) in the proof of Theorem 1.1 of Janson [46].

Theorem 4.15 (Janson [46, Theorem 1.1])**.** *Let $u \in V$ be a fixed vertex of K_n. Then for every $a > 0$, we have*

$$\mathbb{P}\left(\max_{v \in V} D_{K_n}(u, v) \geq \frac{a \log n}{n} \right) = O(e^a n^{2-a} \log^2 n).$$

Lemma 4.16. *Let $8 \leq k \leq n$, and let $S \subseteq V$ with $|S| = k$. Then, for a fixed $u \in S$ and for any constant $c > 0$, we have*

$$\mathbb{P}\left(\max_{v \in S} D_{K_n[S]}(u, v) \geq \frac{c \log n}{k} \right) = O(n^{2-c/2} \log^2 n).$$

Proof. By Theorem 4.15, for any $a > 0$ we have

$$\mathbb{P}\left(\max_{v \in S} D_{K_n[S]}(u, v) \geq \frac{a \log k}{k} \right) = O(e^a k^{2-a} \log^2 k).$$

Setting $a = c \log n / \log k$ we get

$$e^a k^{2-a} \log^2 k = e^{c \log n / \log k} k^2 k^{-c \log n / \log k} \log^2 k \leq (e^{\log n})^{c/2} k^2 (k^{\log_k n})^{-c} \log^2 k.$$

In the last step we used the fact that $1/\log k \leq 1/2$ for $k \geq 8$ and that $\log n / \log k = \log_k n$. Furthermore,

$$(e^{\log n})^{c/2} k^2 (k^{\log_k n})^{-c} \log^2 k = n^{c/2} k^2 n^{-c} \log^2 k = O(n^{2-c/2} \log^2 n),$$

and the result follows. □

Next, we provide an upper bound on $\max_{v \in \overline{S}} D_{K_n[S \times \overline{S}]}(S, v)$.

Lemma 4.17. *Let $1 \leq k \leq n$, let $S \subseteq V$ with $|S| = k$, and let $\overline{S} = V \setminus S$. Then for any constant $c > 0$, we have*

$$\mathbb{P}\left(\max_{v \in \overline{S}} D_{K_n[S \times \overline{S}]}(S, v) \geq \frac{c \log n}{k} \right) = O(n^{1-c} \log n).$$

Proof. Let $Z = \max_{v \in \overline{S}} D_{K_n[S \times \overline{S}]}(S, v)$. Arguing similarly as in the proof of Theorem 1.1 of Janson [46], Z is distributed as

$$\sum_{j=k}^{n-1} X_j,$$

where X_j are independent exponentially distributed random variables with mean $\frac{1}{k(n-j)}$. First, for any constant $c > 0$, the Chernoff bound [37] states that

$$\mathbb{P}(Z \geq \frac{c \log n}{k}) \leq e^{-tc \log n} \mathbb{E}(e^{ktZ}).$$

Further, for $-\infty < t \leq 1$, we have from Janson [46, Equation 2.7]

$$\mathbb{E}(e^{ktZ}) = \prod_{j=k}^{n-1} \mathbb{E}(e^{ktX_j}) = \prod_{j=k}^{n-1} \left(1 - \frac{t}{n-j}\right)^{-1}.$$

Using the inequality $-\log(1-x) \leq x + x^2$ for all $0 \leq x \leq 1/2$, we can bound, for all $0 < t < 1$ and $k \leq j \leq n-2$, each term $(1 - t/(n-j))^{-1}$ as follows

$$\left(1 - \frac{t}{n-j}\right)^{-1} = \exp\left(-\log\left(1 - \frac{t}{n-j}\right)\right) \leq \exp\left(\frac{t}{n-j} + \left(\frac{t}{n-j}\right)^2\right).$$

This gives us

$$\begin{aligned}\mathbb{P}(Z \geq \frac{c\log n}{k}) &\leq (1-t)^{-1} \exp\left(-tc\log n + \sum_{j=k}^{n-2}\left(\frac{t}{n-j} + \left(\frac{t}{n-j}\right)^2\right)\right) \\ &= (1-t)^{-1}\exp(-tc\log n + t\log(n-k) + O(1)).\end{aligned}$$

Taking $t = 1 - 1/\log n$, we finally get

$$\mathbb{P}(Z \geq \frac{c\log n}{k}) \leq (1/\log n)^{-1}\exp(-c\log n + \log n + O(1)) = O(n^{1-c}\log n).$$

□

We are now ready to show that the diameter of $K_n^{(k)}$ is $O(\log n/k)$ with very high probability.

Theorem 4.18. *Let $8 \leq k \leq n$. Then, for any constant $c > 0$, we have*

$$\mathbb{P}\left(\max_{u,v\in V} D_{K_n^{(k)}}(u,v) \geq \frac{c\log n}{k}\right) = O(n^{3-c/4}\log^2 n).$$

Proof. Let $S = \{u, v_1, \ldots, v_k\} \subseteq V$, let $\overline{S} = V \setminus S$, and write $\alpha = \frac{c\log n}{k}$. Then, by inequality (4.1), we have

$$\begin{aligned}\mathbb{P}\left(\max_{v\in V} D_{K_n^{(k)}}(u,v) \geq \alpha\right) &\leq \mathbb{P}\left(\max_{v\in S} D_{K_n[S]}(u,v) + \max_{v\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v) \geq \alpha\right) \\ &\leq \mathbb{P}\left(\max_{v\in S} D_{K_n[S]}(u,v) \geq \frac{\alpha}{2}\right) + \mathbb{P}\left(\max_{v\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v) \geq \frac{\alpha}{2}\right).\end{aligned}$$

By Lemma 4.16, we have

$$\mathbb{P}\left(\max_{v\in S} D_{K_n[S]}(u,v) \geq \frac{\alpha}{2}\right) = O(n^{2-c/4}\log^2 n),$$

and, by Lemma 4.17,

$$\mathbb{P}\left(\max_{u\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v) \geq \frac{\alpha}{2}\right) = O(n^{1-c/2}\log n).$$

Putting everything together, we get

$$\mathbb{P}\left(\max_{v\in V} D_{K_n^{(k)}}(u,v) \geq \alpha\right) = O(n^{2-c/4}\log^2 n),$$

which, by the union bound, implies

$$\mathbb{P}\left(\max_{u,v\in V} D_{K_n^{(k)}}(u,v) \geq \alpha\right) \leq n\mathbb{P}\left(\max_{u\in V} D_{K_n^{(k)}}(v,u) \geq \alpha\right) = O(n^{3-c/4}\log^2 n).$$

□

4.2.2 Lengths

Let all arc weights of K_n be either independent $[0,1]$-uniform random variables or independent exponential random variables with mean 1. In this subsection, we bound the length of the longest shortest k-path in K_n.

The proof of our next lemma follows directly from Theorem 1.1 of Addario-Berry et al. [1] on the longest shortest path in K_n.

Theorem 4.19 (Addario-Berry et al. [1, Theorem 1.1])**.** *The following two properties hold:*

(i) For every $t > 0$, we have

$$\mathbb{P}\left(\max_{u,v \in V} |u \rightsquigarrow v| \geq \alpha^* \log n + t \right) \leq e^{\alpha^* + t/\log n} e^{-t},$$

where $\alpha^* \approx 3.5911$ is the unique solution of $\alpha \log \alpha - \alpha = 1$.

(ii) $\mathbb{E}(\max_{u,v \in V} |u \rightsquigarrow v|) = O(\log n)$.

Lemma 4.20. *The following two properties hold:*

(i) For every $c > 5$ and $8 \leq k \leq n$, we have $\mathbb{P}(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v| \geq c \log n) = O(n^{2-c/2})$.

(ii) $\mathbb{E}(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v|) = O(\log k)$.

Proof. Let $S = \{v_1, v_2, \ldots, v_k\}$. Since every shortest k-path in K_n is of the form $u \to w \overset{k}{\rightsquigarrow} z \to v$ or $u \to v$, we have

$$\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v| \leq \max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2. \tag{4.2}$$

By (i) of Theorem 4.19, for any $t > 0$,

$$\mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| \geq \alpha^* \log k + t \right) \leq e^{\alpha^* + t/\log k} e^{-t},$$

where $\alpha^* \approx 3.5911$ is the unique solution of $\alpha \log \alpha - \alpha = 1$. Using $t = (c - \alpha^*) \log n - 2$ gives us

$$\begin{aligned} \mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2 \geq \alpha^* \log(k/n) + c \log n \right) &\leq e^{\alpha^* - 2/\log k + 2} e^{(c-\alpha^*)(\frac{\log n}{\log k} - \log n)} \\ &\leq e^{\alpha^* - 2/\log k + 2} \left(e^{\log n}\right)^{1/2(\alpha^* - c)} \\ &= O(n^{2-c/2}). \end{aligned}$$

By inequality (4.2), we have

$$\begin{aligned} \mathbb{P}\left(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v| \geq c \log n \right) &\leq \mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2 \geq c \log n \right) \\ &\leq \mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2 \geq \alpha^* \log(k/n) + c \log n \right), \end{aligned}$$

and (i) follows.

To prove (ii), we note that, by (ii) of Theorem 4.19, $\mathbb{E}(\max_{u,v \in S} |u \overset{k}{\rightsquigarrow} v|) = O(\log k)$ and, by inequality (4.2), the result follows. □

4.2.3 Maximum outdegree

Let arc weights of K_n be independent $[0,1]$-uniform random variables. Our goal in this subsection is to show that the maximum outdegree of a shortest path tree OUT_k in $K_n^{(k)}$ is $O(\log k + (n-k)/k)$ with high probability for all $k \geq \log^2 n$.

Let now $S = \{v_1, v_2, \ldots, v_k\}$ and $\overline{S} = V \setminus S$. We can consider OUT_k as consisting of the subtree $OUT_k[S]$ to which each vertex from $\overline{S}$ is attached as a leaf. To see how these vertices are attached to $OUT_k[S]$, let us assume for the moment that arc weights are exponentially distributed with mean 1. Then, due to the memoryless property of the exponential distribution, a vertex $v \in \overline{S}$ is attached with equal probability to any vertex in S, say a^v. Let $(K_n[S \times \overline{S}])^*$ be the subdigraph of $K_n[S \times \overline{S}]$ with the set V of vertices and the set $\{a^v \mid v \in \overline{S}\}$ of arcs. By observing that $OUT_k[S]$ is a subdigraph of the graph $(K_n[S])^{(k)}$ consisting of all arcs that are shortest paths in $K_n[S]$, we have

$$\Delta(OUT_k) \leq \Delta((K_n[S])^{(k)}) + \Delta((K_n[S \times \overline{S}])^*). \tag{4.3}$$

To extend the latter bound to uniform distribution, we use a standard coupling argument as in [1]. Let U be a random variable uniform on $[0,1]$. Then $-\log(1-U)$ is an exponential random variable with mean 1, and so we can couple the exponential arc weights $W'(u,v)$ to uniform arc weights $W(u,v)$ by setting $W'(u,v) = -\log(1 - W(u,v))$. As $x \leq -\log(1-x) \leq x + 2x^2$ for all $0 \leq x \leq 1/2$, we have that, for all arcs (u,v) of K_n, $|W'(u,v) - W(u,v)| = O((W'(u,v))^2)$, uniformly for all $W'(u,v) \leq 1/2$. In particular, if $W'(u,v) \leq 12\log n/k$, say, and $k \geq \log^2 n$, then $|W'(u,v) - W(u,v)| = O(1/\log^2 n)$ for n large enough, and so for a path P with $O(\log n)$ vertices and with $W'(P) \leq 12\log n/k$, we have

$$|W'(P) - W(P)| = O(1/\log n)$$

for n large enough. By Theorem 4.18, with very high probability a shortest $(k-1)$-path in K_n with the exponential arc weights has weight less than $12\log n/k$, while by (i) of Lemma 4.20, with very high probability it has $O(\log n)$ vertices. It then follows easily that, for all $k \geq \log^2 n$, the bound as in (4.3) holds for uniform distribution, as well.

The following result on the maximum outdegree in the subgraph $(K_n[S])^{(k)}$ of the complete graph $K_n[S]$ on k vertices with $[0,1]$-uniform arc weights can be found in Peres et al. [63].

Lemma 4.21 (Peres et al. [63, Lemma 5.1])**.** *Let $1 \leq k \leq n$ and let $S \subseteq V$ with $|S| = k$. Then, for every $c > 6$, we have $\mathbb{P}(\Delta((K_n[S])^{(k)}) > c\log k) = O(k^{1-c/6})$.*

The maximum outdegree in $(K_n[S \times \overline{S}])^*$ is directly related to the maximum load in the balls-into-bins process, which is used in the proof of the following lemma.

Lemma 4.22. *Let $1 \leq k \leq n$, let $S \subseteq V$ with $|S| = k$, and let $\overline{S} = V \setminus S$. Then,*

$$\mathbb{P}(\Delta((K_n[S \times \overline{S}])^*) \geq e^2((n-k)/k + \log k)) = O(k^{-1}).$$

Proof. Consider vertices from S as bins and vertices from $\overline{S}$ as balls. For $v \in \overline{S}$, each arc in $S \times \{v\}$ is equally likely to be the shortest, so v is thrown into a bin chosen uniformly at random, and the result follows by Lemma 4.14 for $N = k$ and $M = n - k$. □

We are now ready to prove the main result of this subsection.

Theorem 4.23. *For every $k \geq \log^2 n$, we have*

$$\mathbb{P}\left(\Delta(OUT_k) \geq (e^2+12)\log k + e^2\frac{n-k}{k}\right) = O(k^{-1}).$$

Proof. Let $S = \{v_1, v_2, \ldots, v_k\}$ and $\overline{S} = V \setminus S$. Further, let us write $\alpha = 12\log k$ and $\beta = e^2((n-k)/k\log k)$. By the inequality (4.3), for every $k \geq \log^2 n$, we have

$$\begin{aligned}\mathbb{P}(\Delta(OUT_k) \geq \alpha+\beta) &\leq \mathbb{P}(\Delta((K_n[S])^{(k)}) + \Delta((K_n[S\times\overline{S}])^*) \geq \alpha+\beta)\\ &\leq \mathbb{P}(\Delta((K_n[S])^{(k)}) \geq \alpha) + \mathbb{P}(\Delta((K_n[S\times\overline{S}])^*) \geq \beta).\end{aligned}$$

By Lemma 4.21, we have $\mathbb{P}(\Delta((K_n[S])^{(k)}) \geq \alpha) \leq 1/k$. Similarly, by Lemma 4.22, we have $\mathbb{P}(\Delta((K_n[S\times\overline{S}])^*) \geq \beta) \leq 1/k$. Hence, $\mathbb{P}(\Delta(OUT_k) \geq \alpha+\beta) \leq 1/k + 1/k = O(1/k)$. □

4.3 Speeding up the Floyd-Warshall algorithm

In general, the APSP problem can be solved by using the technique of **relaxation**. Relaxation consists of testing whether we can improve the weight of a shortest path from u to v found so far by going via w, and updating it if necessary.

The Floyd-Warshall algorithm [26, 83] as presented in Algorithm 2 is a relaxation-based dynamic programming approach to solve APSP on an n-vertex graph $G(V,A)$ where the vertex set has a fixed ordering $V = \{v_1, v_2, \ldots, v_n\}$. The arcs are represented by a weight matrix W, where $W[i,j] = w(v_i, v_j)$ if $(v_i, v_j) \in A$ and ∞ otherwise. Its running time is $O(n^3)$ due to three nested **for** loops. To understand the algorithm, it is helpful to keep in mind that after the k-th iteration, $W[i,j]$ is the shortest distance from v_i to v_j going only through vertices $\{v_1, \ldots, v_k\}$. Unlike Dijkstra's algorithm and the algorithm in Section 4.1, Floyd-Warshall can find shortest paths in graphs which contain negatively-weighted arcs.

Algorithm 2 Floyd-Warshall Algorithm

```
1: procedure FLOYD-WARSHALL(W)
2:     for k := 1 to n do
3:         for i := 1 to n do
4:             for j := 1 to n do
5:                 if W[i,k] + W[k,j] < W[i,j] then          ▷ Relaxation
6:                     W[i,j] := W[i,k] + W[k,j]
       return W
```

4.3.1 The Tree algorithm

Recall that in Section 2.2 we defined $G^{(k)}$ as the subdigraph of G comprised of the set of all arcs that are part of some shortest k-path in G. Let us consider the algorithm at iteration k, and let OUT_k denote a shortest path tree rooted at vertex v_k in $G^{(k-1)}$ (see Algorithm 4 for the exact construction). Intuitively, one might expect that the relaxation in lines 5-6 would not always succeed in lowering the value of W_{ij} which currently contains the weight $w(v_i \overset{k-1}{\rightsquigarrow} v_j)$. This is precisely the observation that we

exploit to arrive at a more efficient algorithm: instead of simply looping through every vertex of V in line 4, we perform a depth-first traversal of OUT_k. This permits us to skip iterations which provably cannot lower the current value of $W[i,j]$. As the following lemma shows, if $w(v_i \overset{k}{\rightsquigarrow} v_j) = w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_i \overset{k}{\rightsquigarrow} v_y) = w(v_i \overset{k-1}{\rightsquigarrow} v_y)$ for all vertices v_y in the subtree of v_j in OUT_k. Refer to Figure 4.3 for an illustration of the paths mentioned.

Lemma 4.24. *Let $v_j \in V \setminus \{v_k\}$ be some non-leaf vertex in OUT_k, $v_y \neq v_j$ an arbitrary vertex in the subtree of v_j in OUT_k, and $v_i \in V \setminus \{v_k\}$. Consider a walk $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$. If $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_y)$.*

Proof. Since v_j is neither a leaf nor the root of OUT_k that means it is an internal vertex on a $(k-1)$-path, so we have $j < k$ and thus $v_i \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y$ is a $(k-1)$-path between v_i and v_y. Because $v_i \overset{k-1}{\rightsquigarrow} v_y$ is a shortest $(k-1)$-path between v_i and v_y, we have

$$\begin{aligned} w(v_i \overset{k-1}{\rightsquigarrow} v_y) &\leq w(v_i \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y) = w(v_i \overset{k-1}{\rightsquigarrow} v_j) + w(v_j \overset{k-1}{\rightsquigarrow} v_y) \\ &\leq w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) + w(v_j \overset{k-1}{\rightsquigarrow} v_y) = w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y), \end{aligned}$$

where the last inequality follows by the assumption. Finally, since v_y is in the subtree rooted at v_j, we have $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y = v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y$, and so the last term is equal to $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y)$. This completes the proof. □

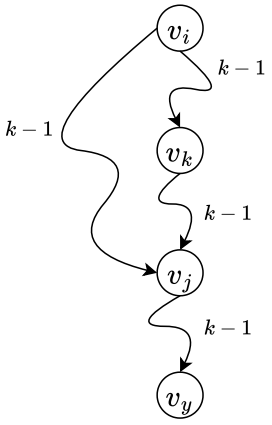


Figure 4.3: Illustration of paths for Lemma 4.24. The squiggly lines between vertices in the figure are $(k-1)$-paths.

The pseudocode of the modified Floyd-Warshall algorithm augmented with the tree OUT_k, named the Tree algorithm, is given in Algorithm 3. To perform depth-first search we first construct the tree OUT_k in line 4 using ConstructOut given in Algorithm 4. For the construction of tree OUT_k an additional matrix π is used, where $\pi[i,j]$ specifies the penultimate vertex on a shortest k-path from v_i to v_j (i.e., the vertex immediately "before" v_j)[1]. More precisely, the tree OUT_k is obtained from π by making v_i a child of $v_{\pi[k,i]}$ for all $i \neq k$. This takes $O(n)$ time. Finally, we replace the iterations in lines 4-6 in Algorithm 2 with a depth-first tree traversal of OUT_k in lines 6-14 in Algorithm 3. Note that if, for a given i and a child v_j, the condition in line 11 evaluates to false we do not traverse the subtree of v_j in OUT_k.

[1] C.f. $\pi_{ij}^{(k)}$ in [18, Sec. 25.2].

Algorithm 3 Tree Algorithm

```
1: procedure TREE(W)
2:     Initialize π, an n × n matrix, as π[i, j] := i.
3:     for k := 1 to n do
4:         OUT_k := CONSTRUCTOUT(k, π)
5:         for i := 1 to n do
6:             Stack := empty
7:             Stack.push(v_k)
8:             while Stack ≠ empty do
9:                 v_x := Stack.pop()
10:                for all children v_j of v_x in OUT_k do
11:                    if W[i, k] + W[k, j] < W[i, j] then          ▷ Relaxation
12:                        W[i, j] := W[i, k] + W[k, j]
13:                        π[i, j] := π[k, j]
14:                        Stack.push(v_j)
       return W
```

Algorithm 4 ConstructOut Algorithm

```
1: procedure CONSTRUCTOUT(k, π)
2:     Initialize n empty trees: OUT_1, ..., OUT_n.
3:     for i := 1 to n do
4:         OUT_i.Vertex := v_i
5:         if i ≠ k then
6:             OUT_i.Parent := OUT_{π[k,i]}
       return OUT_k
```

Corollary 4.25. *The Tree algorithm correctly computes all-pairs shortest paths.*

Proof. The correctness of the algorithm follows directly from Lemma 4.24. □

Time complexity

Let T_k denote the running time of the algorithm Tree(W) in lines 4-14 at iteration k. As already said, line 4 requires $O(n)$ time. To estimate the time complexity of lines 5-14, we charge the vertex v_x in line 9 by the number of its children. This pays for lines 10-14. Furthermore, this means that on the one hand leaves are charged nothing, while on the other hand nobody is charged for the root v_k. To this end, let $SP_k^{(k)}$ be the set of all shortest k-paths that contain v_k and end at some vertex in the set $\{v_1, v_2, \ldots, v_k\}$. In case there are multiple shortest paths between a pair of vertices, we assume that the algorithm at the end of iteration k (implicitly) records, for each pair of vertices, a shortest k-path between them and thus, this path becomes the representative of a class of all other shortest k-paths between them. Now v_x in line 9 is charged at most $|SP_k^{(k)}|$ times over all iterations of i. Since the number of children of v_x is bounded from above by $\Delta(OUT_k)$, we can bound T_k from above by

$$T_k \leq |SP_k^{(k)}| \cdot \Delta(OUT_k) + O(n). \tag{4.4}$$

Practical improvement

Observe that in Algorithm 3 vertices of OUT_k are visited in a depth-first search (DFS) order, which is facilitated by using the stack. However, this requires pushing and popping of each vertex, as well as reading of all its children in OUT_k. We can avoid this by precomputing two read-only arrays *dfs* and *skip* to support the traversal of OUT_k. The array *dfs* consists of OUT_k vertices as visited in the DFS order. On the other hand, the array *skip* is used to skip OUT_k subtree when relaxation in line 11 of Algorithm 3 does not succeed.

In detail, for a vertex v_z, *skip*$[z]$ contains the index in *dfs* of the first vertex after v_z in the DFS order that is not a descendant of v_z in OUT_k. Utilizing the arrays outlined above, we traverse OUT_k by scanning *dfs* in left-to-right order and using *skip* whenever a relaxation is not made. Consequently, we perform only two read operations per visited vertex. This has led to a roughly 20% faster running time. It should be pointed out that the asymptotic time remains the same, as this is solely a technical optimization.

4.3.2 The Hourglass algorithm

We can further improve the Tree algorithm by using another tree. The second tree, denoted by IN_k (see Algorithm 7 for the exact construction), is similar to OUT_k, except that it is a shortest path "tree" for paths $v_i \overset{k-1}{\rightsquigarrow} v_k$ for each $v_i \in V \setminus \{v_k\}$. Strictly speaking, this is not a tree, but if we reverse the directions of the arcs, it turns it into a tree with v_k as the root. Traversal of IN_k is used as a replacement of the **for** loop on variable i in line 5 of Algorithm 3 (in line 3 of Algorithm 2). As the following lemma shows, if $w(v_i \overset{k}{\rightsquigarrow} v_j) = w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_y \overset{k}{\rightsquigarrow} v_j) = w(v_y \overset{k-1}{\rightsquigarrow} v_j)$ for all vertices v_y in the subtree of v_i in IN_k. Refer to Figure 4.4 for an illustration of the paths mentioned.

Lemma 4.26. *Let $v_i \in V \setminus \{v_k\}$ be some non-leaf vertex in IN_k and let $v_y \neq v_i$ be an arbitrary vertex in the subtree of v_i in IN_k, and $v_j \in V \setminus \{v_k\}$. Consider a walk $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$. If $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_y \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_y \overset{k-1}{\rightsquigarrow} v_j)$.*

Proof. Due to the choice of v_i and v_y we have: $v_y \overset{k-1}{\rightsquigarrow} v_k = v_y \overset{k-1}{\rightsquigarrow} v_i \overset{k-1}{\rightsquigarrow} v_k$. We want to show that:

$$w(v_y \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j).$$

Observe that $i < k$, since v_i is neither a leaf nor the root of IN_k. Thus we have:

$$w(v_y \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_j).$$

Together with the assumption $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, we get the desired inequality:

$$w(v_y \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j).$$

□

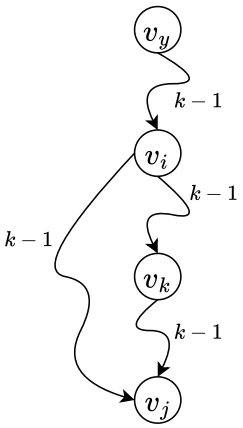


Figure 4.4: Illustration of paths for Lemma 4.26. The squiggly lines between vertices in the figure are $(k-1)$-paths.

The pseudocode of the modified Floyd-Warshall algorithm augmented with the trees OUT_k and IN_k, named the Hourglass algorithm[2], is given in Algorithms 5 and 6. To construct IN_k efficiently, we need to maintain an additional matrix $\phi[i,j]$ which stores the second vertex on the path from v_i to v_j (cf. π and $\pi[i,j]$). Algorithm 7 constructs IN_k similarly to the construction of OUT_k, except that we use the matrix $\phi[i,k]$ instead. The only extra space requirement of the Hourglass algorithm that bears any significance is the matrix ϕ, which does not deteriorate the space complexity of $O(n^2)$. The depth-first traversal on IN_k is performed by a recursion on each child of v_k in line 8 of Algorithm 5. In the recursive step, given in Algorithm 6, we can prune OUT_k as follows: if v_i is the parent of v_y in IN_k and $v_i \overset{k-1}{\rightsquigarrow} v_j \leq v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$, then the subtree of v_j can be removed from OUT_k, while inspecting the subtree of v_i in IN_k. Before the return from the recursion the tree OUT_k is reconstructed to the form it was passed as a parameter to the function.

In practice, the recursion can be avoided by using an additional stack, which further speeds up an implementation of the algorithm.

[2]The hourglass name comes from placing IN_k tree atop the OUT_k tree, which gives it an hourglass-like shape, with v_k being at the neck.

Algorithm 5 Hourglass Algorithm

```
1: procedure HOURGLASS(W)
2:     Initialize π, an n × n matrix, as π[i, j] := i.
3:     Initialize φ, an n × n matrix, as φ[i, j] := j.
4:     for k := 1 to n do
5:         OUT_k := CONSTRUCTOUT(k, π)
6:         IN_k := CONSTRUCTIN(k, φ)
7:         for all children v_i of v_k in IN_k do
8:             RECURSEIN(W, π, φ, IN_k, OUT_k, v_i)
       return W
```

Algorithm 6 RecurseIn Algorithm

```
1: procedure RECURSEIN(W, π, φ, IN_k, OUT_k, v_i)
2:     Stack := empty
3:     Stack.push(v_k)
4:     while Stack ≠ empty do
5:         v_x := Stack.pop()
6:         for all children v_j of v_x in OUT_k do
7:             if W[i, k] + W[k, j] < W[i, j] then                ▷ Relaxation
8:                 W[i, j] := W[i, k] + W[k, j]
9:                 π[i, j] := π[k, j]
10:                φ[i, j] := φ[i, k]
11:                Stack.push(v_j)
12:            else
13:                Remove the subtree of v_j from OUT_k.          ▷ Applies Lemma 4.26
14:    for all children v_i' of v_i in IN_k do
15:        RECURSEIN(W, π, φ, IN_k, OUT_k, v_i')
16:    Restore OUT_k by reverting changes done by all iterations of line 13.
```

Algorithm 7 ConstructIn Algorithm

```
1: procedure CONSTRUCTIN(k, φ)
2:     Initialize n empty trees: IN_1, ..., IN_n.
3:     for i := 1 to n do
4:         IN_i.Vertex := v_i
5:         if i ≠ k then
6:             IN_i.Parent := IN_φ[i,k]
       return IN_k
```

Corollary 4.27. *The Hourglass algorithm correctly computes all-pairs shortest paths.*

Proof. Observe that lines 6-11 of Algorithm 6 are effectively the same as in Algorithm 3. Line 13 of Algorithm 6 does not affect the correctness of the algorithm due to Lemma 4.26, which states that, for any $v_{i'}$ that is a child of v_i in IN_k, these comparisons can be skipped, as they cannot lead to shorter paths. However, Lemma 4.26 does not apply to a sibling v_{i*} of v_i, arising from line 7 of Algorithm 5. Therefore line 16 restores the tree OUT_k, which maintains the correctness of the algorithm. □

Finally, note that the worst-case time complexity of the Hourglass (and Tree) algorithm remains $O(n^3)$. The simplest example of this is when all shortest paths are the arcs themselves, at which point all leaves are children of the root and the tree structure never changes.

4.3.3 Expected-case analysis

We perform an expected-case analysis of the Tree algorithm (Algorithm 3) for the complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0, 1]$. Recall that $SP_k^{(k)}$ is the set of all shortest k-paths that contain v_k and end at some vertex in the set $\{v_1, v_2, \ldots, v_k\}$. We first show that the expected number of paths in $SP_k^{(k)}$ is $O(n \log k)$.

Lemma 4.28. *For each $k = 1, 2, \ldots, n$, we have $\mathbb{E}(|SP_k^{(k)}|) = O(n \log k)$.*

Proof. For $v_i \in V$, let $SP_i^{(k)}$ denote the set of all shortest k-paths that contain v_i and end at some vertex in the set $\{v_1, v_2, \ldots, v_k\}$. Note, that

$$\sum_{i=1}^{k} |SP_i^{(k)}| \leq \sum_{s=1}^{n} \sum_{t=1}^{k} |v_s \overset{k}{\rightsquigarrow} v_t|.$$

To see why the above is true, let $P = v_s, v_{i_1}, v_{i_2}, \ldots, v_{i_w}, v_t$ be the k-shortest path from v_s to v_t, where $s \in \{1, 2, \ldots, n\}$ and $t \in \{1, 2, \ldots, k\}$. Then this path is counted in sets $SP_{i_1}^{(k)}, SP_{i_2}^{(k)}, \ldots, SP_{i_w}^{(k)}, SP_t^{(k)}$ (if $s < k + 1$ then also in $SP_s^{(k)}$). Thus all vertices on P are present in at most $|P|$ sets on the left side.

By symmetry, we have $\mathbb{E}(|SP_i^{(k)}|) = \mathbb{E}(|SP_t^{(k)}|)$ for arbitrary $i, t \in \{1, 2, \ldots, k\}$, and hence by linearity of expectation

$$k\mathbb{E}(|SP_k^{(k)}|) = \sum_{i=1}^{k} \mathbb{E}(|SP_i^{(k)}|) \leq \sum_{s=1}^{n} \sum_{t=1}^{k} \mathbb{E}(|v_s \overset{k}{\rightsquigarrow} v_t|) \leq kn\mathbb{E}(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v|).$$

By (ii) of Lemma 4.20, we get that $\mathbb{E}(|SP_k^{(k)}|) = O(n \log k)$. □

We are now ready to analyse the expected time of the Tree algorithm.

Theorem 4.29. *The Tree algorithm has an expected-case running time of $O(n^2 \log^2 n)$ for complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0, 1]$.*

Proof. To estimate the number T_k of comparisons at iteration k, we consider two cases. First, for $k < \log^2 n$ we bound T_k from above by n^2. Second, we estimate $\mathbb{E}(T_k)$ for $k \geq \log^2 n$. For every $c > 0$, we have

$$\begin{aligned}\mathbb{E}(T_k) = \mathbb{E}(T_k \mid \Delta(OUT_k) < c) \cdot \mathbb{P}(\Delta(OUT_k) < c) \\ + \mathbb{E}(T_k \mid \Delta(OUT_k) \geq c) \cdot \mathbb{P}(\Delta(OUT_k) \geq c).\end{aligned}$$

Using inequality (4.4) we get

$$\begin{aligned}\mathbb{E}(T_k \mid \Delta(OUT_k) < c) &\leq \mathbb{E}((|SP_k^{(k)}| \cdot \Delta(OUT_k) + O(n)) \mid \Delta(OUT_k) < c) \\ &\leq c \cdot \mathbb{E}(|SP_k^{(k)}|) + O(n).\end{aligned}$$

As T_k is always at most n^2, we have $\mathbb{E}(T_k \mid \Delta(OUT_k) \geq c) \leq n^2$. Further, taking into account that $\mathbb{P}(\Delta(OUT_k) < c) \leq 1$, we get

$$\mathbb{E}(T_k) \leq c \cdot \mathbb{E}(|SP_k^{(k)}|) + O(n) + n^2 \cdot \mathbb{P}(\Delta(OUT_k) \geq c).$$

Take $c = (e^2 + 12) \log k + e^2 \frac{n-k}{k}$. Then, by Theorem 4.23, we have $\mathbb{P}(\Delta(OUT_k) \geq c) = O(k^{-1})$. Moreover, by Lemma 4.28, we have $\mathbb{E}(|SP_k^{(k)}|) = O(n \log k)$, which gives us

$$\begin{aligned}\mathbb{E}(T_k) &= O((e^2 + 12)n \log^2 k + e^2(n-k)n \log k/k) + O(n) + O(n^2/k) \\ &= O(n \log^2 n + n^2 \log n/k).\end{aligned}$$

Putting everything together, we bound the expected time of the algorithm from above as

$$\begin{aligned}\mathbb{E}\left(\sum_{k=1}^{n} T_k\right) &= \sum_{k=1}^{\log^2 n - 1} \mathbb{E}(T_k) + \sum_{k=\log^2 n}^{n} \mathbb{E}(T_k) \\ &\leq \sum_{k=1}^{\log^2 n - 1} n^2 + \sum_{k=\log^2 n}^{n} O(n \log^2 n + n^2 \log n/k) = O(n^2 \log^2 n),\end{aligned}$$

as claimed. □

It should be pointed out, that the Hourglass algorithm reduces the number of comparisons relative to the Tree algorithm, but despite the improvement we were unable to obtain lower theoretical bounds for it compared to the Tree algorithm.

4.3.4 Empirical comparison of paths examined

Although we have successfully analyzed the expected-case complexity of the Tree algorithm, we do not yet know how it compares to the Hourglass algorithm, since we were not able to obtain tighter expected-case bounds on the latter. To this end, and to verify the theoretical results in practice, we empirically examine how many relaxations are skipped by the Hourglass and Tree algorithms compared to the Floyd-Warshall algorithm. We performed two experiments: one on random complete digraphs, and one on random digraphs of varying density. We counted the number of relaxations performed by the algorithms. To make the comparison between Floyd-Warshall and its modified versions fairer in the second experiment, we augmented the Floyd-Warshall algorithm with a simple modification that allowed it to skip combinations i, k where $W[i, k] = \infty$,

which reduced the number of relaxations. We denote the number of relaxations after the improvement as R_{FW}. Note that this improvement brings no savings in the case of a complete graph. The results of experiments are presented in the plots relative to the number of relaxations performed by the Floyd-Warshall algorithm; i.e., all numbers of relaxations are divided by R_{FW}, or in the case of complete graphs this is simply divided by n^3.

The experiments were conducted on uniform random digraphs with arc weights uniformly distributed on the interval $[0, 1]$. The digraphs were constructed by first setting the desired vertex count and density. Then, a random Hamiltonian cycle was constructed, ensuring the strong connectivity of the digraph. After the cycle was constructed, the remaining $n(n-2)$ arcs were put into a list and randomly permuted, and then added into the digraph until the desired density was reached.

The first experiment was for the input of random complete digraphs of varying sizes. The results of the first experiment, in which $R_{FW} = n^3$ since the digraphs are complete, are presented in Figure 4.5. To relate the theoretical upper bound of $O(n^2 \lg^2 n)$ of the Tree algorithm and the experimental results, we added also the plot of the function $60\frac{n^2 \lg^2 n}{n^3}$. We chose the constant 60 so that the plots of the Tree algorithm and the added function start at the same initial point, namely at 2^8 vertices.

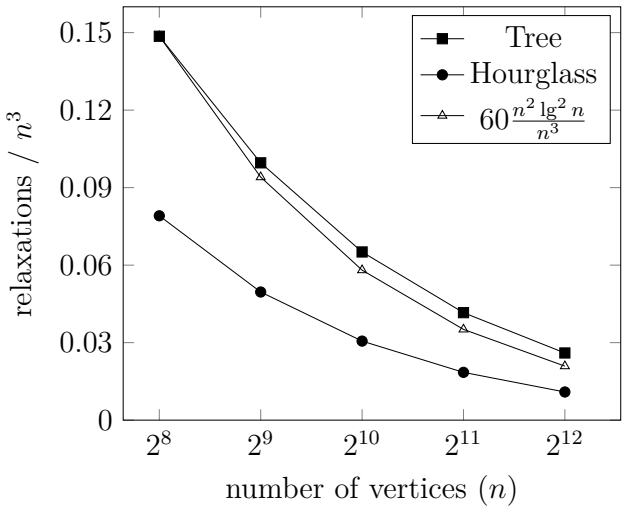


Figure 4.5: Complete digraphs of various sizes with the number of relaxations of algorithms divided by n^3.

The results of the second experiment for $n = 1024$ vertices and sizes of the arc set varying between $n^2/10$ and $8n^2/10$ are shown in Figure 4.6.

In Figure 4.5 we see a significant reduction of relaxations which also implies the decrease of running time of the Tree and Hourglass algorithms. From the plot we can also see that the experimental results indicate that the theoretical upper bound of the Tree algorithm is asymptotically tight. The experiments on digraphs of varying density (see Figure 4.6) also show a reduction in relaxations as the digraphs become sparser.

4.4 Empirical evaluation

In Section 4.3.4 we have used empirical comparisons to measure the number of path comparisons examined by the algorithms derived from Floyd-Warshall. In this section

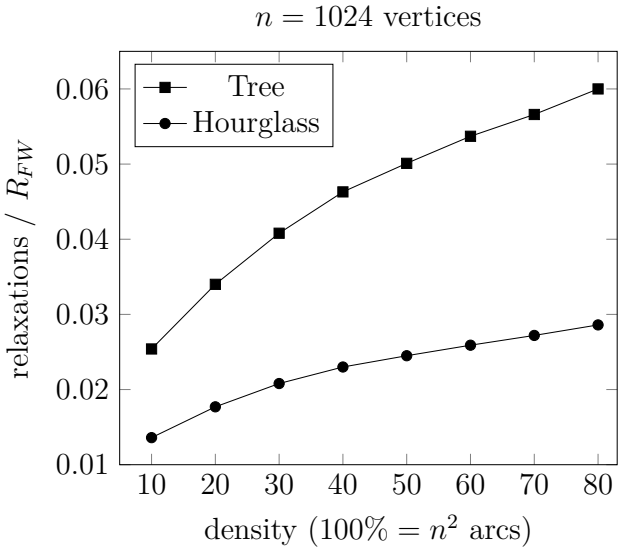


Figure 4.6: Digraphs with $n = 1024$ vertices and various arc densities with the number of relaxations of algorithms divided by R_{FW}.

we are interested in comparing the actual execution times of many different shortest path algorithms, to better determine how they might perform in practice. The results from this section have been previously published in [6, 9].

4.4.1 Graphs

The experiments were conducted on the following two types of random digraphs. Note that these graphs differ slightly from those in Subsection 4.3.4.

Uniform random digraphs: the arc weights are distributed independently of each other and uniformly at random inside the interval $[0, 1]$. As these digraphs grow denser, they start to favor the expected-case algorithms, since the number of essential edges $m^* = O(n \lg n)$ with high probability in complete digraphs with uniformly distributed random arc weights [19].

Unweighted random digraphs: arc weights are set to 1. These digraphs can be viewed as a type of worst-case for the expected-case algorithms, since $m^* = m$ always holds, i.e., every arc is a shortest path between two vertices. It should be pointed out that breadth-first search (BFS) is extremely efficient in solving these instances given its simplicity and $O(mn)$ running time (when solving APSP). However, since we consider these instances only as a worst-case of a more general shortest path problem, we did not include BFS in the comparisons.

In both cases, the digraphs were constructed by first setting a desired vertex count and density. Then, a random Hamiltonian cycle is constructed, ensuring that the digraph is strongly connected. Arcs are added into the digraph at random until the desired density is reached. Finally, algorithms are executed on the instance, and their running times recorded. We have explored densities ranging from $m = n^{1.1}$ to $m = n^2$, and vertex counts ranging from $n = 512$ to $n = 8192$. Tests on vertex counts up to 4096 (inclusive) were conducted ten times and averaged. For the $n = 8192$ vertex instances each test was conducted only once, due to the long execution times involved.

4.4.2 Algorithms

Priority queues are integral to many shortest path algorithms. Pairing heaps [29] were used in all experiments, since they are known to perform especially well in this capacity in practice [61, 67]. Unlike Fibonacci heaps, which have an $O(1)$ amortized decrease key operation, the amortized complexity of decrease-key for pairing heaps is $O(2^{2\sqrt{\lg\lg n}})$ [65]. We compared the following algorithms:

Dijkstra [22]: solves all-pairs by solving multiple independent single-source problems. Using pairing heaps this algorithm runs in $O(n^2 \lg n + mn2^{2\sqrt{\lg\lg n}})$.

Floyd-Warshall [26, 83]: classic dynamic programming formulation as described in Algorithm 2. Does not use priority queues and has a worst-case bound of $O(n^3)$. The same augmentation from Section 4.3.4 that skips combinations i, k where $W[i, k] = \infty$ has been added to it.

Hidden Path [48]: a type of Dijkstra's algorithm for the all-pairs shortest path problem. Using pairing heaps this algorithm has a worst-case bound of $O(n^2 \lg n + m^* n 2^{2\sqrt{\lg\lg n}})$.

Uniform Path [21]: essentially a more refined variant of the Hidden Path algorithm. Using pairing heaps this algorithm has a worst-case bound of $O(n^2 \lg n + |UP| n 2^{2\sqrt{\lg\lg n}})$, where $|UP|$ is the number of uniform paths, i.e., paths where each proper subpath is a shortest path. Note that $|UP| \leq m^* n$ for any graph if the shortest paths are unique, and $|UP| = O(n^2)$ with high probability in complete digraphs with uniformly distributed random arc weights [63]. It should be noted that even if the shortest paths are not unique, the distance metric is easily adjusted to break ties so that the shortest paths become unique for the purposes of the algorithm.

Propagation: the algorithm from Section 4.1 with ψ being Dijkstra's algorithm. We included the *Size reduction* and *Weight bounding* optimizations outlined in Subsection 4.1.5, but not the optimizations mentioned under *Further optimizations*, since we found that after the first two optimizations the running time of ψ did not present a bottleneck. Using pairing heaps this algorithm has a worst-case bound of $O(n^2 \lg n + m^* n 2^{2\sqrt{\lg\lg n}})$.

Tree: the algorithm (with the outlined optimizations) from Section 4.3.1. Does not use priority queues and has a worst-case bound of $O(n^3)$.

Hourglass: the algorithm from Section 4.3.2. Does not use priority queues and has a worst-case bound of $O(n^3)$.

All the code has been written in `C++` and compiled using `g++ -march=native -O3`. We have used the implementation of pairing heaps from the Boost Library, version 1.55, all other algorithms use our own implementation. All tests ran on an Intel® Core™ i7-2600@3.40GHz with 8GB RAM running Windows 7 64-bit.

All results are shown as plots with the y axis representing time in milliseconds in logarithmic scale, and the x axis representing the digraph arc density as $m = n^x$, i.e., $x = \frac{\log m}{\log n}$.

4.4.3 First round of experiments

In the first round of experiments we measured the performance Dijkstra, Propagation, Uniform Paths, Hidden Paths, and Floyd-Warshall algorithms. For the final tally in the second round presented in Subsection 4.4.4, we took the fastest ones among these.

The results for uniform random digraphs are shown in Figure 4.7. Propagation performs remarkably well in these tests, significantly outperforming other algorithms

and only being tied with Dijkstra on the 4096 and 8192 vertex digraphs in the $1.1 - 1.5$ density range. This is expected as the running time of Propagation depends on m^* instead of m, and the ratio $\frac{m}{m^*}$ in the uniform random digraphs increases as the graphs grow denser, so it is expected that Dijkstra would be relatively slower the denser the graph is.

The results for unweighted random digraphs are shown in Figure 4.8, which show the opposite picture. Whereas in the previous test Dijkstra and Floyd-Warshall were dominated by the other algorithms, they are now the dominant ones. Dijkstra is significantly faster than all of the other algorithms, except in the very dense cases where it is outperformed by Floyd-Warshall. These results are expected, since the other algorithms gain their speed by ignoring arcs that are not necessary for the computation. Since $m = m^*$ in these graphs, none of the arcs are ignored. The n^2 case for unweighted digraphs is somewhat pathological, as the instance is already solved since every vertex has a unit weight arc to every other vertex, which can be seen to cause a consistent dip in the running time in the case of Dijkstra.

4.4.4 Second round of experiments

In the second round of experiments we analyzed the dominant algorithms from the first round and our two algorithms from Section 4.3. Specifically, the algorithms compared in this round are: Dijkstra, Propagation, Floyd-Warshall, Tree, Hourglass. The algorithms Hidden Paths and Uniform Paths were omitted in this round, because they were already shown to underperform in the first round relative to Propagation in the case of uniformly random digraphs, and Dijkstra in the case of unweighted digraphs.

The results for uniform random digraphs are shown in Figure 4.9. The tests show that Propagation and Tree together outperform the other algorithms on all densities. As the size of graphs increases, Hourglass starts catching up to Tree, but the constant factors still prove to be too large for it to benefit from its more clever exploration strategy. It is interesting to see the Tree and Hourglass algorithms being so efficient on the graphs of varying density, outperforming even Dijkstra, which further confirms that their expected-case time is indeed much lower than their $O(n^3)$ worst-case time.

The results for unweighted random digraphs are shown in Figure 4.10. What is interesting is that the Tree and Hourglass algorithms remain competitive with Dijkstra, and even outperforming it on the smaller graphs in some instances. This is in stark contrast with the Propagation algorithm, which does not perform as well given that $m = m^*$ in these graphs.

4.5 All-pairs bottleneck paths

The all-pairs bottleneck paths problem [43,68] (APBP) is closely related to the all-pairs shortest path problem. The difference is only in the definition of the weight of a path $P = v_{P,0}, v_{P,1}, \ldots, v_{P,r}$, which we now call the width and define as

$$w(P) = \min_{i=0}^{r-1} w(v_{P,i}, v_{P,i+1}).$$

Analogous to how we denoted the weight of a shortest path from vertex u to vertex v as $D_G(u, v)$, we denote the width of a widest path from vertex u to vertex v as $\tilde{D}_G(u, v)$. A solution to this problem is readily available by simply modifying the relaxation equation

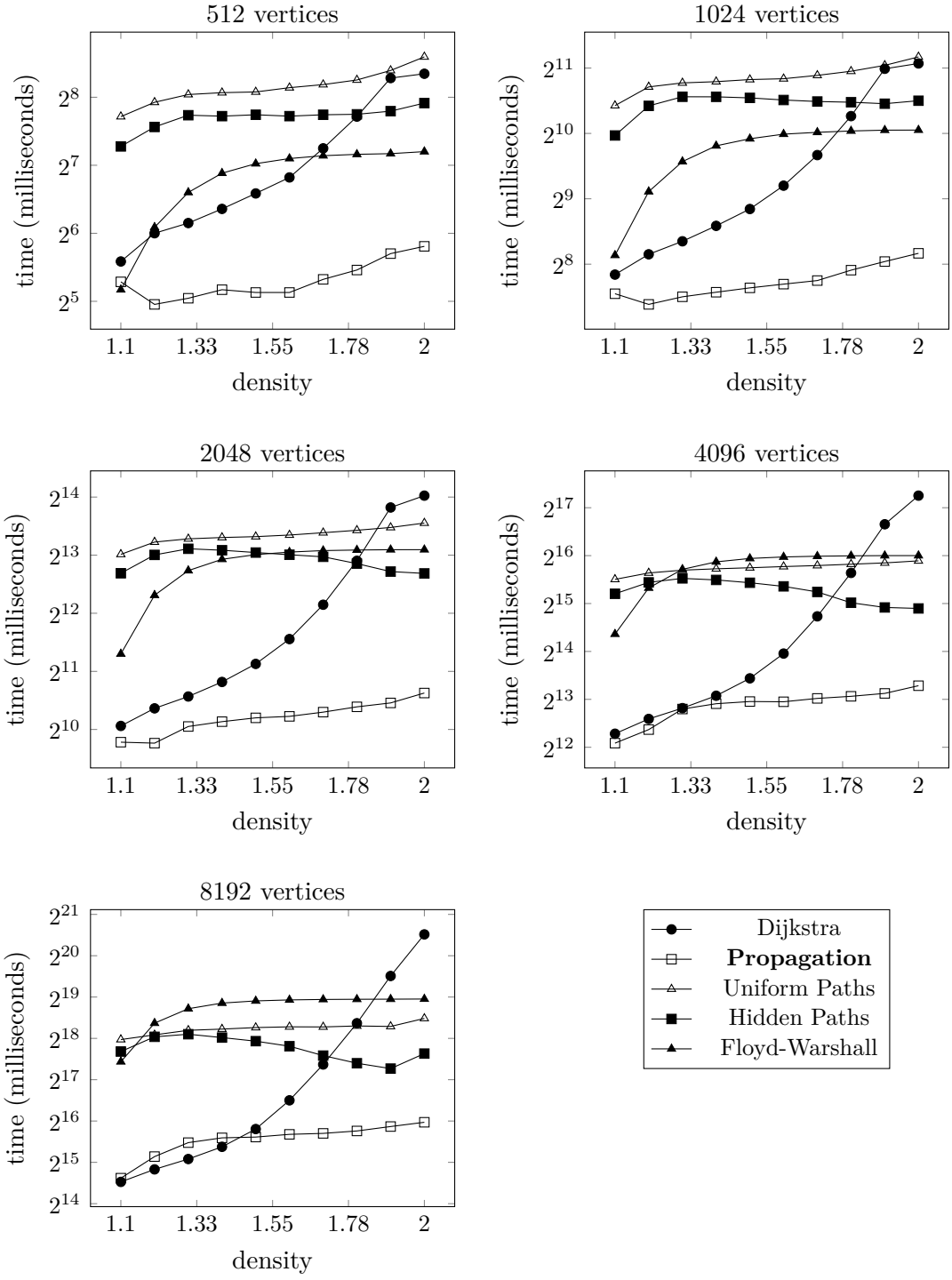


Figure 4.7: Uniform digraphs, first round

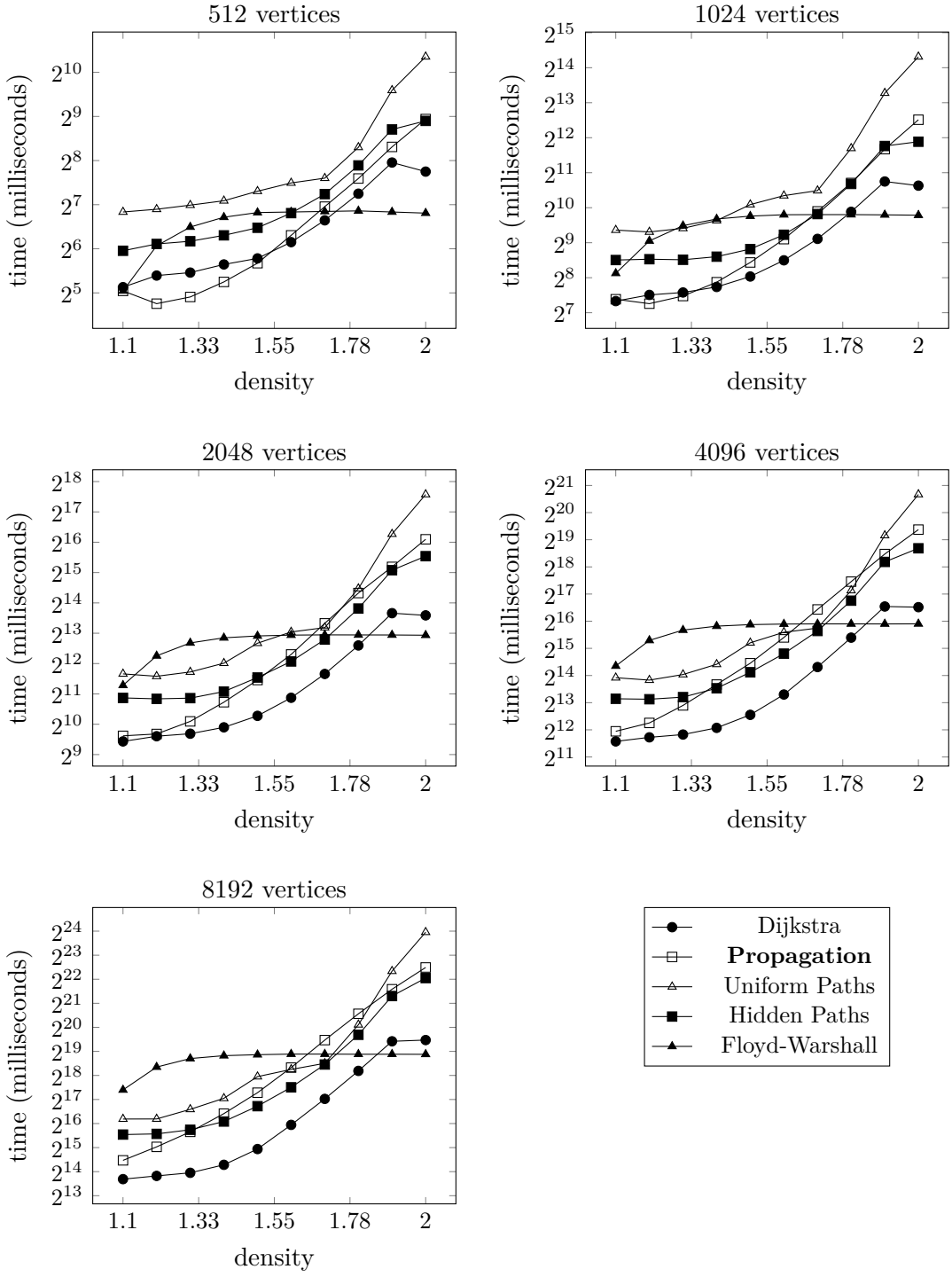


Figure 4.8: Unweighted digraphs, first round

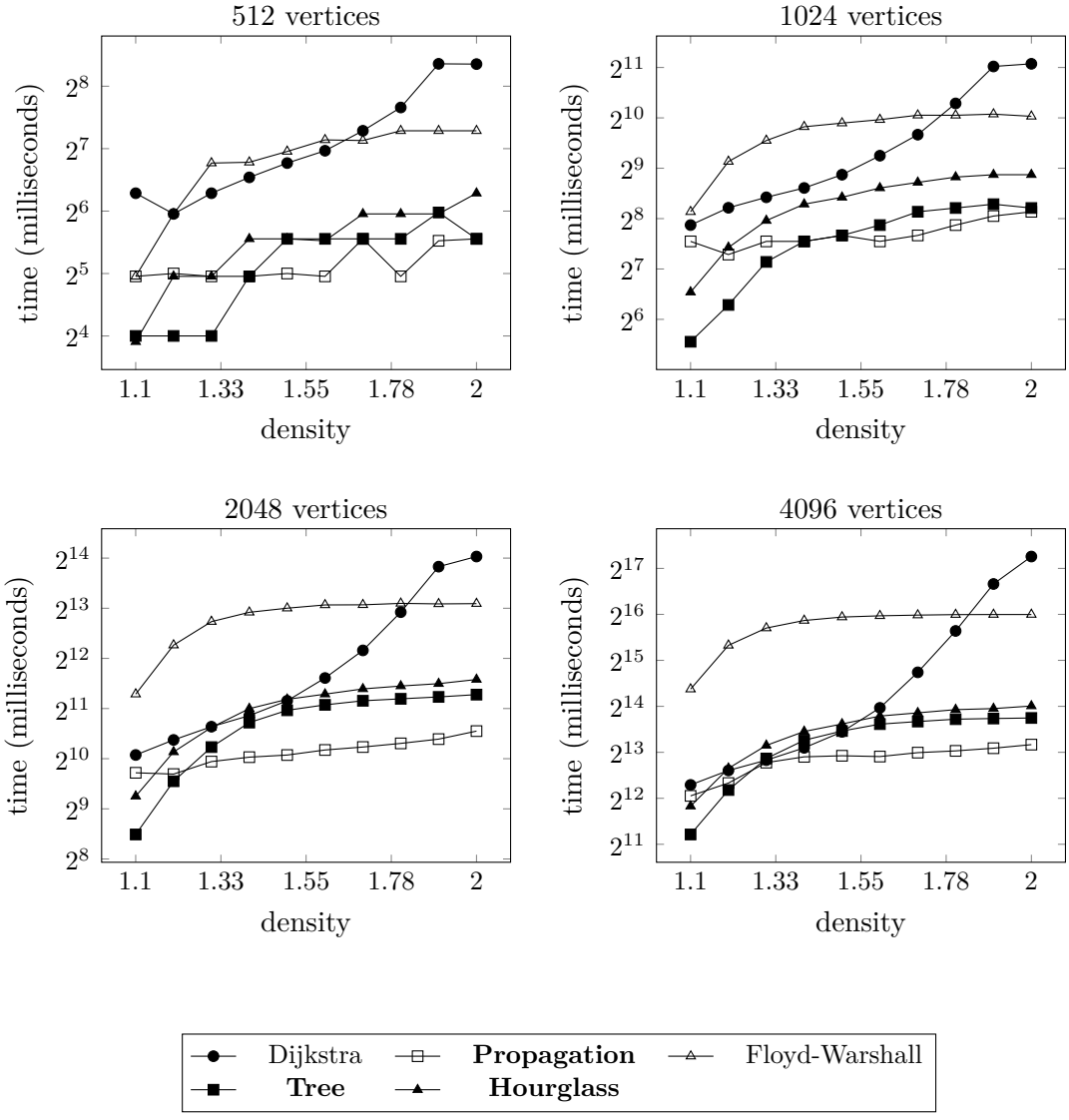


Figure 4.9: Uniform digraphs, second round

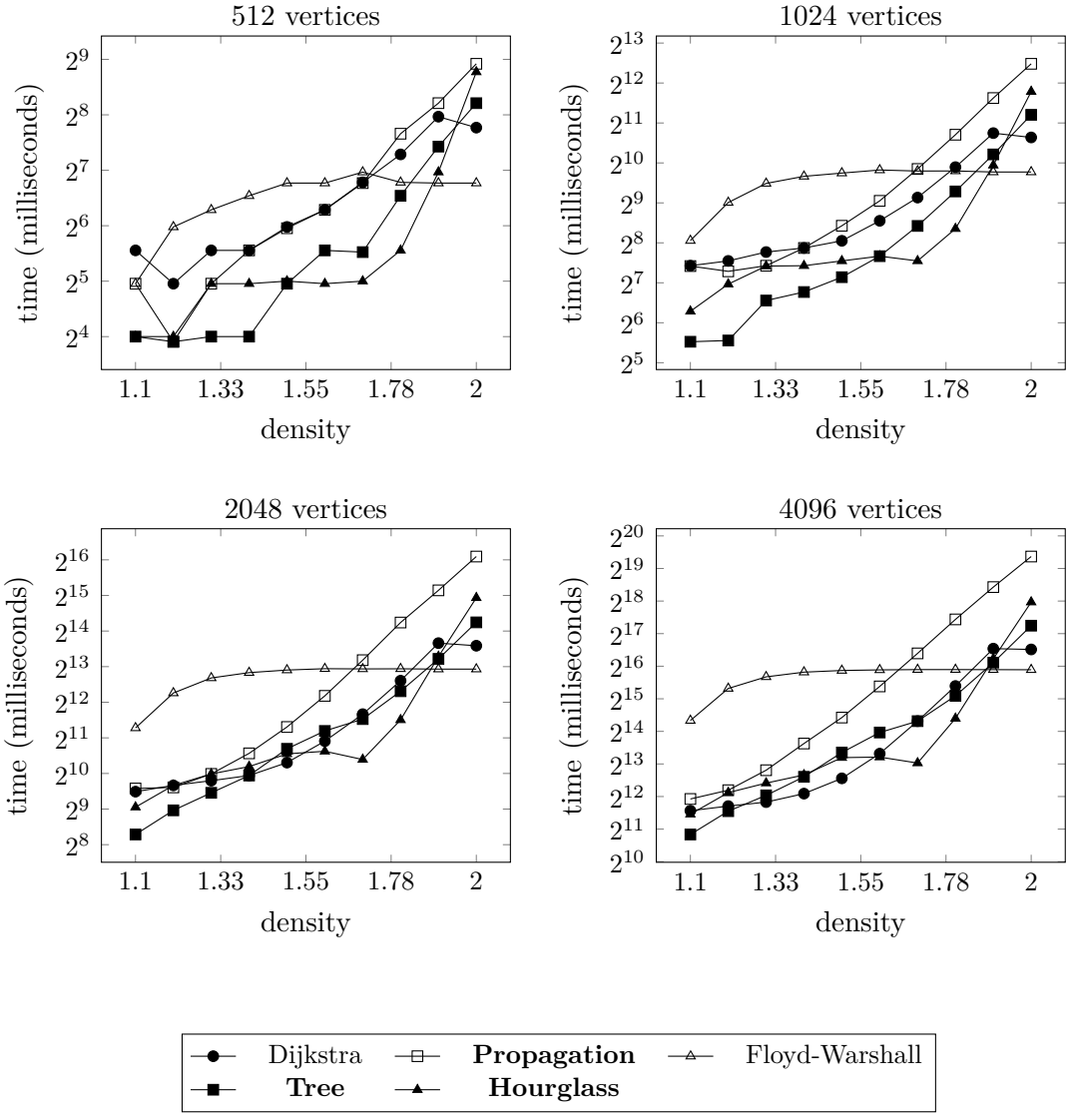


Figure 4.10: Unweighted digraphs, second round

of shortest path algorithms to use maximum instead of minimum and minimum instead of addition, e.g., lines $5-6$ in Algorithm 2. Modifying Dijkstra's algorithm in this way leads to a solution that runs in time $O(mn + n^2 \lg n)$ using Fibonacci heaps [30]. A more efficient folklore modification of Dijkstra's algorithm is known to reduce this time down to $O(mn)$. This folklore modification sorts the arcs by their weights beforehand in $O(m \lg n)$ time and then maps weights to integers in $\{1, \ldots, m\}$. This permits it to use a bucket instead of a heap and each SSSP computation takes $O(m)$ time in total. It should be pointed out that in the case of undirected graphs, we can solve APBP on a maximum spanning tree of G instead of on G itself [43], and still obtain the correct result. This can be a significant speed-up, since $m = n - 1$ for any maximum spanning tree.

In this section, we describe an algorithm that is more efficient, with an asymptotic running time of $O(m^* n + m \lg n)$. This algorithm will also allow us to state a connection between APBP and the dynamic transitive closure problem.

Given a strongly connected weighted digraph $G = (V, A)$ with a weight function w, the algorithm works by incrementally building a weighted digraph $G^* = (V, A^*)$ where $A^* \subset A$ is the set of essential arcs, i.e., arcs that are part of some widest path. It accomplishes this by inserting arcs into an initially disconnected set of vertices. The first step is to sort the set of arcs A with respect to their weights. This can be done with any off-the-shelf sorting algorithm in $O(m \lg n)$ time.

Now we consider each arc in this sorted list from heaviest to lightest. Given an arc (u, v), check if v is reachable from u in G^*. If it is, ignore it and move to the next arc, and if it is not, add (u, v) to G^*, and for every pair of vertices (p, q) such that q becomes reachable from p, we have $\tilde{D}_G(p, q) = w(u, v)$. The algorithm finishes when we have considered every arc.

We summarize the algorithm in pseudocode as Algorithm 8.

Algorithm 8 APBP Algorithm

```
1: procedure APBP(V, A)
2:     Initialize D, an n × n matrix, as D[i, j] := ∞.
3:     A* := ∅
4:     for all (u, v) ∈ A from heaviest to lightest do
5:         if D[u, v] = ∞ then
6:             A* := A* ∪ {(u, v)}
7:             D[u, v] := w(u, v)
8:             for all (p, q) where D[p, q] = ∞ and q is reachable from p in G* = (V, A*)
   do
9:                 D[p, q] := w(u, v)
       return D
```

Lemma 4.30. *For a weighted digraph $G = (V, A)$ Algorithm 8 correctly computes $\tilde{D}_G(u, v)$ for all $u, v \in V$.*

Proof. Let $a_1, a_2, \ldots, a_m$ be the arcs in sorted order, i.e., $w(a_1) \geq w(a_2) \geq \ldots \geq w(a_m)$. Assume the algorithm is at a stage k, i.e., having examined the first $k - 1$ arcs. For the case $k = 1$, the heaviest arc in the graph clearly forms the widest path between the two vertices it connects. For some $m \geq k > 1$, let $a_k = (u, v)$ and consider first the case that we can reach vertex v from vertex u in the current version of the graph G^*.

That would imply that $\tilde{D}_G(u,v) \geq w(a_{k-1})$, due to the definition of path width, which means a wider (or equal) path as a_k already exists, thus the arc can be safely omitted as it is redundant.

In the case that we cannot yet reach vertex v when starting from vertex u, then (u,v) is the heaviest arc to connect the two vertices, and thus clearly $\tilde{D}_G(u,v) = w(a_k)$. For any additional vertex pairs (p,q) that become reachable after a_k is added into the graph, they clearly contain a_k on any path that connects them. Since all the other arcs in the current graph have a weight greater than or equal to $w(a_k)$, by the definition of path width it holds that $\tilde{D}_G(p,q) = w(a_k)$, which completes the proof. □

The running time of the algorithm depends heavily on how we check which previously unreachable vertex pairs have become reachable (line 8 in Algorithm 8). The following simple approach works when adding some arc (u,v):

1. Gather all vertices x in G^* that can reach u but not v. This takes $O(n)$ time by using the D matrix from Algorithm 8 and checking for infinity.

2. For each vertex x from Step 1, start a breadth-first exploration of G^* from v, visiting only vertices that were previously not reachable from x.

Over the entire course of the algorithm, m^* arcs are added to G^*, so the time for the first step is $O(m^*n)$. The second step is not more expensive than the cost of each vertex performing a full breadth-first exploration of G^* when it is fully built, thus at most $O(m^* + n)$ per vertex, amounting to $O(m^*n)$ in total and yielding an overall cost of $O(m^*n)$.

Combining both times for the arc sorting and reachability checking, we arrive at the bound of $O(m^*n + m \lg n)$. It is worth pointing out that in the case of undirected graphs, G^* corresponds to a maximum spanning tree of G. This is interesting, because it means $m^* = O(n)$, so the running time of the algorithm becomes simply $O(n^2 + m \lg n)$ for undirected graphs. This remains true even if the *representation* is directed, i.e., each arc is simply repeated in both directions with the same weight. In some sense, the algorithm is adaptive to the input graph.

4.5.1 Connection to the dynamic transitive closure problem

We now consider two cases of dynamic transitive closure and point out connections to APBP. In the incremental version we are given a sequence of arc *insertions* on an initially empty graph, and the task is to maintain the ability to efficiently answer reachability queries. In the decremental version, we are given a graph, and a sequence of arc *deletions* and must support the same reachability queries. In both cases, we are able to use these algorithms to solve APBP if we assume that the algorithms also provide which pairs of vertices became reachable (or unreachable) after the operation, by using Algorithm 8. In the case of decremental transitive closure, we have to do some minor modifications to Algorithm 8 and insert the edges from lightest to heaviest, and then checking which have become unreachable. The pairs that become reachable (and unreachable) can usually be provided by algorithms that maintain an explicit representation and answer queries in $O(1)$. Relatively recent advancements in decremental transitive closure have led to an algorithm that has a total running time of $O(mn)$ under m arc deletions [51], while an algorithm achieving that bound for incremental transitive closure has been around for much longer [45]. Both of these algorithms

maintain the transitive closure explicitly, and so can provide the relevant pairs we require. Since transitive closure can be computed in $O(\frac{mn}{\lg n})$ time [13], a decremental algorithm that matches that running time could also lead to an $o(mn)$ *combinatorial* algorithm for APBP. While subcubic algebraic algorithms for APBP based on matrix multiplication exist [81], no $o(mn)$ combinatorial algorithm is known.

Chapter 5

Ant system

We study the parallel variants of the Ant System (AS) algorithm, which is part of the ant colony optimization (ACO) family of metaheuristics. The ACO family of algorithms simulate the behavior of real ants which find paths using pheromone trails. A number of variations on the basic idea exist, such as the Ant System [24], Ant Colony System [23], the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [76], and many others. We focus on the canonical Ant System algorithm, which can be adapted to solve a variety of combinatorial optimization problems such as vehicle routing [11], quadratic assignment [56], subset problems [52], and others. We limit ourselves to the traveling salesman problem (TSP).

The adoption of the graphics processing unit (GPU) as a computing platform in recent years has triggered a wave of papers discussing the parallelization of known algorithms. Recent papers [12, 53, 85] have focused on providing a parallel version of Ant System for the GPU. In this chapter, we show that these algorithms are more general and can be studied in absence of GPU specifics. In line with this observation, we suggest a move towards more well-understood theoretical models such as the PRAM. This greatly facilitates asymptotic analysis and subsequently allows one to see where the algorithms could be improved.

Our main goal is to investigate efficient AS algorithms for variants of the PRAM model of computation and to identify how these might be useful in practice. We break down the AS algorithm into two separate phases: Tour Construction and Pheromone Update. We then show that the existing GPU algorithms for AS can be translated to the PRAM model, which permits us to perform asymptotic analysis. While Tour Construction remains efficient even under PRAM, we identify bottlenecks in the Pheromone Update phase, which are caused by reliance on atomic instructions that are not readily available on most variants of PRAM (or older GPUs). We overcome this with a novel Pheromone Update algorithm that does not require such instructions. Finally, we show that these results are relevant in practice when atomic instructions are not available. We do this by implementing the resulting Pheromone Update algorithm on the GPU and obtain significantly better results in the case of absence of atomic instructions. It is important to note that throughout this chapter we do not consider the quality of the obtained solutions, as we are only interested in the running time while staying true to the original Ant System algorithm.

The chapter is structured as follows. In Section 5.1 we briefly introduce the Ant System algorithm in its sequential form. In Section 5.2 we provide a PRAM implementation of the Ant System algorithm and show how to improve it, and finally we

provide results of empirical tests on the GPU.

5.1 Background

5.1.1 The traveling salesman problem

In the traveling salesman problem, we are given a complete weighted digraph K_n and a positive arc weight function $w\colon A \to \mathbb{R}^+$. The task, then, is to produce a path $P = v_{P,0}, \ldots, v_{P,n-1}$ which minimizes the following weight:

$$w(v_{P,n-1}, v_{P,0}) + w(P).$$

Such a path joined at the end with the first vertex is also known as a Hamiltonian cycle. Observe that even though our formulation requires a complete graph, sparse graphs can be handled by inserting the missing arcs with weight ∞. Undirected graphs can also be handled simply by creating two arcs and then assigning equal weights to both of them with the weight function w.

5.1.2 Ant system for the TSP

We are ready to describe the canonical Ant System for solving the TSP [24] in sequential form. As in the description of the TSP problem, we assume we are given a complete weighted digraph K_n with weight function $w\colon A \to \mathbb{R}^+$. We then define the heuristic matrix η and the pheromone matrix τ, both of dimensions $n \times n$ as follows. The heuristic matrix η does not change throughout the algorithm and represents the quality of an arc (v_i, v_j) as follows:

$$\eta_{i,j} = \begin{cases} \frac{1}{w(v_i,v_j)} & (v_i, v_j) \in A \\ 0 & \text{otherwise.} \end{cases}$$

The pheromone matrix τ initially contains the value 1 everywhere, but changes throughout the execution of the algorithm. Two parameters $\alpha \geq 0$ and $\beta \geq 0$ regulate the importance of pheromone and heuristic information, respectively.

In the canonical algorithm, ants build solutions according to:

$$\mathbb{P}(j|i, \mathcal{S}) = \frac{(\tau_{i,j})^\alpha \cdot (\eta_{i,j})^\beta}{\sum_{k \in N(i,\mathcal{S})} (\tau_{i,k})^\alpha \cdot (\eta_{i,k})^\beta} \ , \tag{5.1}$$

where $\mathbb{P}(j|i, \mathcal{S})$ is the probability of choosing vertex v_j when at vertex v_i and according to the current partial solution $\mathcal{S}$, which is just a sequence of vertices. The feasible neighborhood of the current incomplete solution is a set of vertices denoted by $N(i, \mathcal{S})$. In practice, this selection is implemented via the so-called roulette wheel selection algorithm, which cannot be efficiently parallelized. We instead use the independent roulette selection method, which selects j according to

$$\arg \max_{j \in \{1 \ldots n\}} \mathbb{P}(j|i, \mathcal{S}) R_j \ , \tag{5.2}$$

where R_j for $1 \leq j \leq n$ are random variables distributed independently of each other and uniformly at random inside the interval $(0, 1]$. Although the independent roulette selection works in a qualitatively different way to the canonical selection, studies show

it does not affect the quality of the solutions obtained, while allowing a significantly faster parallel implementation [54].

The algorithm stores the probability from Eq. (5.1) of choosing certain arcs in the *chance* matrix. Since the TSP does not permit returns to previously included vertices (except for the last vertex), those vertices have probability zero of being included. This is typically accomplished by having each ant keep track of a *tabu* list. When considering AS for TSP, the recommended number of ants equals the number of vertices [25], which is also a common choice in the case of parallel variants [12, 85] of AS. From hereon we always assume we have n ants, each starting its solution in a different vertex.

Once solutions are constructed, each solution $\mathcal{S}$ is evaluated to obtain its quality $f(\mathcal{S})$, which in most cases is simply the sum of the inverses of the edge weights. Once all solutions are evaluated, their qualities are used to update the pheromone matrix. First, each cell of the pheromone matrix is multiplied by a constant factor $0 \leq \rho \leq 1$ (evaporation) and then increased according to the solution score (pheromone deposit). Let Z be the set of all solutions produced by the ants, where each ant contributes a single solution. Then, the pheromone update stage is defined by:

$$\tau_{i,j} \leftarrow (1-\rho) \cdot \tau_{i,j} + \sum_{\mathcal{S} \in Z \; s.t. \; (v_i, v_j) \in \mathcal{S}} f(\mathcal{S}). \tag{5.3}$$

The Ant System algorithm, as we have described it, can be formalized as Algorithm 9 and consists of three stages: initialization (Algorithm 10), tour construction (Algorithm 11), and pheromone update (Algorithm 12). The algorithm also uses a number of matrices, which play the following roles: $chance[i,j]$ stores the visit probability value (cf. Eq. (5.1)) for edge (v_i, v_j), $\pi[i,j]$ stores the solutions, specifically the vertex v_j of the ant i, and $tabu[i,j]$ is used to prevent infeasible solutions by storing whether ant i had already visited vertex v_j. The vector $score[i]$ holds the computed score for the solution of ant i. The function $rand()$ is supposed to return a random uniformly distributed real number in the range $(0, 1]$. This is the source of randomness in the algorithm, and allows the algorithm to implement the probabilistic selection according to Eq. (5.2).

Algorithm 9 Sequential Ant System

```
1:  procedure AntSystem(α, β, ρ, totalIterations)
2:      Allocate matrices of size n × n: η, τ, chance, π, tabu
3:      Allocate vector of size n: score
4:      for i := 1 to n do
5:          for j := 1 to n do
6:              τ[i, j] := 1
7:              if i = j then
8:                  η[i, j] := 0
9:              else
10:                 η[i, j] := 1/w(v_i, v_j)
11:     for iter := 1 to totalIterations do
12:         Initialize(α, β, τ, η, score, chance, π, tabu)      ▷ Algorithm 10
13:         TourConstr(η, score, chance, π, tabu)               ▷ Algorithm 11
14:         PheromoneUpdate(τ, ρ, score)                        ▷ Algorithm 12
```

Algorithm 10 Sequential Initialize

```
1:  procedure INITIALIZE(α, β, τ, η, score, chance, π, tabu)
2:      Allocate vector of size n: sum
3:      for i := 1 to n do
4:          sum[i] := 0
5:          for j := 1 to n do
6:              sum[i] := sum[i] + τ[i, j]^α · η[i, j]^β
7:          for j := 1 to n do
8:              chance[i, j] := τ[i, j]^α · η[i, j]^β / sum[i]
9:              tabu[i, j] := 1
10:         π[i, 1] := i                          ▷ First vertex in ant i's solution
11:         score[i] := 0
12:         tabu[i, i] := 0
```

Algorithm 11 Sequential Tour Construction

```
1:  procedure TOURCONSTR(η, score, chance, π, tabu)
2:      for i := 1 to n do
3:          for k := 2 to n do
4:              v := 0
5:              c := −∞
6:              for j := 1 to n do
7:                  t := chance[π[i, k − 1], j] · rand() · tabu[i, j]
8:                  if t ≥ c then
9:                      c := t
10:                     v := j
11:             π[i, k] := v
12:             tabu[i, v] := 0
13:             score[i] := score[i] + η[π[i, k − 1], v]
14:         score[i] := score[i] + η[π[i, n], π[i, 1]]
```

Algorithm 12 Sequential Pheromone Update

```
1:  procedure PHEROMONEUPDATE(τ, ρ, score)
2:      for i := 1 to n do
3:          for j := 1 to n do
4:              τ[i, j] := (1 − ρ) · τ[i, j]
5:      for i := 1 to n do
6:          for k := 2 to n do
7:              τ[π[i, k − 1], π[i, k]] := τ[π[i, k − 1], π[i, k]] + score[i]
8:          τ[π[i, n], π[i, 1]] := τ[π[i, n], π[i, 1]] + score[i]
```

5.2 Parallel Ant system

It is conceptually simpler to consider Ant System as a combination of two algorithms: tour construction and pheromone update (lines 13 and 14 in Algorithm 9, respectively). Attempts at parallel AS, e.g., [3, 20], are usually not very attractive for the PRAM model, since they either employ coarse parallelization or neglect certain parts of parallel AS, typically pheromone update. However, it turns out that parallel AS algorithms for the GPU model [12, 53, 85] translate almost without effort to the PRAM model. It is important to note that the unit of parallelism in the GPU is a thread, while on a PRAM the unit of parallelism is a processor. However since the PRAM is a theoretical model, the actual meaning of processor in this context is abstract. Compared to the GPU, the PRAM model is much simpler. While programs under the PRAM execute in SIMD (Single instruction, multiple data) lock-step fashion, the GPU model of execution is the significantly more ambiguous SIMT (Single instruction, multiple threads), where such lock-step guarantees are lost. Together with details like different levels of memory with different speeds and capacities, writing programs becomes a matter of mixing theoretical and practical considerations. In this thesis we mainly focus on the theoretical aspects of such programs by studying them in the cleaner PRAM model, then transferring them over to the "messier" GPU when doing empirical comparisons.

Due to the decomposition of AS into two algorithms (construction and update), the complexity of AS becomes the worse of the two. We now explore strategies for each algorithm.

5.2.1 Tour construction

The simplest method (cf. [53, 85]) delegates each ant to a separate processor. Now, since each ant stochastically considers each vertex n times (cf. Eq. (5.2)) and has $p = n$ processors, this amounts to step complexity $S(n) = O(n^2)$ and work complexity $W(n) = O(n^3)$.

A remarkable contribution of [12] is their strategy for parallel tour construction. Their tour construction method uses $p = n^2$ processors and associates each ant with n processors. When each ant can make use of n processors, it can effectively generate multiple random numbers in parallel. Then, the maximum operation is used to choose one among n vertices, again in parallel. In total, n maximum operations are performed per ant. When translating this result to the PRAM model, the step complexity of the algorithm depends on the model of computation. In the case of CREW, the maximum can be found with a step complexity $S(n) = O(\lg n)$ and work complexity $W(n) = O(n)$. Since there are n maximum operations per ant, this brings the step complexity to $S(n) = O(n \lg n)$. There are n ants in total, each performing n maximum operations, meaning the work complexity remains $W(n) = O(n^3)$. However, under CRCW, maximum can be performed in $S(n) = O(\lg \lg n)$ step complexity (see, e.g., [73]), thus the step complexity of the algorithm becomes $S(n) = O(n \lg \lg n)$, with the work complexity remaining the same as in the CREW case. Under COMBINING CRCW, this is further reduced to $S(n) = O(n)$ by simply taking the combining operation to be maximum.

It is possible to further reduce the step complexity of the CRCW algorithm to $S(n) = O(n)$ while keeping $W(n) = O(n^3)$ using $p = n^3$ processors and a different method to find the maximum which takes $S(n) = O(1)$ and $W(n) = O(n^2)$: simply

compare all pairs of elements in the array in parallel. However, we will restrict ourselves to $p = n^2$, since the large amount of additional processors required hardly justifies the $\lg \lg n$ gain.

5.2.2 Pheromone update

Once the tour is constructed, the pheromone update must be performed. In [53, 85] the latter is performed sequentially rather than in parallel, i.e., one processor performs the update in $S(n) = O(n^2)$ and $W(n) = O(n^2)$ while others wait. This method is appropriate if we use the first tour construction method, which also has a step complexity of $O(n^2)$, but it becomes a bottleneck if we choose the more parallel tour construction method of [12].

Two pheromone update methods can be found in [12]. The first is straightforward and is based on atomic instructions for addition (cf. the summation in Eq. (5.3)). This method corresponds to the use of COMBINING CRCW with the combining operation set to addition. Thus, we already have one parallel method for pheromone update with $p = n$ and running with a step complexity of $S(n) = O(n)$ and a work complexity of $W(n) = O(n^2)$. If we allow $p = n^2$, then the update can be performed in $S(n) = O(1)$.

The second method of [12] which they refer to as "scatter to gather" is more computationally intensive, but does not use atomic instructions. In this case each cell of the pheromone matrix is represented by a distinct processor, so we require $p = n^2$. Each processor loops through all solutions, summing only the relevant qualities. Solutions are of size $O(n)$ and there are n solutions, meaning each processor performs $S(n) = O(n^2)$ operations. Since there are n^2 processors, this yields a $W(n) = O(n^4)$ work complexity. This method works under both CREW and CRCW models, but in terms of computational complexity, it is uninteresting. Better bounds are achievable by performing pheromone update sequentially, i.e., by a single processor while others wait. Nonetheless, we mention this method since we show how to improve its complexity to $S(n) = O(n)$ and $W(n) = O(n^2)$. We summarize the current best bounds for both tour construction and pheromone update on PRAM in Table 5.1.

5.2.3 Improvements

We now propose a novel method for pheromone update, which improves the currently best known bounds under the CREW and CRCW models. Tour construction in our algorithm is performed as in [12], which translates effortlessly to the PRAM. However, instead of using their "scatter to gather" pheromone update, we develop a new technique which is outlined in the proof of Theorem 5.1.

Theorem 5.1. *Pheromone update (Algorithm 16) can be performed in $S(n) = O(n)$ and $W(n) = O(n^2)$ under a CREW PRAM using $p = n$ processors.*

Proof. Each ant already stores a list of n entries, which correspond to vertices in the order it visited them. In addition to this list, we require that each ant also stores an array *arc* of length n, implicitly storing which arc was used to reach a particular vertex. For example, if the arc (v_i, v_j) was used to visit vertex v_j by ant k, then we would set $arc[k, j] := i$. During pheromone update, processor x updates $\tau[arc[k, x], x]$ for all $k \in [n]$. Without this array, we would have to inspect every element of the solution. There are n solutions, so the step complexity of pheromone update becomes $S(n) = O(n)$ and the work complexity becomes $W(n) = O(n^2)$. □

The pseudocode for the parallel algorithm is shown in Algorithms 13-16. PRAM algorithms use a scalar processor identifier. To improve readability we use a two-dimensional processor identifier $(x, y) \in [n] \times [n]$ when the algorithm calls for n^2 processors and $x \in [n]$ when using n processors. Recall that each ant is using n processors, so the x component of the identifier denotes an ant and the y component denotes an ant's processor. We explicitly denote variables that are local to each processor by prefixing them with a *local* identifier in their initialization. All matrices in the algorithm are of size $n \times n$. The matrices η, τ, *chance*, π, *tabu* and vector *score* were already described in Section 5.1. Additional matrices exist for the parallel algorithm which have the following roles: R holds the results from parallel random number generation and *arc* is used as described in the proof of Theorem 5.1.

Theorem 5.2. *Algorithms 13-16 execute on CREW PRAM.*

Proof. It is easy to see that writes to R (line 3 in Algorithm 15) and τ (lines 3 and 5 in Algorithm 16) preserve write exclusivity since only processor (x, y) writes to $R[x, y]$ and only processor x writes to $\tau[k, x]$ for all $k \in [n]$. We lump together the proof of write exclusivity for *chance* (line 6 in Algorithm 14), *tabu* (lines 7 and 10 in Algorithm 14 and line 7 in Algorithm 15), *score* (line 9 in Algorithm 14 and lines 8 and 11 in Algorithm 15) and *arc* (lines 6 and 10 in Algorithm 15). Observe that in each case the processor's y index is set to one, or is using a one-dimensional index (i.e., $p = n$) which is effectively the same thing. For *score*, which only has one dimension, this avoids conflicts. The rest are matrices and all writes from processor $(x, 1)$ are to cells (x, k) where $k \in [n]$, which does not lead to any conflicts. Note that the proof for the write exclusivity of π (line 8 in Algorithm 14 and line 4 in Algorithm 15) is the same. We require that the parallel implementation of arg max observes the write exclusivity of π. □

Corollary 5.3. *Algorithms 13-16 execute on CRCW PRAM.*

Proof. The CRCW PRAM is a stronger model, so the result follows. In addition, the parallel implementation of arg max no longer requires write exclusivity of π, allowing for a faster implementation (see, e.g., [73]). □

Table 5.1 summarizes complexity bounds derived from previous work as well as new bounds resulting from the improvements presented in this chapter. Since a single iteration of the parallel Ant System algorithm requires both tour construction and pheromone update, the bound becomes the worse of the two.

5.2.4 Empirical comparison

We implemented different pheromone update methods on the GPU. We used Nvidia CUDA, which was also used in recent papers [12, 53, 85] studying the parallel GPU implementation of the Ant System algorithm.

The tests were run on an NVIDIA GeForce GTX 560Ti using stock NVIDIA frequencies. Test instances were taken from TSPLIB [70], which are standard test cases. We included some of the instances that have also been used by [12] to facilitate comparisons their solutions and ours. We compared only the pheromone update stage, since our tour construction step is identical to the one presented in [12], to which we refer readers interested in comparisons between various tour construction methods or comparisons between the parallel and sequential code.

Algorithm 13 Parallel Ant System

```
1: procedure PAntSystem(α, β, ρ, totalIterations)
2:     Allocate matrices of size n × n: R, η, τ, chance, π, tabu, arc
3:     Allocate vector of size n: score
4:     for i := 1 to n do
5:         for j := 1 to n do
6:             τ[i, j] := 1
7:             if i = j then
8:                 η[i, j] := 0
9:             else
10:                η[i, j] := 1/w(v_i, v_j)
11:    for i := 1 to totalIterations do
12:        for x ∈ [n] in parallel do
13:            PInitialize(x, α, β, τ, η, score, chance, π, tabu)
14:        for (x, y) ∈ [n] × [n] in parallel do
15:            PTourConstr(x, y, R, η, score, chance, π, tabu, arc)
16:        for x ∈ [n] in parallel do
17:            PPheromoneUpdate(x, τ, ρ, arc, score)
```

Algorithm 14 Parallel Initialize

```
1: procedure PInitialize(x, α, β, τ, η, score, chance, π, tabu)
2:     local sum := 0
3:     for i := 1 to n do
4:         sum := sum + τ[x, i]^α · η[x, i]^β
5:     for i := 1 to n do
6:         chance[x, i] := τ[x, i]^α · η[x, i]^β / sum
7:         tabu[x, i] := 1
8:     π[x, 1] := x
9:     score[x] := 0
10:    tabu[x, x] := 0
```

Algorithm 15 Parallel Tour Construction

```
1: procedure PTourConstr(x, y, R, η, score, chance, π, tabu, arc)
2:     for k := 2 to n do
3:         R[x, y] := chance[π[x, k − 1], y] · rand() · tabu[x, y]
4:         π[x, k] := arg max_{y∈{1...n}} R[x, y]   ▷ Find max in parallel using n processors
5:         if y = 1 then
6:             arc[x, π[x, k]] := π[x, k − 1]
7:             tabu[x, π[x, k]] := 0
8:             score[x] := score[x] + η[π[x, k − 1], π[x, k]]
9:     if y = 1 then
10:        arc[x, π[x, 1]] := π[x, n]
11:        score[x] := score[x] + η[π[x, n], π[x, 1]]
```

Algorithm 16 Parallel Pheromone Update

```
1: procedure PPheromoneUpdate(x, τ, ρ, arc, score)
2:     for k := 1 to n do
3:         τ[k, x] := (1 − ρ) · τ[k, x]
4:     for k := 1 to n do
5:         τ[arc[k, x], x] := τ[arc[k, x], x] + score[k]
```

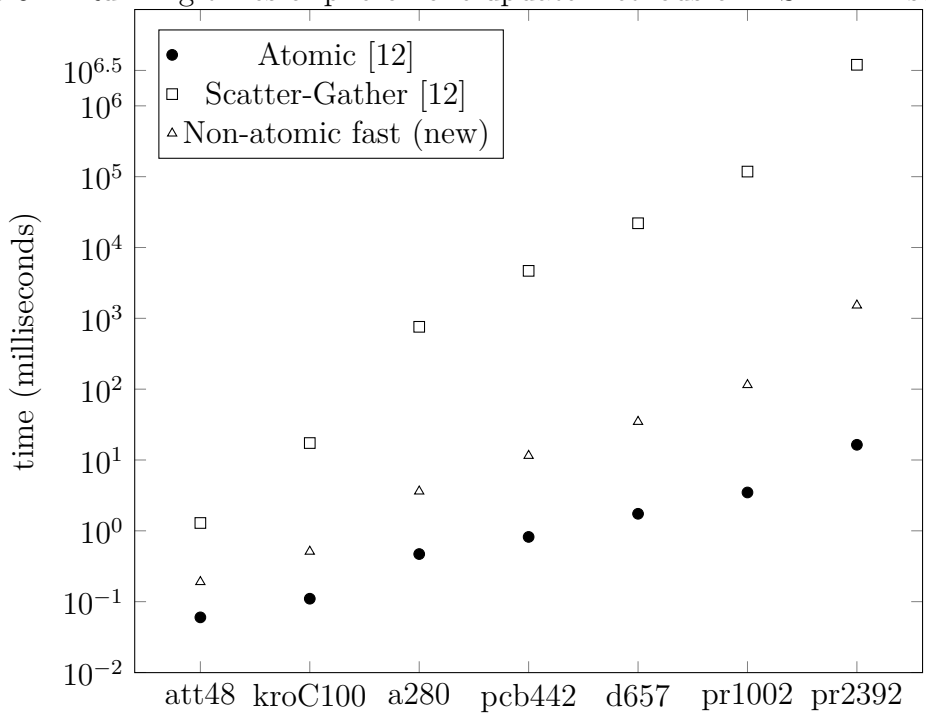
Table 5.1: Previous and new bounds for the parallel Ant System comprised of two sub-algorithms: tour construction and pheromone (PH) update. The COMBINING CRCW model is denoted by CMB. CRCW, the step and work complexities as $S(n)$ and $W(n)$, respectively, and p denotes the number of processors.

Previous work				
		CREW	**CRCW**	**CMB. CRCW**
Tour [12]	**S(n)**	$O(n \lg n)$	$O(n \lg \lg n)$	$O(n)$
	W(n)	$O(n^3)$	$O(n^3)$	$O(n^3)$
	p	n^2	n^2	n^2
PH [12]	**S(n)**	$O(n^2)$	$O(n^2)$	$O(1)$
	W(n)	$O(n^4)$	$O(n^4)$	$O(n^2)$
	p	n^2	n^2	n^2
Total	**S(n)**	$O(n^2)$	$O(n^2)$	$O(n)$
	W(n)	$O(n^3)$	$O(n^3)$	$O(n^3)$
	p	n^2	n^2	n^2
This chapter				
		CREW	**CRCW**	
PH	**S(n)**	$O(n)$	$O(n)$	
	W(n)	$O(n^2)$	$O(n^2)$	
	p	n	n	
Total	**S(n)**	$O(n \lg n)$	$O(n \lg \lg n)$	
	W(n)	$O(n^3)$	$O(n^3)$	
	p	n^2	n^2	

Table 5.2: Running time (milliseconds) of pheromone update methods on TSPLIB instances.

	Method		
Instance	**Atomic** [12]	**Scatter-Gather** [12]	**Non-atomic fast (new)**
att48	0.06	1.29	0.19
kroC100	0.11	17.35	0.51
a280	0.47	759.14	3.61
pcb442	0.82	4681	11.5
d657	1.74	$22 \cdot 10^3$	34.7
pr1002	3.48	$118 \cdot 10^3$	114.8
pr2392	16.39	$3800 \cdot 10^3$	1525.4

Figure 5.1: Running times of pheromone update methods on TSPLIB instances.



We tested three methods of pheromone update: atomic, scatter-gather, and non-atomic fast. The atomic method updates the pheromone matrix using atomic instructions for addition. The scatter-gather method is the non-atomic method proposed by [12]. Finally, the non-atomic fast method is the one suggested in this chapter. We also remark that, in our case, the atomic update method made full use of $p = n^2$ threads, since we found its performance to be significantly better compared to $p = n$ threads as used in [12]. The results are shown in Table 5.2 and plotted in Figure 5.1.

It is reassuring to see that the theoretical improvements also translate into practice. Although atomic instructions are typically slow, the atomic variant is significantly faster in these experiments. This is because the slow execution of atomic instructions only comes into play when multiple threads actually attempt simultaneous writes to the same location. In our scenario this rarely happens, because every ant is unlikely to have chosen the same edges in its solution. While the atomic variant is significantly faster, older GPUs do not have access to the appropriate atomic instructions. Thus, these results can be practically relevant for GPU implementations if code is expected to work on all GPUs.

Chapter 6

Conclusion

After an extensive study of various aspects related to the topic of this dissertation, it is time to draw some concluding remarks. In this work, we have explored the fundamental concepts, theoretical foundations, and practical evaluations of dynamic programming, shortest path, and ant system algorithms. We have presented novel algorithms, which have shown significant improvements over existing methods in terms of efficiency. We have thoroughly evaluated these algorithms in the context of theoretical and practical performance in different scenarios, which have helped us understand their strengths and limitations and has highlighted possible avenues for future research. In this concluding chapter, we summarize the main findings of this work, discuss their significance, and outline directions for future work.

6.1 Dynamic programming

In Chapter 3 we studied a simple minimum computation arising in Dynamic Programming problems described by Eq. (3.1), and have shown how to compute it in $O(Bn \lg(\frac{n}{B}))$ time where B can be upper bounded by the number of inflection points of g.

It should be noted that, since the algorithm does not work with functions explicitly, but rather with sequences, we can also obtain a speedup by working on on the vector of real values X (cf. Section 3.1), in an analogous way. For example, we could traverse X and find blocks, then decide to use either X or g, whichever has fewer blocks. In many dynamic programming applications [62, 71, 77], the minimum computation that we study is used as a subroutine. In these cases, X changes between calls to the subroutine, whereas g remains static. Consequently, it makes sense to analyze g, especially since it is a function. However, working on X would have its merits if one could show that, for a given application, the number of blocks B in X can be bounded by some value.

6.2 Shortest paths

6.2.1 Propagation

In Section 4.1, we showed that an algorithm with a similar time bound to the Hidden Paths Algorithm can be obtained. Unlike the Hidden Paths Algorithm, the resulting

method is general in that it works for any SSSP algorithm, effectively providing a speed-up for arbitrary SSSP algorithms. The proposed method, given an SSSP algorithm ψ, has an asymptotic worst-case running time of $O(m \lg n + nT_\psi(m^* + n, n + 1))$ and space complexity of $O(S_\psi(m^* + n, n + 1) + n^2)$. For the case of ψ being Dijkstra's algorithm, this is asymptotically equivalent to the Hidden Paths Algorithm. However, since the algorithm ψ is arbitrary, we show in Section 4.1.3 that the combination of our method, Johnson's reweighting technique [47], and topological sorting gives an $O(m^*n + m \lg n)$ APSP algorithm for directed acyclic graphs with arbitrary arc weights.

However, we should mention that in recent years, asymptotically efficient algorithms for APSP have been formulated in the so-called component hierarchy framework. These algorithms can be seen as computing either SSSP or APSP. Our algorithm is only capable of speeding up SSSP hierarchy algorithms, such as Thorup's [79], but not those which reuse the hierarchy, such as Pettie's [64], Pettie-Ramachandran [66], or Hagerup's [36] since our SSSP reduction requires modifications to the graph G', which we construct at each iteration. These modifications would require the hierarchy to be recomputed, making the algorithms prohibitively slow. This raises the following question: is there a way to avoid recomputing the hierarchy at each step, while keeping the number of arcs in the hierarchy $O(m^*)$?

If there exists an $o(mn)$ algorithm for the general SSSP problem with arbitrary arc weights, then by using Johnson's reweighting technique, our algorithm might become an attractive solution for that case. For the general case, no such algorithms are known, but for certain types of graphs, there exist algorithms with an $o(mn)$ asymptotic time bound [33, 35].

Furthermore, we can generalize the approach used on DAGs. Namely, in Algorithm 1 we can use an SSSP algorithm ψ that works on a specialized graph G, as long our constructed graph G' has these properties. Therefore, our algorithm can be applied to undirected graphs, graphs with integer-weight arcs, etc., but it cannot be applied, for example, to planar graphs, since G' is not necessarily planar.

We have also shown a connection between the sorted all-pairs shortest path (SAPSP) problem and the single-source shortest path problem. If a meaningful lower bound can be proven for SAPSP, then this would imply a non-trivial lower bound for SSSP. Alternatively, if SAPSP can be solved in $O(mn)$ time, then this implies a Dijkstra-like algorithm for APSP, which visits vertices in increasing distance from the source.

Additionally, we have studied the practical implementation of five algorithms by comparing their execution times. The results of the test show that our algorithm outperforms other algorithms for graphs with uniformly distributed weights, and could thus be an attractive option for these types of graphs in practice.

6.2.2 Speeding up the Floyd-Warshall algorithm

In Section 4.3 we have looked at a practical algorithm for solving the all-pairs shortest path problem. It is typical of the more practically-minded APSP algorithms to rely on expected-case properties of graphs, and most of them are modifications of Dijkstra's algorithm. However, the Floyd-Warshall algorithm is known to perform well in practice when the graphs are dense. To this end, we have suggested the Tree and Hourglass algorithms: modifications of the Floyd-Warshall algorithm that combine it with a tree structure that allows it to avoid checking unnecessary path combinations. Only those path combinations that provably cannot change the values in the shortest path matrix

are omitted. The Tree algorithm is simple to implement, uses no fancy data structures and in empirical tests is faster than the Floyd-Warshall algorithm for random complete graphs on 512–4096 vertices by factors ranging from 3–5 as can be seen on Figure 4.9. When we inspect the number of path combinations examined in Figure 4.5 however, the Tree modification reduces the number by a staggering factor of 10–38.

Motivated by these results, we have gone on to prove that the Tree algorithm has an expected-case running time of $O(n^2 \log^2 n)$ for complete digraphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0, 1]$. Since the Tree algorithm allows negative arc weights, it would be interesting to analyze its expected-case running time with respect to a model that permits such arcs, for example, the vertex potential model [15, 17]. The Hourglass algorithm even further decreases the number of comparisons performed, and therefore also improves the running time of the algorithm. Obviously, the expected-case time complexity of the Hourglass algorithm is at most $O(n^2 \log^2 n)$ since it is never worse than the Tree algorithm, but it remains an open problem whether the Hourglass algorithm has an $o(n^2 \log^2 n)$ expected-case time complexity in the uniform model.

To compare practical performance, we have devised empirical tests in Section 4.4 using actual implementations. Since, as mentioned, the algorithms studied typically rely on expected-case properties of graphs, we looked at both uniform random graphs and unweighted random graphs of varying density. The latter present a hard case for many of the algorithms and can highlight their worst-case performance, whereas the former are much more agreeable to the algorithms' assumptions. For the choice of algorithms we have included those known from past work, as well as the novel Hourglass and Tree algorithms. As it turns out, the new algorithms have proven to be quite efficient in the empirical tests that we have performed. The simpler Tree algorithm has ranked especially well alongside the Propagation algorithm, while at the same time it was more resilient when it came to worst-case inputs.

6.2.3 Bottleneck paths

We have also briefly considered the case of all-pairs bottleneck paths, and in Section 4.5 we proposed a simple algorithm, the asymptotic running time of which can be parametrized with m^*. Additionally, we have shown ties to the dynamic transitive closure problem, which might lead to faster algorithms for all-pairs bottleneck paths if faster algorithms for the dynamic transitive closure problem can be found.

6.3 Ant system

In Chapter 5 we have shown that recent parallel variants of the Ant System algorithm for the GPU systems can be easily modeled by the more general PRAM model. This makes them both simpler to understand and to analyze. The facilitation in theoretical analysis allowed us to determine which parts of the algorithm needed improvement. It turned out that in two out of three variants of PRAM models studied, the parallel Ant System algorithm was dominated by pheromone update. We proposed a new pheromone update method that improves the asymptotic bound of the parallel Ant System algorithm to such an extent that the entire algorithm becomes dominated by the tour construction phase. Empirical tests have also confirmed that these improvements

translate back to improved performance on the GPU when atomic instructions are unavailable.

Future research directs us to study the possibility of application of the proposed pheromone update method to other algorithms in the ACO family. Moreover, optimization problems other than the TSP could be parallelized in a similar fashion. The algorithms could be studied under various other parallel computation models. Last but not least, we are also interested in other algorithms that could be more efficiently parallelized if they are split into two phases or more phases.

Bibliography

[1] L. Addario-Berry, N. Broutin, and G. Lugosi. The longest minimum-weight path in a complete graph. *Comb. Probab. Comput.*, 19(1):1–19, 2010.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., USA, 1993.

[3] B. Bullnheimer, G. Kotsis and C. Strauss. Parallelization strategies for the ant system. *Applied Optimization*, 24:87–100, 1998.

[4] P. A. Bloniarz. A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$. *SIAM J. Comput.*, 12(3):588–600, 1983.

[5] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[6] A. Brodnik, M. Grgurovič, and R. Požar. Modifications of the Floyd-Warshall algorithm with nearly quadratic expected-time. *Ars Math. Contemp.*, 22(1):1–22, 2022.

[7] A. Brodnik and M. Grgurovič. Practical algorithms for the all-pairs shortest path problem. In A. Adamatzky, editor, *Shortest Path Solvers. From Software to Wetware*, pages 163–180. Springer International Publishing, Cham, 2018.

[8] A. Brodnik and M. Grgurovič. Speeding up shortest path algorithms. In K. Chao, T. Hsu, and D. Lee, editors, *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, volume 7676 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2012.

[9] A. Brodnik and M. Grgurovič. Solving all-pairs shortest path by single-source computations: Theory and practice. *Discrete Applied Mathematics*, 231(Supplement C):119 – 130, 2017. Algorithmic Graph Theory on the Adriatic Coast.

[10] A. Brodnik and M. Grgurovič. Parallelization of ant system for GPU under the PRAM model. *Comput. Informatics*, 37(1):229–243, 2018.

[11] B. Bullnheimer, R. F. Hartl, and C. Strauss. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89:319–328, 1997.

[12] J. M. Cecilia, J. M. Garcia, A. Nisbet, M. Amos, and M. Ujaldon. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42 – 51, 2013.

[13] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\lg n)$ time. *Algorithmica*, 50:236–243, 2008.

[14] T. M. Chan and R. R. Williams. Deterministic APSP, orthogonal vectors, and more: Quickly derandomizing Razborov-Smolensky. *ACM Trans. Algorithms*, 17(1), 2021.

[15] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Programming*, 73(2, Ser. A):129–174, 1996.

[16] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[17] C. Cooper, A. Frieze, K. Mehlhorn, and V. Priebe. Average-case complexity of shortest-paths problems in the vertex-potential model. *Random Structures Algorithms*, 16(1):33–46, 2000.

[18] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[19] R. Davis and A. Prieditis. The expected length of a shortest path. *Information Processing Letters*, 46(3):135 – 141, 1993.

[20] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné. Parallel implementation of an ant colony optimization metaheuristic with OpenMP. *International Conference of Parallel Architectures and Complication Techniques, Proceedings of the third European workshop on OpenMP*, pages 8–12, 2001.

[21] C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms*, 2(4):578–601, 2006.

[22] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[23] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[24] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[25] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

[26] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[27] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, 1972.

[28] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118, New York, NY, USA, 1978. Association for Computing Machinery.

[29] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1–4):111–129, 1986.

[30] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[31] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 1–7, New York, NY, USA, 1990. Association for Computing Machinery.

[32] A. M. Frieze and G. R. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Appl. Math.*, 10(1):57–77, 1985.

[33] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[34] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1):107 – 118, 1989.

[35] A. V. Goldberg. Scaling algorithms for the shortest paths problem. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 222–231, Philadelphia, PA, USA, November 1993. Society for Industrial and Applied Mathematics.

[36] T. Hagerup. Improved shortest paths on the word RAM. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ICALP '00, pages 61–72, London, UK, July 2000. Springer-Verlag.

[37] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990.

[38] S.-C. Han, F. Franchetti, and M. Püschel. Program generation for the all-pairs shortest path problem. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 222–232, New York, NY, USA, 2006. ACM.

[39] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[40] R. Hassin and E. Zemel. On shortest paths in graphs with random weights. *Math. Oper. Res.*, 10(4):557–564, 1985.

[41] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 137–143, Washington, DC, USA, 1985. IEEE Computer Society.

[42] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–322, 1961.

[43] T. C. Hu. Letter to the editor—the maximum capacity route problem. *Operations Research*, 9(6):898–900, 1961.

[44] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Higher Education, 1st edition, 1992.

[45] G. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[46] S. Janson. One, two and three times $\log n/n$ for paths in a complete graph with random weights. *Combinatorics, Probability and Computing*, 8(4):347–361, 1999.

[47] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[48] D. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[49] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, editors, *Complexity of Computer Computations. The IBM Research Symposia Series.*, pages 85–103, Boston, MA, 1972. Springer US.

[50] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[51] J. Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3), 2013.

[52] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problems. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1459–1464, 1999.

[53] J. Li, X. Hu, Z. Pang, and K. Qian. A parallel ant colony optimization algorithm based on fine-grained model with GPU-acceleration. *Internation Journal of Innovative Computing, Information, and Control*, 5(11(A)):3707 – 3716, 2009.

[54] H. Lloyd and M. Amos. Analysis of independent roulette selection in parallel ant colony optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 19–26, New York, NY, USA, 2017. Association for Computing Machinery.

[55] M. Luby and P. Ragde. A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica*, 4:551–567, 1989.

[56] V. Maniezzo and A. Colorni. The ant system applied to the quadratic assignment problem. *IEEE Trans. on Knowl. and Data Eng.*, 11(5):769–778, 1999.

[57] J. J. McAuley and T. S. Caetano. An expected-case sub-cubic solution to the all-pairs shortest path problem in R. *CoRR*, abs/0912.0975, 2009.

[58] C. C. McGeoch. All-pairs shortest paths and the essential subgraph. *Algorithmica*, 13:426–441, 1995.

[59] K. Mehlhorn and V. Priebe. On the all-pairs shortest-path algorithm of Moffat and Takaoka. *Random Structures Algorithms*, 10(1-2):205–220, 1997.

[60] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. *SIAM J. Comput.*, 16(6):1023–1031, 1987.

[61] B. M. E. Moret and H. D. Shapiro. *An empirical assessment of algorithms for constructing a minimum spanning tree*, pages 99–117. DIMACS Monographs. 15. AMS Press, 1994.

[62] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[63] Y. Peres, D. Sotnikov, B. Sudakov, and U. Zwick. All-pairs shortest paths in $o(n^2)$ time with high probability. *J. ACM*, 60(4), 2013.

[64] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theor. Comput. Sci.*, 312(1):47–74, January 2004.

[65] S. Pettie. Towards a final analysis of pairing heaps. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 174–183, Washington, DC, USA, Oct 2005. IEEE Computer Society.

[66] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34(6):1398–1431, 2005.

[67] S. Pettie, V. Ramachandran, and S. Sridhar. Experimental evaluation of a new shortest path algorithm. In D. M. Mount and C. Stein, editors, *Algorithm Engineering and Experiments*, ALENEX '02, page 126–142, Berlin, Heidelberg, 2002. Springer-Verlag.

[68] M. Pollack. Letter to the editor – the maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.

[69] M. Raab and A. Steger. "Balls into bins"—a simple and tight analysis. In *Randomization and approximation techniques in computer science (Barcelona, 1998)*, volume 1518 of *Lecture Notes in Comput. Sci.*, pages 159–170. Springer, Berlin, 1998.

[70] G. Reinelt. TSPLIB – a traveling salesman problem library. *INFORMS Journal on Computing*, 3(4):376 – 384, 1991.

[71] D. Sankoff and J. B. Kruskal. *Time warps, string edits, and macromolecules.* Cambridge University Press, Cambridge, England, 2000.

[72] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency.* Algorithms and combinatorics. Springer, 2003.

[73] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88 – 102, 1981.

[74] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. *SIAM J. Comput.*, 2:28–32, 1973.

[75] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, 1984.

[76] T. Stützle and H. H. Hoos. $\mathcal{MAX}$–$\mathcal{MIN}$ Ant system. *Future Gener. Comput. Syst.*, 16(9):889–914, 2000.

[77] M. S. W. T. F. Smith. New stratigraphic correlation techniques. *Journal of Geology*, 88(4):451 – 457, 1980.

[78] T. Takaoka and A. Moffat. An $O(n^2 \log n \log \log n)$ expected time algorithm for the all shortest distance problem. In *Mathematical foundations of computer science, 1980 (Proc. Ninth Sympos., Rydzyna, 1980)*, volume 88 of *Lecture Notes in Comput. Sci.*, pages 643–655. Springer, Berlin-New York, 1980.

[79] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[80] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. A)*, pages 1–66, Cambridge, MA, USA, 1990. MIT Press.

[81] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 585–589, New York, NY, USA, 2007. ACM.

[82] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics*, 8, 2003.

[83] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[84] F. F. Yao. Speed-up in dynamic programming. *SIAM J. on Alg. Discr. Meth.*, 3(4):532 – 540, 1982.

[85] Y.-S. You. Parallel ant system for traveling salesman problem on GPUs. In *Proceedings of GECCO 2009*, pages 1 – 2, 2009.

Povzetek v slovenskem jeziku

Disertacija se nanaša na področje algoritmov in podatkovnih struktur, ki jih uporabljamo pri reševanju problemov kombinatorične optimizacije. Prispevki disertacije so tako teoretični kot tudi praktični. Kombinatorična optimizacija je veja uporabne matematike in teoretičnega računalništva, ki se ukvarja z iskanjem najboljše rešitve določenih diskretnih problemov. Osredotičili se bomo predvsem na probleme kombinatorične optimizacije, ki jih definiramo na grafih. Slednji so ena najpogostejših diskretnih struktur, ki jih uporabljamo pri modeliranju. Potemtakem ni presenetljivo, da poznamo raznorazne probleme, ki so podani ravno na grafih. Čeprav gre za probleme, ki kronološko segajo v sam začetek področja računalništva, je iskanje hitrejših algoritmov in netrivialnih spodnjih mej teh problemov še dandanes aktualno.

Problemi, ki jih rešujejo algoritmi, opisani v disertaciji, so osnovni, klasični problemi iz kombinatorične optimizacije, ki so vsesplošno uporabni: iskanje najkrajših poti v grafu, optimiranje poti trgovskega potnika, problemi v bioinformatiki [62], geologiji [77], razpoznavi govora [71], itd. Formalna, teoretična analiza zahtevnosti algoritmov je v disertaciji združena z implementacijo in empiričnim ovrednotenjem. Algoritme analiziramo na podlagi idealiziranih teoretičnih modelov. Čeprav je teoretična analiza algoritma lahko zelo optimistična, je pomembno upoštevati dejstvo, da tovrstni modeli ne odražajo vedno realnosti v praksi. Z implementacijo in posledičnim empiričnim testiranjem lahko ocenimo, kako bi se tovrstni algoritmi obnesli v praksi, in jih primerjamo z že obstoječimi rešitvami.

Dokazovanje pravilnosti algoritmov je ključnega pomena pri njihovi analizi, saj pomeni zagotovilo, da so dobljeni rezultati tudi to, kar sicer pričakujemo. Bolj podrobna formalna analiza algoritmov vsebuje tudi asimptotično analizo časovne in prostorske zahtevnosti. Na osnovi rezultatov časovne in prostorske zahtevnosti pogosto primerjamo algoritme na teoretični ravni. Poznamo več vrst asimptotičnih analiz, za namene disertacije je najbolj aktualna analiza najslabšega primera in analiza povprečnega obnašanja.

Osrednja tema disertacije je iskanje novih algoritmov in možnih izboljšav, kar seveda vključuje tudi pregled obstoječih algoritmov, ter obstoječih spodnjih mej časovnih in prostorskih zahtevnosti problemov. Temu sledi iskanje možnih izboljšav in novih algoritmov, ter morebitno iskanje izboljšanih spodnjih mej problemov. Za dobljene algoritme formalno dokažemo pravilnost in analiziramo časovno in prostorsko zahtevnost ter jih empirično ovrednotimo. Pogosto se zgodi, da novi algoritmi nimajo boljše asimptotične časovne zahtevnosti v najslabšem primeru, a so kljub temu izredno učinkoviti v praksi. Primer takšnega algoritma nastopi pri urejanju [42], kjer se pa nato uporabi asimptotična analiza pričakovanega časa izvajanja. Tovrstne analize so pogoste tudi pri algoritmih, ki rešujejo problem iskanja najkrajših poti [21, 48, 63] v grafu. Pri analizi pričakovanega časa izvajanja si pomagamo predvsem z matematično analizo strukture najkrajših poti v naključnih grafih [19, 63].

Algoritme je moč med seboj primerjati tudi empirično tako, da jih implementiramo v izbranem programskem jeziku ter merimo čas izvajanja in porabo prostora. V ta namen uporabimo programska jezika `C++` in (za implementacije na grafični procesorski enoti) `CUDA C`. Poleg konkretnih implementacij so za empirično primerjavo potrebni tudi vhodni podatki za dani problem. Vhodne podatke pridobimo iz dveh virov: naključno generirani primeri ter primeri iz uveljavljenih zbirk, kot je na primer `TSPLIB` [70].

Nekateri rezultati doktorske disertacije so bili objavljeni v sledečih člankih:

- [8] A. Brodnik in M. Grgurovič. Speeding up shortest path algorithms. V: K. Chao, T. Hsu in D. Lee (ur.). *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, volume 7676 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2012.

- [9] A. Brodnik in M. Grgurovič. Solving all-pairs shortest path by single-source computations: Theory and practice. *Discrete Applied Mathematics*, 231(Supplement C):119 – 130, 2017. Algorithmic Graph Theory on the Adriatic Coast.

- [7] A. Brodnik in M. Grgurovič. Practical algorithms for the all-pairs shortest path problem. V: A. Adamatzky (ur.). *Shortest Path Solvers. From Software to Wetware*, pages 163–180. Springer International Publishing, Cham, 2018.

- [10] A. Brodnik in M. Grgurovič. Parallelization of ant system for GPU under the PRAM model. *Comput. Informatics*, 37(1):229–243, 2018.

- [6] A. Brodnik, M. Grgurovič in R. Požar. Modifications of the Floyd-Warshall algorithm with nearly quadratic expected-time. *Ars Math. Contemp.*, 22(1):1–22, 2022.

Osnove

Modeli računanja

Izmed teoretičnih modelov računanja se osredotočimo predvsem na dva. Model RAM [80] (angl. *random access machine*) je klasičen model, ki se uporablja pri analizi zaporednih algoritmov. Osnovni gradnik je pomnilnik, ki sestoji iz neomejenega števila registrov, v katerih se nahajajo cela števila. Do registrov dostopamo preko neposrednega naslavljanja v konstantnem času. Pogosto imamo opravka z realnejšim modelom, ki se imenuje besedni RAM (angl. *word RAM*) in dovoljuje izvedbo nekaterih aritmetičnih in logičnih operacij nad registri v konstantnem času. Pri tem je pomembno poudariti, da so registri omejeni na velikost besede (angl. *word*).

Model PRAM [75] (angl. *parallel random access machine*) je razširitev modela RAM, ki omogoča analizo vzporednih algoritmov. Sestoji iz p procesorjev in pomnilnika, ki je deljen med vsemi procesorji. Taki arhitekturi pomnilnika pravimo UMA [44] (angl. *uniform memory access*). Procesorji izvajajo ukaze vzporedno in sinhrono. Pri analizi časovne zahtevnosti ločimo med koračno zahtevnostjo (angl. *step complexity*), ki predstavlja največjo klasično časovno zahtevnost med vsemi procesorji, in delovno zahtevnostjo (angl. *work complexity*), ki predstavlja seštevek klasične časovne zahtevnosti po vseh procesorjih. Po Flynnovi taksonomiji [27] se model PRAM pojavlja

nekje med modeloma SIMD (angl. *single instruction, multiple data*) ter MIMD (angl. *multiple instruction, multiple data*), saj lahko v primeru vejitev različni procesorji izvajajo različne ukaze. Znotraj modela PRAM poznamo več podvrst, v disertaciji se osredotočimo predvsem na model CREW (angl. *concurrent read, exclusive write*) in CRCW (angl. *concurrent read, concurrent write*). Model CREW predpostavlja, da lahko istočasno na posamezno pomnilniško lokacijo piše samo en procesor. Za razliko model CRCW nima take omejitve, torej lahko na vsako pomnilniško lokacijo piše hkrati več procesorjev. Zaradi možnosti sočasnega pisanja v modelu CRCW je potrebno definirati, kaj se zgodi v primeru sočasnega pisanja. To nas privede do modela COMMON CRCW, kjer predpostavimo, da vsi procesorji pišejo enako vrednost ter do modela COMBINING CRCW, kjer se vsa istočasna pisanja na lokacijo združijo z uporabo nekega operatorja, npr. seštevanje, množenje, maksimum, itd. Pri obeh modelih velja, da lahko več procesorjev sočasno bere iz posamezne pomnilniške lokacije.

Grafi

Eden izmed ključnih konceptov, ki se pojavijo v disertaciji, je usmerjen graf ali digraf $G = (V, A)$, ki je sestavljen iz množice vozlišč V (angl. *vertices*) in množice usmerjenih povezav A (angl. *arcs*). Usmerjenost izrazimo z urejenim parom (u, v), kar pomeni, da je to usmerjena povezava iz vozlišča u v vozlišče v. Digrafi, s katerimi se bomo ukvarjali, bodo uteženi, kar pomeni, da bo imela vsaka povezava $(u, v) \in A$ določeno težo $w(u, v)$, ki jo bomo podali s funkcijo $w : A \to \mathbb{R}$. Pot P v G iz vozlišča $v_{P,0}$ do vozlišča $v_{P,r}$ je končno zaporedje $P = v_{P,0}, v_{P,1}, \ldots, v_{P,r}$ paroma različnih vozlišč, kjer je $(v_{P,i}, v_{P,i+1})$ povezava v A, za vse $i = 0, 1, \ldots, r - 1$. Utež poti je vsota uteži vseh povezav vzdolž poti. Najkrajša pot od vozlišča u do vozlišča v je vsaka pot med u in v z minimalno dolžino. Dolžini najkrajše poti od u do v rečemo tudi razdalja od u do v in jo označimo z $D_G(u, v)$. Zaradi preprostejšega zapisa definiramo $n = |V|$ in $m = |A|$.

Kombinatorična optimizacija

Za razliko od problemov zvezne optimizacije – kjer je prostor rešitev zvezne narave – se pri kombinatorični optimizaciji ukvarjamo s problemi, kjer je prostor rešitev diskreten. Načini reševanja problemov kombinatorične optimizacije med drugim vključujejo dinamično programiranje in požrešno metodo. Rešitve problemov zavzamejo vrednost množic, grafov, celih števil in podobnih diskretnih struktur. Uporabnost kombinatorične optimizacije je razvidna iz široke palete praktičnih problemov: optimizacija voznega reda, iskanje najbolj ekonomičnih poti, iskanje minimalnih vpetih dreves, problem nahrbtnika, itd.

Ena izmed oblik optimizacijskih algoritmov so tako imenovani metahevristični optimizacijski algoritmi. Metahevristike so načeloma splošni algoritmi oz. strategije za reševanje problemov optimizacije, ki nam ne zagotavljajo optimalnosti najdenih rešitev. Za preproste probleme, kjer lahko najdemo optimalno rešitev razmeroma hitro, je metahevristika slaba izbira. Veliko je takih problemov, kjer je čas, potreben za iskanje optimalne rešitve, preprosto predolg. To so npr. NP-težki problemi. Veliko metahevristik posnema naravne pojave, sem spadajo algoritmi kot so npr. sistem kolonije mravelj, genetski algoritmi, simulirano ohlajanje, in mnogi drugi [5].

Rezultati

Dinamično programiranje

Za razliko od tehnike *deli in vladaj*, kjer problem razdelimo na podprobleme in jih neodvisno rešimo, imajo problemi, ki jih rešujemo z dinamičnim programiranjem, to lastnost, da se podproblemi prekrivajo in niso povsem neodvisni. Dinamično programiranje [18] je metoda reševanja problemov, kjer problem razdelimo na manjše podprobleme, jih rešimo enkrat in si rešitve zapomnimo. Te rešitve ponovno uporabimo, kadar naletimo na isti podproblem, da se izognemo ponovnemu računanju rešitve. Algoritmi dinamičnega programiranja se uporabljajo pri problemih optimizacije, npr. za iskanje najkrajših poti [26, 83].

Veliko optimizacijskih problemov, ki jih rešujemo z dinamičnim programiranjem, je mogoče prevesti na naslednji problem: za vektor $X = [X_0, \ldots, X_{n-1}]$ in funkcijo $g(x)$, in $1 \leq x \leq n$ ter $0 \leq k \leq n$, želimo izračunati za vse $1 \leq i \leq n$:

$$Y_i = \min_{0 \leq k < i} \{X_k + g(i-k)\}. \tag{6.1}$$

Na ta problem lahko prevedemo algoritme dinamičnega programiranja, ki se uporabljajo v bioinformatiki [62], geologiji [77] in razpoznavi govora [71]. Rešitev enačbe lahko izračunamo po naivni metodi v času $O(n^2)$. Obstajajo algoritmi, ki rešitev poiščejo hitreje, npr. v $O(n \lg n)$, kadar je funkcija g konveksna ali konkavna [34]. V disertaciji pokažemo, da lahko obstoječe algoritme za posebne primere funkcije g posplošimo in dobimo nov algoritem za poljubno funkcijo g. Dobljeni algoritem ima časovno zahtevnost odvisno od števila prevojev funkcije g, ali drugače povedano točk, kjer se koveksnost spremeni v konkavnost oziroma obratno. Pri tem dokažemo spodnjo lemo.

Lema 1 (*Lemma 3.6*)**.** *Dobljeni algoritem, ki deluje za poljubno funkcijo g, poišče rešitev v asimptotičnem času $O(Bn \lg(\frac{n}{B}))$, kjer je B število prevojev funkcije g.*

Najkrajše poti v grafih

Problem iskanja najkrajših poti v grafu je eden izmed klasičnih problemov, ki ga lahko definiramo na dva načina [18]. V prvi različici nas zanimajo najkrajše poti od danega izvornega vozlišča (tudi izvora) do ostalih vozlišč v grafu (angl. *single source shortest path, SSSP*), medtem ko v drugi različici iščemo najkrajše poti med vsemi pari vozlišč (angl. *all-pairs shortest path, APSP*). Problema APSP in SSSP sta tesno povezana. V kolikor imamo algoritem, ki reši problem SSSP, ga lahko uporabimo za reševanje problema APSP. To storimo tako, da izvedemo n poizvedb tipa SSSP, kjer spreminjamo izvorno vozlišče. V povezavi z najkrajšimi potmi pogosto definiramo množico $A^* \subset A$, ki vsebuje povezave, za katere velja, da so vsebovane v vsaj eni izmed vseh najkrajših poti v grafu. Zaradi poenostavitve zapisa definiramo še $m^* = |A^*|$. Dijkstrov in Floyd-Warshallov algoritem sta dva klasična algoritma, ki rešujeta problem najkrajših poti. Obe metodi sta znani že od 1960ih let, s tem da je bil Dijkstrov algoritem deležen nekaj posodobitev in izboljšav, medtem ko je Floyd-Warshallov algoritem še dandanes enak prvotnemu opisu.

V disertaciji dokažemo naslednji izrek:

Izrek 2 (*Theorem 4.9*)**.** *Naj bo ψ algoritem, ki reši problem najkrajših poti iz enega izvora do vseh ostalih vozlišč v grafu z nenegativnimi utežmi. Poleg tega naj bo $T_\psi(m,n)$ čas, ki ga porabi algoritem ψ na grafu z m povezavami in n vozlišči, ter naj bo $S_\psi(m,n)$ prostor, ki ga porabi algoritem na istem grafu. Potem lahko problem najkrajših poti med vsemi pari vozlišč rešimo v času $O(m \lg n + nT_\psi(m^* + n, n + 1))$ in prostoru $O(n^2 + S_\psi(m^* + n, n + 1))$.*

Odvisno od izbire algoritma ψ dobimo različne časovne zahtevnosti. Na primer, ko za ψ izberemo Dijkstrov algoritem, dobimo algoritem, ki je asimptotično ekvivalenten algoritmu Hidden Paths [48]. Dobljeni algoritem se izkaže kot učinkovit tudi v praksi. Algoritem smo primerjali na grafih z enakomerno porazdeljenimi utežmi, kjer se je izkazal za hitrejšega v primerjavi z obstoječimi algoritmi [21, 22, 26, 48]. Nekoliko drugačen algoritem dobimo, če kot ψ vzamemo združitev metode Johnsonovega načina spremembe uteži [47] in topološke ureditve grafa. V tem primeru dobimo algoritem z asimptotično časovno zahtevnostjo $O(m^*n + m \lg n)$, ki rešuje problem APSP na poljubno uteženih usmerjenih acikličnih grafih.

Algoritmi dinamičnega programiranja, kot je Floyd-Warshallov, so pogosto izredno preprosti, velikokrat gre za tri enostavne gnezdene zanke. V disertaciji se sprašujemo, ali je možno to enostavnost zamenjati z nekoliko bolj zapleteno izvedbo, ki bi se izognila nepotrebnim sproščanjem, ter te spremembe analizirati in ovrednotiti tako teoretično kot tudi empirično. Pokažemo, da Floyd-Warshalllov algoritem [26, 83] lahko preoblikujemo v algoritem Tree, ki izkorišča informacije o sami strukturi najkrajših poti in se tako izogne nekaterim nepotrebnim sproščanjem. Posledično pokažemo tudi, da je algoritem Tree bolj učinkovit od Floyd-Warshallovega algoritma tako v teoriji kot tudi v praksi. Dobljeni algoritem ima enostavno implementacijo, ne uporablja posebno zahtevnih podatkovnih struktur in je v empiričnih testih hitrejši od Floyd-Warshallovega algoritma. Na naključnih polnih grafih s 512–4096 vozlišči je nov algoritem od Floyd-Warshallovega algoritma hitrejši za faktor 3–5. Če se osredotočimo samo na število sproščanj, jih novi algoritem za omenjene grafe zmanjša za faktor 10–38. Ravno zaradi take pohitritve smo se tudi lotili analize pričakovane časovne zahtevnosti in dokazali naslednji izrek.

Izrek 3 (*Theorem 4.29*)**.** *Algoritem Tree ima za razred polnih usmerjenih grafov na n vozliščih z naključno izbranimi utežmi povezav, porazdeljenimi enakomerno na intervalu $[0, 1]$, pričakovano časovno zahtevnost $O(n^2 \log^2 n)$.*

Poleg algoritma Tree opišemo tudi algoritem Hourglass, ki je posplošitev algoritma Tree in za katerega obstaja možnost, da je teoretično še hitrejši, vendar te domneve zaenkrat ne uspemo dokazati.

Problem iskanja najširših poti v grafu je soroden problem problemu iskanja najkrajših poti, le da tu namesto uteži poti definiramo širino poti in sicer kot minimalno utež povezave na poti. Najširša pot od vozlišča u do vozlišča v pa je vsaka pot med u in v z maksimalno širino. Tudi tu ločimo dve različici. Pri problemu iskanja najširših poti v grafu z danim izvornim vozliščem [68] (angl. *widest path problem* ali *bottleneck shortest path problem*) torej iščemo najširše poti od izvora do vseh ostalih vozlišč v grafu. V drugi različici se sprašujemo po najširših poteh med vsemi pari vozlišč v grafu. V disertaciji pokažemo, da je problem iskanja najširših poti v grafih rešljiv v asimptotičnem času $O(m^*n+m \lg n)$. Pokažemo tudi, da ima problem iskanja najširših poti v grafu določene podobnosti s problemom dinamične hrambe tranzitivne ovojnice.

Sistem mravelj

Pri problemu trgovskega potnika [18] (angl. *traveling salesman problem, TSP*) imamo podan poln graf K_n, kjer vozlišča predstavljajo mesta in povezave predstavljajo povezave med mesti. Povezave so utežene z nenegativno funkcijo $w\colon A \to \mathbb{R}^+$ tako, da utež povezave predstavlja razdaljo med mestoma, ki ju povezuje. Želimo poiskati pot $P = v_{P,0}, \ldots, v_{P,n-1}$, ki minimizira naslednjo vrednost:

$$w(v_{P,n-1}, v_{P,0}) + w(P).$$

Sistem mravelj [24] deluje tako, da požene na grafu več umetnih mravelj z nalogo iskanja najcenejše poti. Algoritem posnema metodo, ki jo uporabljajo mravlje pri iskanju poti od mravljišča do nahajališča hrane ter nazaj. Mravlje postopoma gradijo rešitev s premikanjem po grafu. To počnejo stohastično na podlagi feromonskega modela. Feromonski model ni nič drugega kot množica parametrov, vezanih na povezave grafa, ki jih lahko mravlje spreminjajo med izvajanjem algoritma.

Eden izmed ciljev disertacije je iskanje strategij povzporejanja zaporednih metahevristik, ki bodo dobre rešitve našle hitreje, kadar imamo na voljo več procesorjev. Osredotočimo se na problem trgovskega potnika, ki je NP-težek. Posvetimo se povzporejanju sistema mravelj v čisto teoretičnem smislu na modelu PRAM in rezultate tega povzporejanja nato prevedemo tudi v prakso z implementacijo na GPE. Gre za nekoliko drugačen pristop pri snovanju vzporednih algoritmov na GPE, kjer je pogosto teoretična analiza učinkovitosti povsem izpuščena.

V disertaciji pokažemo, da je algoritem sistema mravelj za GPE [12, 53, 85] mogoče prevesti na PRAM, kar omogoča natančnejšo teoretično analizo. Posledično ugotovimo, da je postopek posodobitve feromonske matrike tako počasen, da povečuje asimptotično koračno zahtevnost celotnega algoritma. Podamo nov algoritem za posodobitev feromonske matrike na PRAM in tako dokažemo naslednji izrek:

Izrek 4 (*Theorem 5.1*)**.** *Posodobitev feromonske matrike lahko opravimo s koračno zahtevnostjo $O(n)$ in delovno zahtevnostjo $O(n^2)$ na CREW PRAM z uporabo n procesorjev.*

Z novim načinom posodobitve feromonske matrike pohitrimo celoten algoritem sistema mravelj, in sicer na koračno zahtevnost $O(n \lg n)$ na CREW in $O(n \lg \lg n)$ na CRCW ter delovno zahtevnost $O(n^3)$ v obeh modelih računanja z uporabo n^2 procesorjev. Izboljšave sistema mravelj za PRAM, opisane v disertaciji, je moč uspešno prevesti tudi nazaj na GPE. Po prevedbi na GPE praktično ovrednotimo nov način posodobitve feromonske matrike v primerjavi z obstoječimi ter pri tem za vhodne podatke uporabimo zbirko `TSPLIB` [70]. Ugotovimo, da je nova metoda hitrejša v primerih, ko nimamo na voljo atomarnih ukazov, kar se zgodi npr. na nekaterih starejših GPE, ki jih dejansko modeliramo kot CREW PRAM. Novejše GPE v resnici lahko modeliramo kot COMBINING CRCW PRAM, kjer je agregatna funkcija seštevanje.