On the Quantum Hardness of Matching Colored Triangles

MSc Thesis *(Afstudeerscriptie)*

written by

Koen Leijnse
(born April 24th, 1996 in The Hague, Netherlands)

under the supervision of **Harry Buhrman** and **Florian Speelman**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
07-07-2022	Harry Buhrman (co-supervisor)
	Ronald de Haan
	Maris Ozols
	Florian Speelman (co-supervisor)
	Yde Venema (chair)

Institute for Logic, Language and Computation

I would like to dedicate this thesis to Soppie and Kees

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Acknowledgements

I would like to acknowledge my supervisors Florian, Subha and Harry for guiding me through the thesis. Our meetings were always very fun and inspiring, you made the whole experience enjoyable and stress-free.

Furthermore, I would like to thank Andris Ambainis for providing us with the idea to use Variable Time Grover Search to solve Δ-Matching Triangles and Triangle Collection quickly.

Also my friends, for keeping me sane.

My family, for supporting me whatever I do.

And God, for granting me insight and knowledge.

Abstract

Classically, many problems can be shown to have computational lower bounds in their time complexity conditioned on the hardness of three popular problems; k-SAT, 3SUM and APSP. This is done through fine-grained reductions and research in this classical field has recently exploded, as can be seen in V. Williams' overview [52]. The reductions from all three popular problems to two intuitive graph triangle problems, Δ-Matching Triangles and Triangle Collection, makes for a notable result in the field of fine-grained computational complexity [3]. By reducing all three problems to the two graph triangle problems Abboud, V. Williams and Yu prove that if we find an $O(n^{3-\epsilon})$ time algorithm, with n the number of nodes and $\epsilon > 0$, for either of the two graph triangle problems, all three hardness conjectures must be false. This result makes the two problems very worthwhile to work with.

Many computational problems allow for speed-ups on quantum computers and recent work has taken the concept of fine-grained reductions to quantum computers to prove many new conditional quantum lower bounds based on quantum hardness conjectures for k-SAT and 3SUM [1, 6, 13, 15]. Continuing this work, we formulate an $n^{2.5-o(1)}$ quantum hardness conjecture for APSP and provide arguments for why this hardness conjecture is likely to be valid in the quantum case. Based on this conjecture, we prove a series of quantum lower bounds for graph and matrix problems. Starting at APSP we follow the classical chain of reductions from [3, 53] all the way through to Δ-Matching Triangles and Triangle Collection, first reviewing the classical reductions and then applying them in the quantum setting. We do the same for k-SAT, proving the quantum hardness of the two graph triangle problems based on the Quantum Strong Exponential Time Hypothesis from [1], using classical reductions from [3]. Combined with quantum hardness results based on 3SUM proven in [13], we show that an $O(n^{1.5-\epsilon})$ algorithm for Δ-Matching Triangles, with $\omega(1) \leq \Delta \leq n^{o(1)}$, or Triangle Collection would contradict all three quantum hardness conjectures, mirroring the classical result.

The thesis finishes by showing matching upper bounds for all problems that we reduce APSP to. For Δ-Matching Triangles and Triangle Collection we find matching upper bounds by using the Variable Time Grover Search algorithm from [6] as suggested by Ambainis. For all other problems, matching upper bounds are found through a simple application of Grover Search.

Contents

1 Introduction **4**
- 1.1 Results . . . 7
- 1.2 Thesis Outline . . . 8
- 1.3 Related Work . . . 10

2 Preliminaries **13**
- 2.1 Classical Computational Complexity . . . 13
  - 2.1.1 Efficient Computation . . . 14
  - 2.1.2 Random-Access Machines . . . 16
  - 2.1.3 Complexity Classes . . . 18
  - 2.1.4 Randomized Algorithms . . . 20
  - 2.1.5 Circuits . . . 21
- 2.2 Quantum Computing . . . 21
  - 2.2.1 Bounded-Error Quantum Time . . . 25
  - 2.2.2 Quantum Queries and Oracles . . . 26
  - 2.2.3 Quantum Speed-Ups . . . 27
- 2.3 Overview . . . 29

3 Known Fine-Grained Reductions **31**
- 3.1 The Fine-Grained Model . . . 32
  - 3.1.1 Coarse-Grained Reductions . . . 32
  - 3.1.2 Fine-Grained Reductions . . . 33
  - 3.1.3 Popular Hardness Conjectures . . . 34
- 3.2 Fine-Grained Reductions From APSP . . . 42
  - 3.2.1 APSP and Matrix Multiplication . . . 42
  - 3.2.2 Graph Triangle Finding . . . 46
- 3.3 Reductions from 3SUM, SAT and k-SAT . . . 55
  - 3.3.1 SAT and k-SAT . . . 56
- 3.4 Extremely Popular Conjecture . . . 61

4 Quantum Fine-Grained Conditional Lower Bounds **63**
- 4.1 Hard Quantum Problems . . . 63
- 4.2 Quantum Fine-Grained Reductions from APSP . . . 70

4.3 Quantum Fine-Grained Reductions from SAT and k-SAT . . . 75

5 Quantum Upper Bounds **78**
5.1 Delta Matching Triangles and Triangle Collection 79

6 Conclusion **83**
6.1 Discussion 83
6.2 Future Work 84

A Graph Definitions **85**

Chapter 1

Introduction

The field of computational complexity is a well-developed and well-studied field. One of the major open questions in the field is whether the class of easily solvable problems (P) the same as the class of easily verifiable problems (NP).[1] In other words: Is it necessarily easier to verify a solution than it is to find one? The question has been around for a long time, and we may wonder whether an answer can even be found. The consensus seems to be that it is more likely that the classes are different. The question of whether P is the same as NP can be rephrased as whether there exists a polynomial time algorithm that can determine whether a propositional logic formula is satisfiable or not. This satisfiability problem, known as SAT and formally defined in Chapter 3, is NP-complete, meaning that SAT is at least as hard as any other problem in NP. This comparable hardness makes it so that if we find an efficient algorithm for just one NP-complete problem, we can solve all NP-problems efficiently. We show NP-completeness through reductions which lets us solve problems by using oracles to other algorithms. Comparable hardness provides us with structure in the computational complexity classes, regardless of the outcome of the P vs. NP question.

The different universes resulting from whether P is the same as NP have been explored in detail by Impagliazzo in [32]. For example, if we had a fast satisfiability algorithm, cryptography as we use it today would fail, since it relies on the assumption that certain parts of the code are 'hard' to break, in the sense that they take an exponential amount of time. A fast satisfiability algorithm could be used to break these codes quickly. While the consequences of finding a fast algorithm for satisfiability would be of great significance, it is much more likely that such an algorithm does not exist and the universe in which this is the case has been studied much more in depth. The Strong Exponential Time Hypothesis (SETH), proposed by Impagliazzo and Paturi

[1] Technical terms such as computational classes, computational problems and other computational lingo will be formally defined in appropriate parts of the thesis.

in [33], states that indeed, no algorithm exists that solves satisfiability in sub-exponential time. This is a stronger assumption than simply assuming that $\mathsf{P} \neq \mathsf{NP}$: It is possible that $\mathsf{P} \neq \mathsf{NP}$ but satisfiability can be solved in sub-exponential, i.e. $2^{o(n)}$, time, since that would not admit the satisfiability problem to the class P. From this hypothesis we can prove plenty of interesting results, mainly the conditional hardness of many computational problems in NP using reductions as shown in [18].

Reductions were commonly used to show that a problem is in P by reducing it to another problem in P or that a problem is NP-complete by reducing another NP-complete problem to it. These reductions don't tell us much about the actual run time of our problems; a problem can be in P, but still require time of some very high polynomial degree. In practical settings, such as applications of algorithms, we are interested in the exact run times of problems, at least asymptotically. Cygan, Dell, Lokshtanov, Marx, Nederlof, Paturi, Saurabh and Whalström showed in [18] that many other NP-hard problems have a run time equivalent to SAT. This implies that under SETH, no sub-exponential algorithms can exist for these problems either. We already knew that these problems are NP-hard, but the definition of NP-hardness combined with SETH alone does not exclude the possibility of a sub-exponential algorithm existing for an NP-hard problem other than SAT. To prove these results, Cygan et al. used reductions that preserve sub-exponential run times, a guarantee that is not provided by the standard Turing reductions.

This line of thinking, where we use the conjectured hardness of a single problem to show conditional lower bounds of many other problems, was taken a step further recently by proving hardness of many computational problems in P based on the conjectured hardness of 3SUM and APSP [27, 41, 43, 53]. In the 3SUM problem we are tasked with finding three integers in a set of integers that sum to 0, while APSP is a popular graph problem with applications in networks, where we have to find the shortest paths between all nodes in a graph. Just like SAT, the problems 3SUM and APSP are popular computational problems with an upper bound on their complexity that has not been improved significantly in many years [51]. There are many computational problems in P for which we have no reductions from SAT, so it seems natural to formulate conjectures similar to SETH for the other popular problems that have had no prominent speed-up in a long time.

The conditional lower bounds are proven by conjecturing the hardness of one of the three problems and using fine-grained reductions, a special type of computational reduction that lets us prove lower bounds of specific polynomial degree. Where the comparable hardness of NP-complete problems versus problems in P is coarse grained in the sense that it draws a coarse conditional boarder between the complexity classes P and NP, showing conditional lower bounds of polynomial degree in the input size is fine-grained: We show com-

parable hardness *within* the class P. We can then prove lower bounds on the specific asymptotic run time of many computational problems based on one of the three hypotheses.

It would of course be great if we had a reduction between any two of the three problems, as this would make one conjecture weaker than the others. If we were able to prove, e.g. a conditional lower bound on 3SUM from APSP, then conjecturing hardness of 3SUM would imply hardness of APSP, since we'd be able to use a fast 3SUM algorithm to solve APSP fast as well. Unfortunately this is not the case, but we do have the next best thing, problems to which all three hard problems reduce. In [3] Aboud, V Williams and Yu found two problems, Δ-Matching Triangles and Triangle Collection, to which all three hard problems reduce. Both Δ-Matching Triangles and Triangle Collection are intuitive graph triangle problems: In Δ-Matching Triangles we are tasked to find a triple of node colors for which there are Δ triangles in a node-colored graph and for Triangle Collection we are asked whether there is a triangle in the graph for every possible triple of colors. Finding an $O(n^{3-\epsilon})$ algorithm for one of these problems would then imply that all three popular hardness conjectures are false, making the graph triangle problems very worthwhile to work with.

Since many of the computational lower bounds proved using fine-grained reductions are of polynomial degree, it becomes very important to be clear about what computational model we are using to run our reductions on. For coarse-grained hardness this is not as important: the common classical computational models give rise to the same complexity classes P and NP. We often assume when working in fine-grained computational complexity that our machines are classical and have random access memory. We can make these assumptions, because they model how physical computers work. With the rise of quantum technologies, it becomes more and more relevant to take classically established frameworks and results and consider the advantages that could come with changing the classical computational model to a quantum model. We want our computational models to reflect what our physical computers are capable of and although quantum computers come with many physical challenges, they also promise additional computing power.

In quantum computing we let our computational models work with qubits, allowing for superposition of bits, entanglement and interference effects. The result is that many computational problems can be solved faster on a quantum computer than on a classical computer. Many of the speed-ups are due to variations and different applications of Grover Search [29], which provides a quadratic speed up as compared to classical search algorithms. As such, any computational problem that has a search algorithm embedded can likely be sped-up on a quantum computer.

Since Grover Search provides only a quadratic speed-up, it doesn't let us solve NP-complete problems in polynomial time. As of this moment, there is no known efficient quantum algorithm for any NP-complete problem, although we do have an efficient quantum algorithm for an NP problem that is not known to be in P. In 1994 Shor found a quantum algorithm that efficiently decomposes composite numbers into smaller integers [45]. Because there exists no classical algorithm that solves integer factorization in polynomial time, the speed-up on a quantum computer is exponential. However, integer factorization is not NP-complete, and the speed-up does not have any bearing on whether quantum computers can solve all NP problems efficiently.

In recent research the ideas from fine-grained computational complexity have been extended to quantum computational models with the formulations of quantum hardness conjectures for SAT and 3SUM [1, 15, 13]. By applications of Grover's Search algorithm we find speed-ups for all three popular problems and the lower conjectured hardness of the popular problems leads to smaller lower bounds for the problems they are reduced to. Our popular problems don't have as much history in the quantum setting as they do in the classical setting, and it might therefore be unclear how we should formulate our hardness conjectures. Many of the problems that we reduce to have matching upper bounds through simple applications of Grover Search, which to a certain extent validates our hardness conjectures: Suppose our conjectured lower bound is too big. Then, if we reduce our popular problems to many other computational problems, we'd only have to find an algorithm for one of these problems that beats the proven conditional lower bound to show that our conjectured hardness was indeed too big. As such, the more problems we reduce to and the more matching upper bounds we find, the more likely we are to have arrived at the correct hardness conjecture.

The plan for this thesis is to continue research in quantum fine-grained complexity in the natural direction: We formulate a quantum hardness conjecture for the third popular problem, APSP, and investigate the known classical fine-grained reductions to find out what conditional lower bounds we can prove in the quantum setting. We also continue the unifying approach of reducing all three popular problems to the common problems Δ-Matching Triangles and Triangle Collection.

1.1 Results

We explore the quantum fine-grained computational complexity landscape, starting by adding a quantum hardness conjecture for APSP to the two previously formulated hardness conjectures for k-SAT and 3SUM.[2] Classically

[2]Note that in the previous section we used SAT to refer in general terms to the satisfiability problem. Since SETH is formally defined for k-SAT and implies hardness of SAT,

APSP is conjectured to have no $O(n^{3-\epsilon})$ time algorithm where n is the number of nodes in the graph and for $\epsilon > 0$. Although in fine-grained literature we often work with the shortest distance version of APSP, the cubic lower bound is conjectured for both the version of APSP where we are tasked to output the explicit paths between all points and the one where we have to output the shortest distances between all points. We investigate natural speed-ups to the classical algorithms for both versions and arrive at an $n^{2.5-o(1)}$ lower bound for both versions of APSP in the quantum setting.

Having 'natural' algorithms for APSP that match the conjectured lower bound is of course not a very strong argument for the conjecture being valid. Furthermore, we can't claim that an $O(n^{2.5-\epsilon})$ algorithm has not been found for many years, as in the classical case. However, for all conditional lower bounds that follow, we find matching quantum upper bounds, further reinforcing the validity of our quantum hardness conjecture for APSP as discussed in the previous section. We reduce APSP to Δ-Matching Triangles and Triangle Collection and prove many conditional lower bounds for intermediate problems in the process. We also prove the same lower bounds for Δ-Matching Triangles and Triangle Collection conditioned on the quantum hardness of k-SAT and, using a result from [13], conditioned on the quantum hardness of 3SUM. This lets us prove a lower bound of $n^{1.5-o(1)}$ to both Δ-Matching Triangles, for $\omega(1) \leq \Delta \leq n^{o(1)}$, and Triangle Collection based on a quantum version of the extremely popular hardness conjecture from [3].

Lastly, for all problems for which we prove quantum lower bounds we provide matching upper bounds. In most cases this is done through a simple application of Grover Search. In the cases of Δ-Matching Triangles and Triangle Collection we find matching upper bounds by applying Variable Time Grover Search from [6].

An overview of all relevant classical- and quantum lower bounds based on hardness conjecture for APSP can be found in Table 1.1 below. We also provide the corresponding upper bounds.

1.2 Thesis Outline

Here we give a brief overview of how the thesis is structured.

After this introduction we continue the thesis with a chapter on preliminary knowledge in Chapter 2. We introduce some basic concepts from computational complexity and quantum computing and discuss our computa-

we talk about k-SAT when discussing the formal hardness of satisfiability. We make the distinction between the two problems more clear in Chapter 3.

Problem		Classical	Quantum
$(\min,+)$-Matrix Multiplication	Lower bound	$n^{3-o(1)}$ [25, 37]	$n^{2.5}$ Theorem 4.2.2
	Upper bound	$O(n^3)$ $(*)$	$O(n^{2.5})$ $(**)$
All-Pairs Negative Triangle	Lower bound	$n^{3-o(1)}$ [53]	$n^{2.5}$ Theorem 4.2.3
	Upper bound	$O(n^3)$ $(*)$	$O(n^{2.5})$ $(**)$
Negative Triangle	Lower bound	$n^{3-o(1)}$ [53]	$n^{1.5}$ Theorem 4.2.4
	Upper bound	$O(n^3)$ $(*)$	$O(n^{1.5})$ $(**)$
0-Weight Triangle	Lower bound	$n^{3-o(1)}$ [54]	$n^{1.5}$ Theorem 4.2.5
	Upper bound	$O(n^3)$ $(*)$	$O(n^{1.5})$ $(**)$
Δ-Matching Triangles	Lower bound	$n^{3-o(1)}$ [3] $(***)$	$n^{1.5}$ Theorem 4.2.6 $(***)$
	Upper bound	$O(n^{3-o(1)})$ [3] $(***)$	$O(n^{1.5})$ Corollary 5.1.2 $(***)$
Triangle Collection	Lower bound	$n^{3-o(1)}$ [3]	$n^{1.5}$ Theorem 4.2.7
	Upper bound	$O(n^3)$ $(*)$	$O(n^{1.5})$ Theorem 5.1.2

Table 1.1: Overview of lower bounds based on a hardness conjecture for APSP, both in the classical and in the quantum setting. Corresponding upper bounds are also provided.
$(*)$: These upper bounds are the most straight forward algorithms, like exhaustive search, and therefore have no particular source.
$(**)$: By applying Grover Search, potentially as subroutine.
$(***)$: Holds only for $\omega(1) \leq \Delta \leq n^{o(1)}$.

tional models. In this thesis we will be comparing lower and upper bounds for computational problems on different computational models, and we also mentioned in the introduction that we need our models to have random access memory. It is therefore important to clearly state on which computational models we will be working in the classical and quantum settings respectively.

In Chapter 3 we present the classical framework of fine-grained computational complexity and provide definitions for the computational problems encountered in this thesis. We then review known fine-grained reductions from APSP to Δ-Matching Triangles and Triangle Collection through 0-Weight Triangle, where we have to determine whether a graph contains a triangle of 0 total edge weight. We also review the reductions from k-SAT and SAT to Δ-Matching Triangles and Triangle Collection. This combined with the known reduction from 3SUM to 0-Weight Triangle from [54] lets us restate the classical hardness result from [3] for Δ-Matching Triangles and Triangle Collection based on the disjunction of the three popular hardness conjectures. The problems and classical reductions discussed in this section will be exactly

those for which we prove quantum lower bounds in Chapter 4 and quantum upper bounds in Chapter 5. We don't explicitly review the reductions from 3SUM since the quantum reductions from 3SUM to 0-Weight Triangle have already been discussed in [13].

We proceed in Chapter 4 by considering fine-grained complexity in our quantum computational model. We give an overview of the new hardness bounds for k-SAT and 3SUM in the quantum setting and review quantum algorithms for APSP to arrive at a $n^{2.5-o(1)}$ quantum lower bound for APSP. We then go over the reductions from Chapter 3 to find quantum lower bounds for many computational problems, conditioned on the quantum hardness of APSP. We end up with quantum lower bounds for Δ-Matching Triangles and Triangle Collection and review also the reductions from k-SAT and SAT to these two problems. This combined with the quantum fine-grained reduction from 3SUM to 0-Weight Triangle from [13] lets us prove hardness of Δ-Matching Triangles and Triangle Collection based on extremely popular quantum conjecture.

Chapter 5 is dedicated to discussing quantum upper bounds to the different computational problems encountered in the thesis. Many of the upper bounds are through simple application of Grover Search. Upper bounds for Δ-Matching Triangles and Triangle Collection are found by using Variable Time Grover Search from [6].

We finish the thesis in Chapter 6 with a discussion of the results found and suggestions for future research.

1.3 Related Work

Before fine-grained computational complexity was considered a sub-field, lower bounds implicitly based on the hardness of one of the popular problems were already found. After the hardness conjectures and fine-grained reductions were formally defined, the amount of results has exploded. A lot of recent work on fine-grained computational complexity was spearheaded by V. Williams, who provides excellent overviews of many of the old and recent results in [51, 52].

Work on lower bounds from k-SAT started with the formulation of SETH and ETH in [33] and was continued in [34]. The first hardness results were for other NP-problems, as shown in [18]. Work on proving polynomial lower bounds based on SETH was started in [50], where a quadratic lower bound was proven for the Orthogonal Vectors problem, where an algorithm

is tasked with determining whether a set of vectors contains two orthogonal vectors. Since k-SAT is naturally parameterized, it is also the object of study for parameterized conditional lower bounds, as found in [19]. In recent years many hardness results have been found based on Orthogonal Vectors and SETH, such as hardness for Δ-Matching Triangles and Triangle Collection from [3]. An overview can be found in [52]. Upper bounds for k-SAT are studied in [42].

Proofs for lower bounds based on 3SUM started with the formulation of a hardness conjecture for 3SUM in [27]. Notable reductions are 3SUM's hardness equivalence to Convolution-3Sum [41] and the reduction from Convolution-3Sum to 0-Weight Triangle [54]. In Convolution-3Sum we are asked to determine whether for an array of integers S there exists indices i, j such that $S[i] + S[j] = S[i + j]$. Other recent reductions can be found in the overview in [52]. Although 3SUM is a relatively simple problem, extensive work has been done to improve its run time, which depends heavily on the computational model used and the bounds on the set of integers from which a problem instance may draw its array. Nonetheless, no sub-quadratic algorithm has been found for 3SUM in the general case. Different $n^{2-o(1)}$ run times were found in [9, 16, 30].

The hardness of APSP was first officially formulated in [53], although many earlier results already implicitly used it hardness, such as in [43] and the hardness equivalence between APSP and $(\min, +)$-Matrix Multiplication, also known as the matrix distance product, from [25, 37]. Reductions from $(\min, +)$-Matrix Multiplication to All-Pairs Negative Triangle and Negative Triangle from [53] will feature in this thesis alongside the reduction from Negative Triangle to 0-Weight Triangle from [54] and from 0-Weight Triangle to Δ-Matching Triangles and Triangle Collection from [3]. In Negative Triangle we have to determine whether an input graph contains a triangle of total negative edge weight and in All-Pairs Negative Triangle we have to determine for $O(n^2)$ pairs of nodes in a tripartite graph over $O(n)$ nodes whether they are part of a negative triangle. Cubic upper bounds to APSP have been known for half a century. Famous are: the Floyd-Warshall algorithm, the Bellman-Ford algorithm and lastly Johnson's algorithm, which makes use of Dijkstra's single-source shortest paths algorithm [10, 22, 26, 35, 48]. The best known randomized $O(n^{3-o(1)})$ algorithm for APSP was found in [49].

Not much work has been done in quantum fine-grained complexity. Quantum versions of SETH were formulated in both [1] and [15] with reductions from k-SAT to different problems. The quantum hardness of 3SUM

was investigated in [7, 13], with [13] showing a quantum reduction to 0-Weight Triangle. No formal quantum hardness conjecture for APSP has been made. For showing quantum upper bounds to the different problems that we have quantum reductions to, we make regular use of the Grover Search algorithm from [29], the Grover Minimum Finding algorithm from [24] and Variable Time Grover Search from [6].

Chapter 2

Preliminaries

This chapter introduces the different computational models that appear in the thesis and lays the ground-works for the algorithmic language used throughout. In the first section we discuss different classical models and explain which one is the most appropriate for our work. We also provide some definitions and tools that will help us in the analysis of the complexity of our algorithms. In the second section we discuss the quantum model and which additions to the usual quantum model will be necessary for us. We finish off the chapter with a brief overview, comparison of the models and some final observations.

2.1 Classical Computational Complexity

In this section we describe some concepts from classical computational complexity. We assume that the reader has some familiarity with the topic, but for consistency's sake will briefly go over the basics. We mostly follow [8] and [19] for formalism, conventions and notation.

The field of computational complexity deals with computational problems, the algorithms that allow us to solve them and the resources required to do so. Computational problems are problems that may be solved by algorithms and the algorithms themselves are executed in computational models. The complexity of the algorithms is measured by the time or space required to run them. The different models discussed here are definitions for different machines or circuits that execute these algorithms. In the classical setting we will let our algorithms be executed on machines as opposed to the circuits we will see in the quantum setting.

Computational problems can be easily represented by boolean functions, independent of the model in which they're solved.

We will be comparing a framework for analyzing computational complexity lower bounds in two different computational models. The main parameter that we use for comparison in performance of the algorithms in different

models is their 'time' or 'efficiency', which we will define here first.

2.1.1 Efficient Computation

To judge which computational model is most suitable for our work we start by laying the foundations for analyzing our algorithms.

To execute a set of instructions in a model we need to first translate our instructions into the mathematical language of the model. The time required to solve a problem in a model then depends on the time of the translation and how long it takes to execute the translated instruction in our model of choice. Usually this adds about a constant factor in steps, meaning that every single instruction in the mathematical language of our problem leads to a constant number of instructions in the model. These constant factors are not very meaningful when running the algorithms on very large input. In order to analyze our algorithms properly, we introduce big-O (or asymptotic) notation. Big-O notation also allows us to define asymptotic upper and lower bounds.

Definition 2.1.1 (Big-O Notation)**.** Let f and g be real-valued functions, with $g(n)$ strictly positive for large enough values of n. Then:

- If there are some positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$, then $f(n) = O(g(n))$.
- If there exists some c such that $f(n) = O(g(n)\log^c(g(n)))$, then $f(n) = \tilde{O}(g(n))$.
- If $g(n) = O(f(n))$, then $f(n) = \Omega(g(n))$.
- If for any positive c there exists an n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$, then $f(n) = o(g(n))$.
- If $g(n) = o(f(n))$ then $f(n) = \omega(g(n))$.

We care about the asymptotic complexity of our algorithms and write down the time in big-O notation, which allows us to ignore the time cost of translating our simple algorithmic instructions into model instructions or any other type of asymptotically irrelevant overhead. Generally we parameterize the time complexity in terms of the input size of the problem n. The parameter n may refer to the length of the binary encoding of our problem, but in many cases we parameterize the complexity by a more useful parameter of the problem such as the number of nodes in a graph or distinct variables in a propositional logic formula. If the more useful parameter is linear in the binary encoding of the entire problem instance, the parameter and the

problem encoding will have the same asymptotic size.

Next we need to clarify what it means to *solve* a problem in a model. Our algorithms take a problem instance as input and produce an output, and we can therefore view them as maps. Since the input can be encoded into a binary language, we can view problems more specifically as functions over the domain $\{0,1\}^n$, with n the size parameter for the given problem instance. Decision problems with a 'yes' or 'no' answer then become Boolean functions and problems outputting a different kind of data structure like a list as in APSP will simply output another binary string.

Thinking of computational problems as functions and algorithms as maps allows us to introduce the following notion.

Definition 2.1.2 (Computability)**.** An algorithm R computes a function $f : \{0,1\}^n \to \{0,1\}^m$, if $f(x) = R(x)$ for all $x \in \{0,1\}^n$.

- An algorithm computes a function f in time $O(T(n))$ if the machine computes f and never takes more than $O(T(n))$ time to reach its output state.

- An algorithm efficiently computes a function f if it computes f in time $O(n^c)$ for some constant c.

In computational complexity terminology, efficient therefore means 'in asymptotic polynomial time'. What constitutes as 'time' for our algorithm depends on the specific model in question, but is usually determined by the amount of 'basic operations' the machine performs before arriving in its halting state. Furthermore, it is important to note that in the way the above definition for computable functions is worded we calculate a worst-case time complexity when calculating the time it takes for a machine to compute a function.

We can encode problem instances in binary to provide as input to our algorithm, but there are some problems that arise in doing so. First, encoding very large numbers in binary will require a large amount of bits, so we can't let our problems run over unrestricted sets of integers. In computational complexity, if n denotes the main variable for parameterizing a problem's complexity, e.g., the number of nodes in a graph or the number of variables in a formula, we assume that integers used in describing the problem instance, e.g., edge weights, come from the interval $[-n^c, \ldots, n^c]$ for some constant c. This way we will only need $O(\log(n))$ bits to encode integers.

The second problem that arises is that the input can come in variable size. If we want to be able to say that an algorithm solves a problem, we want the algorithm to be able to solve it for any input size. For any computational problem we then have a family of computable functions. Such families of

Boolean functions define languages as subsets of the set of all strings of any length n,

$$\{0,1\}^* := \bigcup_{n\in\mathbb{N}} \{0,1\}^n.$$

For the family of Boolean functions $\{f_n\}_{n\in\mathbb{N}}$ with $f : \{0,1\}^n \to \{0,1\}$ we have language

$$L_f := \{x : f_{|x|}(x) = 1\} \subseteq \{0,1\}^*.$$

A single element $x \in \{0,1\}^*$ is called an instance.

Definition 2.1.3 (Decidability)**.** Let $L \subseteq \{0,1\}^*$ be a language. An algorithm R decides L in time $O(T(n))$ if for any $x \in \{0,1\}^n$ it holds that $R(x) = 1$ if and only if $x \in L$ and $R(x)$ halts in time $O(T(n))$.

There are many types of computational problems that are not decision problems, e.g. APSP. These problems cannot be represented by a Boolean function. Common concepts such as the complexity classes P and NP are defined for decision problems and for this section we will therefore stick with these definitions. The definitions relying on decidability can easily be extended to work for functional problems as well, by requiring that our machine or circuit outputs the correct bit-string.

We use the same definition of efficiency when it comes to decidability as we did for computability. In fact, whenever efficient is used in this thesis, we mean 'in polynomial time'.

2.1.2 Random-Access Machines

Without giving strict requirements for what an algorithm needs to be capable of and what it is allowed to do, our definitions for efficiency and computability are not of much use yet. There are no clear rules for what an abstract machine needs to look like; so far it just needs to be able to compute functions according to a set of instructions. Since we do want our abstract machine to model physical computers, there are some limitations to what we should and shouldn't allow. There needs to be some way for the machine to be *implementable*.

The first and most well known computational machine model is the Turing Machine (TM). The TM should be familiar to the reader and is not the model that we end up using for our analysis. However, since it can be considered as a baseline for computational machines, we give a short informal description of the model.

A TM is an abstract machine that consists of a series of tapes with symbols and a set of instructions for reading and rewriting the tapes.

More specifically, a TM is a collection of tapes divided into cells, each containing a symbol. One of the tapes contains the machine input and another will contain the machine output. The machine then starts in some input state and uses a predefined set of instructions, provided by the transition function to produce an output. While executing the instructions, the machine reads and alters the symbols on the tapes using tape-heads and the transition function tells the machine whether to move the tape-heads left, right or to stay. When the tape heads reach a halting state, the output is read from the output tape. Time on a TM is easily defined as the number of state transitions required to go from the starting to the halting state.

There are many types of TM's; machines with multiple tapes and with different alphabets. The simplest type are single tape machines with a binary alphabet: We have one tape that serves as input, write and output tape with alphabet $\{0, 1\}$. Restricting our TM's to a single tape can increase the run-time at most quadratically compared to multi-tape machines [8], but since this is not our actual working model, this won't pose an issue.

In many areas of computational complexity, even this quadratic increase from working on single-tape TM's is of no concern, since any function that is efficiently computable on a single-tape TM will be efficiently computable on a multi-tape TM and vice-versa. In this thesis we work with algorithms that have polynomial run times and as a result, *do* care about these differences in polynomial factors and another problem arises for why TM's are not the most suitable machines for working in polynomial levels of complexity.

Consider for example a graph problem like the APSP, represented on a graph with n nodes. We translate the adjacency matrix A to a binary code and at some point during our algorithm we want to access the value $A[ij]$. This value will be stored somewhere in the binary code, and it may take us $O(n^2)$ transitions of the tape head to read it. Our actual computers or physical machines are designed to have direct access to such a value, and we want our abstract machine to model that. Furthermore, the complexity of many popular algorithms, such as binary search, is measured under the assumption of random-access to certain data-structures. To incorporate random-access in our model we define the random-access machine (RAM) from [46].

The RAM is a machine that can:

- Execute simple operations such as basic arithmetic or if/else statements in a single time step.
- Store an infinite amount of integers.
- Access memory in a single time step.

The main feature here is of course the fact that we can always access stored integers in a single time step. The 'simple operations' specified for the RAM are the same operations that a multi-tape TM can perform in constant time. We can therefore view a RAM as a TM with memory tapes whose memory can be accessed in constant time. More formally, this would work by writing down a memory address on a write-tape, then querying the specified memory address and receiving the contents of the memory cell on an output tape. In this more formal description, accessing a cell of a bit-string of length n would take $O(\log(n))$ for writing down the bit index in binary. In our formulation of RAM we choose to let our machine access memory in a single time-step. We are more interested in polynomial bounds, and it is not unconventional to ignore these logarithmic factors altogether.

Formulating the RAM in this somewhat loose way allows us to be quite flexible in describing algorithms later on during the thesis. We don't have to stick close to a very rigorous mathematical language or encode instructions into binary and can instead describe our algorithms more intuitively.

It can be shown that any algorithm executed on a RAM in time $O(n^c)$ for some constant c can be executed on a TM in time $O(n^{ck})$ for some constant $k \geq 0$ [17]. That means that any problem that is efficiently decidable on a TM is efficiently decidable on a RAM and vice-versa, but as expected, it is still very possible that a RAM outperforms a TM asymptotically, which is why we settle on the RAM as the appropriate computational model for our work.

2.1.3 Complexity Classes

TM's, RAM's and other common computational models give rise to the same classes of efficiently decidable languages and verifiable languages. It is therefore useful to explicitly define the classes.

Definition 2.1.4 (P and NP)**.** Let $L \subseteq \{0,1\}^*$ be a language.

- The language L is in P if there is an algorithm that efficiently decides it.
- The language L is verified efficiently by an algorithm R if R runs in polynomial time and if it holds that $x \in L$ if and only if there is some $u \in \{0,1\}^{O(|x|)}$ such that $R(x,u) = 1$.
- The language L is in the complexity class NP if it can be verified efficiently by an algorithm.

In the above definition, verification of a language means that given a solution or certificate to a problem, we can check efficiently whether the solution

is correct. If we were to allow non-determinism in our machines, we could in polynomial time compute one of an exponential amount of different solutions. Since the space of certificates is exponential in the size of the instance x, an alternative definition for the class NP is 'the class of languages that can be decided by a non-deterministic TM'. We'll use the definitions synonymously. Clearly it must be that $\mathsf{P} \subseteq \mathsf{NP}$, but whether $\mathsf{P} = \mathsf{NP}$ is an important open problem in computational complexity.

Whether it is true that $\mathsf{P} = \mathsf{NP}$ may also be phrased as whether there exists a language that is in NP but not in P. The most interesting candidates are NP-complete problems. If we were to find a fast algorithm for an NP-complete problem, we'd be able to solve all NP problems efficiently through reductions. Before we go deeper into the notion of reductions, we need to talk about oracles.

In some cases we may access the solution to certain problem instances in a black box setting, and we can do that by querying an oracle. A RAM can in one time-step query an instance $x \in \{0,1\}^*$ to an oracle $O \subseteq \{0,1\}^*$ to find out whether x is contained in the language represented by O. Of course constructing the specific oracle instance may take the RAM more than constant time.

In cases where we don't have an oracle to a desired language L, we can simulate one by constructing an algorithm for L and then using the algorithm as a black-box. Whenever we query the simulated black-box we incur a cost in time complexity.

The idea to simulate oracles does require another component of our RAM that we haven't discussed; the capacity to simulate other RAM's. In the TM model we have the universal TM and similarly there is such a thing as a universal RAM, the Random-Access Stored-Program machine (RASP). We add the simulation of other RAM's to the list of what our RAM is capable of.

If we can decide a language L efficiently using an oracle to an NP language, then finding an efficient algorithm for the NP oracle would allow use to decide L efficiently as well, using a simulated oracle. We use the definition of Turing reductions to make this concept more clear.

Definition 2.1.5 (Turing Reductions)**.** A language L is Turing-reducible to a language L', denoted as $L \leq_T L'$ if there is an algorithm with oracle access to L', denoted $R^{L'}$ that efficiently decides L.

- A language L is NP-hard if $L' \leq_T L$ for any L' in NP.
- A language L is NP-complete if it is NP-hard and in NP.

Reductions give us a notion of comparable hardness: Any NP-complete problem is at least as hard as any NP-problem. If we find an algorithm

for any NP-complete language that computes it efficiently, we know that it holds for every language in NP that it is also in P: Let L be a language in NP and L' the NP-complete language for which we have an efficient algorithm. Since L' is NP-complete, there is some oracle machine with oracle L' that decides L efficiently. We use this machine but instead of querying the L' oracle, we simply run the machine that decides L' efficiently.

2.1.4 Randomized Algorithms

The object of this thesis is to investigate potential speed-ups of many computational problems on quantum computers as compared to classical computers. The interesting quantum algorithms naturally turn out to be probabilistic due to the uncertainty inherent to quantum measurements. To be sure that our speed-ups come from quantum effects and not merely from allowing randomization, we need to include randomization in our classical machines as well.

On the level of the model, this simply means that at every step of the algorithm our machines are allowed to make decisions based on probability distributions. The consequence is that we end up with new complexity classes.

Definition 2.1.6. Let $L \subseteq \{0,1\}^*$ be a language.

- The language L is in BPP if there is a randomized algorithm R such that if $x \in L$ it holds that $R(x) = 1$ with probability at least $\frac{2}{3}$ and if $x \notin L$ it holds that $R(x) = 0$ with probability at least $\frac{2}{3}$.

- The language L is in MA if there is a randomized verifier algorithm R such that the following holds. If $x \in L$ then it must be that there exists a $u \in \{0,1\}^{O(|x|)}$ for which $R(x,u) = 1$ with probability at least $\frac{2}{3}$. If $x \notin L$ then it must be that for all $u \in \{0,1\}^{O(|x|)}$ it holds that $R(x,u) = 0$ with probability at least $\frac{2}{3}$.

It is clear that P $\subseteq$ BPP and NP $\subseteq$ MA. It is an open question whether P = BPP, although it is believed to be indeed the case. Many problems for which efficient randomized algorithms had been found and for which no efficient deterministic algorithm was known to exist were eventually solved efficiently by a deterministic algorithm. Different derandomization techniques can be used to turn a random algorithm into a deterministic algorithm with only polynomial overhead [36]. For our comparative analysis this polynomial overhead could of course be problematic, so we will stick to randomized algorithms for our work. Similar arguments can be made for whether NP = MA.

2.1.5 Circuits

The computational model of TM's that we've talked about is one of many classical computational models and because it models how classical algorithms work and in a somewhat circular way, ends up defining what an algorithm is by the Church-Turing thesis. Our classical RAM is essentially a TM with additional functionality for memory access. Although there is such a thing as a Quantum Turing Machine, the most workable computational model for performing quantum computations is that of quantum circuits, and we would rather add random-access capacity to the quantum circuit model than the quantum TM model. In this section we'll therefore give an overview of classical circuits first.

The classical or Boolean circuit model is equivalent to the TM and RAM models, in that any function that can be efficiently computed on a RAM can be efficiently computed by a Boolean circuit and vice-versa. As opposed to a machine with tapes and tape-heads, we can consider applying boolean gates to our input string to generate an output bit.

Definition 2.1.7 (Boolean Circuit)**.** A Boolean circuit C_n of input size n is a directed acyclic graph with n nodes that have only outgoing edges, called sources and 1 node having only incoming edges, called a sink. All other vertices are called gates and are labelled by the logical operations AND, OR and NOT. The size of C is the number of nodes in C. For $x \in \{0,1\}^n$, the output $C(x)$ is defined naturally by assigning bit values to every node recursively depending on the value of the nodes from the incoming edges and the gate label.

Of course, we need to be careful now about how we define efficiency, since it is not obvious how time in the TM language translates to circuit time. The time it takes for a circuit to compute a function is given by the circuit-size complexity, which is the total amount of logic-gates in the circuit. We often refer to circuit size as time complexity as well.

Another key difference is that while a TM is uniformly defined for any input string length, a circuit acts on strings of a specified length n. When providing circuits for deciding a language, we describe a family of circuits.

Definition 2.1.8 (Circuit decidability)**.** A family of circuits $\{C_n\}_{n\in\mathbb{N}}$ decides a language $L \subseteq \{0,1\}^*$ if for any $x \in \{0,1\}^*$ it holds that $C_{|x|}(x) = 1$ if and only if $x \in L$.

2.2 Quantum Computing

In this section we provide some background information for quantum computing with formalism and conventions taken from [55] and [40]. Again, some

familiarity is assumed but for the sake of consistency we start with the basics.

The quantum circuit model uses quantum gates as opposed to boolean gates and applies them to qubits as opposed to bits.

When working with quantum algorithms, what our algorithms can and can't do becomes a lot less intuitive and more mathematical rigor is required. While we can still represent quantum circuits as graphs, it is more useful to think of them using the Hilbert space formulation of quantum mechanics.

Definition 2.2.1 (Hilbert Space and Quantum States)**.**

- A Hilbert space $\mathcal{H}$ is a complex vector space equipped with an inner product.
- A pure quantum state $|\psi\rangle$ is an element of a Hilbert space with inner product equal to 1.
- The complex conjugate of a state $|\psi\rangle$ is denoted $\langle\psi|$.
- The inner product of states $|\psi\rangle$ and $|\phi\rangle$ is denoted $\langle\psi|\phi\rangle$.
- We use shorthand $|\phi\psi\rangle = |\psi\rangle \otimes |\phi\rangle$ to denote the tensor product of states.

In quantum computing we will generally only be dealing with pure states and not with mixed states. As a consequence, when talking about quantum states, we will always be talking about pure states. Furthermore, Hilbert spaces can be infinite, but since we always deal with a finite number of qubits, we may stay with finite quantum mechanics. Lastly, we restrict our attention to real valued Hilbert spaces.

Definition 2.2.2 (Qubit)**.** A qubit is a state $|\psi\rangle$ in a two-dimensional Hilbert space $\mathbb{R}^{\otimes 2}$. A qubit in the state 0 is denoted $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and a qubit in the state 1 is denoted $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$. The states $|0\rangle$ and $|1\rangle$ are called the computational basis states.

- A qubit-string $|\psi\rangle$ of length n is an element of a 2^n-dimensional Hilbert space. We write $|\psi\rangle = \sum_{i\in\{0,1\}^n} \alpha_i |i\rangle$ with complex α_i and $\sum_{i\in\{0,1\}^n} \alpha_i^2 = 1$. The j'th element of the string i in the above formula for $|\psi\rangle$ denotes the value of the j'th qubit.
- The Hadamard basis consists of the states $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$.

The main point of course in our above definition for the qubit is that our bit strings can be in superposition over many strings.

In the quantum circuit model, we have an n-qubit bit-string described by a normalized 2^n-dimensional vector as input. Gates in the circuit correspond to operations on the string and since our input is a vector, these operators are matrices. Furthermore, since our states need to always have unit length, we can restrict our quantum operations to unitary matrices. It follows that unitary matrices represent exactly those operations that send qubits to qubits.

Definition 2.2.3 (Quantum Gates)**.** A quantum gate acting on a single qubit is a 2×2 unitary matrix U.

- A quantum gate acting on n qubits is a $2^n \times 2^n$ unitary matrix.
- Common quantum gates are:
  - The X-gate (also called NOT-gate): $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
  - The Y-gate: $Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$.
  - The Z-gate (also called phase-gate): $Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$.
  - The Hadamard gate: $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

The most common gates are the X-gate, which simply flips a qubit, the Z-gate, which adds a negative to the computational basis state $|1\rangle$ and the Hadamard-gate, which allows us to prepare uniform superpositions from computational basis states. Another type of gate that we will regularly encounter and that can be constructed from the above elementary gates are controlled gates, mainly controlled-NOT-gates (CNOT) and controlled-phase-gates. If we'd let ourselves use any possible unitary as elementary quantum gate, we would be able to decide any decidable language in unit time, since any sequence of unitaries is again a single unitary. As a result, in quantum computing we often limit ourselves to the use of elementary gate sets of easily implementable quantum operations. For our purpose we allow use of all single-qubit gates and the CNOT-gate. Circuit complexity is then defined as the number of elementary gates we need for a quantum circuit to decide a language.

It is easy to see that a quantum circuit can do anything a classical circuit can do at least as fast. We can simply let the computational basis states $|0\rangle$ and $|1\rangle$ represent the classical states 0 and 1 and we can simulate classical propositional logic gates using our quantum gates.

Now we run into the same issue as in the classical case where we want our model to be able to access stored memory efficiently. On top of having random access, we want our quantum models to be ably to query memory bits in superposition, to fully access the power of quantum computing. We do this by allowing quantum-Random-Access-Gates (qRAG's) in our quantum circuit, using the definition from [14].[1]

Definition 2.2.4 (Quantum-Random-Access-Gates)**.** A Quantum-Random-Access-Gate is a $O(n2^n)$ unitary qRAG acting on n memory bits and $\log(n)+1$ work-bits that operates as follows:

$$\text{qRAG}\,|i, b, x_1, \ldots, x_n\rangle = |i, x_i, x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n\rangle$$

We also restrict our circuits to only wire qRAG's so that they use the work bits for addressing the memory and the memory bits as the addressed qubit, to prevent our circuits to use qRAG with permuted inputs of the work and memory registers.

A quantum circuit for an instance input of size n is therefore a sequence of elementary gates and qRAG's applied to a work register of $\log(n)$ qubits and an unrestricted amount of memory registers of size n. We read the circuit output by performing a measurement in the computational basis on the first qubit of the work register.

Since measurements are non-deterministic, they are also irreversible and can therefore not be represented by a unitary operation. We do actually want our circuit output to be observable, so we want measurements to be part of our algorithm in some way. We can incorporate measurements in the definition of circuit decidability as opposed to letting them be part of our actual circuits. This lets us ensure that our circuits stay deterministic and reversible, which then lets us represent them as unitary operations. Some quantum algorithms may want to use measurements *during* the algorithm, as opposed at the end. The deferred measurement principle tells us that by conditioning quantum gates on measurement outcomes we can always defer measurements to the end of our algorithm [40].

We then have the following definition for quantum qRAG circuit decidability.

Definition 2.2.5. Let $L \subseteq \{0,1\}^*$ be a language.

- A family of quantum circuits with qRAG's $\{C_n\}_{n\in\mathbb{N}}$ decides the language L with probability $\frac{2}{3}$ in time $T(n)$ if C_n has circuit complexity $T(n)$

[1] Quantum random-access memory has previously been discussed in works such as [28] and [38].

and for every $x \in \{0,1\}^*$ the following hold: If $x \in L$, then the probability of observing a 1 after a measurement of the first qubit of $C_{|x|}(x)$ in the computational basis is at least $\frac{2}{3}$. If $x \notin L$, then the probability of observing a 0 after a measurement of the first qubit of $C_{|x|}(x)$ in the computational basis is at least $\frac{2}{3}$.

- A family of quantum circuits with qRAG's $\{C_n\}_{n\in\mathbb{N}}$ verifies the language L with probability $\frac{2}{3}$ in time $T(n)$ if C_n has circuit complexity $T(n)$ and for every $x \in \{0,1\}^*$ the following hold: If $x \in L$ there exists $u \in \{0,1\}^{O(|x|)}$ such the probability of observing a 1 after a measurement of the first qubit of $Q(x,u)$ in the computational basis is at least $\frac{2}{3}$. If $x \notin L$ it holds for all $u \in \{0,1\}^{O(|x|)}$ that the probability of observing a 0 after a measurement of the first qubit of $Q(x,u)$ in the computational is at least $\frac{2}{3}$.

With our preferred quantum model at hand, we can move on to the natural analogues of BPP and MA discussed in the previous section.

2.2.1 Bounded-Error Quantum Time

Before showing where quantum circuits potentially outperform classical circuits, we need to consider the complexity classes naturally defined in quantum computing. The classes of problems solved by deterministic quantum algorithm are not very interesting since they make it hard to work with quantum measurements. We will therefore directly skip ahead to the bounded-error quantum classes. First we need to say a bit more about measurements.

Quantum measurements are performed relative to a basis spanning the Hilbert space of the measured state. Since we generally want to observe bits or bit-strings by the end of our algorithm, we perform measurements in the computational basis.

Theorem 2.2.1 (Born's Rule)**.** *The probability of observing the state* $|x\rangle$ *with* $x \in \{0,1\}^n$ *after performing a measurement in the computational basis on the state* $|\psi\rangle$ *in Hilbert space* $\mathbb{R}^{\otimes 2^n}$ *is given by* $|\langle x|\psi\rangle|^2$*. If* $|\psi\rangle = \sum_{i\in\{0,1\}^n} \alpha_i |i\rangle$ *it follows that* $|\langle x|\psi\rangle|^2 = \alpha_x^2$*.*

Simply said, the probability of observing a string i is given by the squared norm of the amplitude of the state $|i\rangle$ in the measured state $|\psi\rangle$.

Due to the probabilistic nature of measurements, the natural quantum classes of efficiently computable and efficiently verifiable decidable languages are probabilistic.

Definition 2.2.6 (BQP and QMA)**.** Suppose $L \subseteq \{0,1\}^*$.

- A language L is in BQP if there is a polynomial time quantum algorithm that decides it with probability $\frac{2}{3}$.
- A language L is in QMA if there is a polynomial time quantum verifier algorithm that verifies L with probability $\frac{2}{3}$.

Here we use a verifier definition of QMA similar to the one used in for NP. Again, we can think of Quantum Merlin-Arthur QMA as the class of problems decidable by polynomial time non-deterministic quantum algorithms.

2.2.2 Quantum Queries and Oracles

We will be working with quantum reductions in this thesis and our quantum algorithms can therefore have access to quantum oracles. A quantum oracle works similar to a classical oracle in that we query a problem instance to an oracle in order to retrieve some type of function or algorithm output for that instance. The main difference is that since our oracles are also quantum objects, we can query superpositions of instances, similarly to how we let qRAG's query superpositions of memory addresses. Formally, we define oracle queries as follows.

Definition 2.2.7 (Quantum Oracle)**.** A quantum oracle to the function $f : \{0,1\}^n \to \{0,1\}$ is a unitary O_f such that

$$O_f |x, b\rangle = |x, (f(x) + b) \bmod 2\rangle$$

for each $x \in \{0,1\}^n$.

In many algorithms we use oracles as placeholders for actual algorithms that can be simulated by our computational model. Because the algorithms that we use to simulate our oracles end up being quantum algorithms they will naturally have some error probability. In order for us to use oracles in a useful manner, we assign a success probability to our quantum oracles. The bounded-error of our oracles will be the same as those in our BQP and QMA complexity classes, and we assume that oracles are successful with probability $\frac{2}{3}$.

We could also introduce uncertainty for our oracles in the classical setting, since we do work with randomized hardness conjectures. However, we conjecture the same hardness for our three popular problems for randomized classical algorithms as we do for deterministic classical algorithms. As a result, lower bounds conditioned on these hardness conjectures will generally be the same as they would be for the deterministic hardness conjectures. On top of that, all the problems that we use classical oracles to have deterministic matching upper bounds, so the extra baggage of introducing randomness in our classical oracles is not necessary.

Due to the useful interaction between quantum algorithms and queries, we often measure quantum algorithms in their query complexity as opposed to their time complexity; how many queries to a given input does an algorithm require to solve a problem? The time complexity is, of course, lower bounded by the query complexity.

The query complexity is also useful when using quantum algorithms as subroutines of our main algorithms. In RAM models, querying the input requires constant time, but if we apply our quantum algorithms as sub routines, we may be querying a different mathematical object. If the queried property can be computed within the main algorithm, we don't need to explicitly construct the full mathematical object but can simply compute the properties, potentially in superposition, as they are queried. The queries then come at a cost of the time that is required to compute the property. We will be making use of this strategy in Chapter 4 and will discuss it more in-depth there.

For classical algorithms it is more conventional to measure the complexity of an algorithm by its time complexity and since we are comparing the effectiveness of quantum computers to the effectiveness of classical computers, we will be talking about quantum time complexity in this thesis.

2.2.3 Quantum Speed-Ups

Now that we've provided definitions of the common quantum complexity classes and our quantum model, we can actually talk about what it means for a quantum algorithm to outperform a classical algorithm. In the comparison we compare quantum circuits with qRAG's to classical randomized RAM's. If we find a quantum circuit that solves a problem faster than the fastest known randomized RAM, we say we have a quantum speed-up. One of the most important quantum speed-ups is found through the Grover Search algorithm and as it turns out, a lot of the quantum speed-ups found in more complicated computational problems are a result of applications of Grover's algorithm as sub-routines. Such speed-ups are called Grover type speed-ups.

Consider a string $x \in \{0,1\}^n$ and suppose we have query access to the string, i.e., we can access the value of the i'th bit in one time-step. Now suppose we want to ask if there is at least one index i for which $x_i = 1$. A classical RAM would need time $O(n)$ to solve this problem, since in the worst case only the last queried index is set to 1. Quantumly we can do this in $O(\sqrt{n})$ time by Grover's search. We don't go into the details of the algorithm here, but the main idea is that we can query every index of the string x in superposition and then use state interference effects to increase the amplitude of indices that are set to 1 by applying controlled-phase gates. We end the algorithm by

performing a measurement in the computational basis and query the observed index to see whether it is a solution [29].

Theorem 2.2.2 (Grover Search)**.** *Suppose $x \in \{0,1\}^n$ and we have query access to the different bits x_i of x. There exists a quantum algorithm such that the following hold: If there is some $i \in \{0,1\}^{\log n}$ such that $x_i = 1$, then the algorithm will output some $j \in \{0,1\}^{\log n}$ such that $x_j = 1$ with probability $\frac{2}{3}$. If there is no such $i \in \{0,1\}^{\log n}$, then the algorithm will output 0 with probability $\frac{2}{3}$. The algorithm takes $O(\sqrt{n})$ time and makes $O(\sqrt{n})$ queries to x.*

The type of search problem presented in the above theorem comes in many shapes. Essentially, if we have a set of elements X and we want to know if X contains some kind of marked element; an element with a property that we can compute, we can apply Grover Search to that set. Since Grover Search requires $O(\sqrt{|X|})$ queries, finding the marked element will take time $O(\sqrt{|X|}T)$, if T is the time required time to compute the desired property. In this scenario we can view T as the cost of making a query to our search space.

To make this idea more concrete, we can consider the satisfiability problem. Given a propositional logic formula ϕ over n literals, we have a search space of size 2^n. To find out whether there is a satisfying assignment to ϕ would classically take time 2^n times the time it takes to verify whether a given variable assignment satisfies the formula. Since verifying a formula takes polynomial time, it takes time $\tilde{O}(2^n)$ to solve the problem. Using Grover Search we can solve the problem by querying the truth table of ϕ a number of $O(\sqrt{2^n})$ times. Each query takes polynomial time, and we find a quantum complexity of $\tilde{O}(2^{\frac{n}{2}})$.

In the APSP problem we are trying to look for shortest paths and the minimization operation will therefore be an operation naturally occurring in this work. We provide here also a Grover Minimum Finding algorithm that will be useful later on. The algorithm uses a generalization of the Grover Search algorithm and was first shown in [24].

Theorem 2.2.3 (Grover Minimum Finding)**.** *Let S be an unsorted array of elements with a linear order and suppose that $|S| = n$. There exists a quantum algorithm that outputs the minimal element of S with probability at least $\frac{2}{3}$, taking $O(\sqrt{n})$ time and making $O(\sqrt{n})$ queries to S.*

In Grover Search, we assume query access to the values of the set where we want to find a marked element. In some cases like the satisfiability example it is not so obvious whether we have that type of access, as we need to compute the queried value. In the satisfiability example this posed no problem, since

we could always compute the queried value in polynomial time, and we were searching an exponentially sized space.

A variation of Grover's algorithm found in [6] allows us to find the complexity of Grover search when computing the properties of elements in our search space has a variable time, e.g., when the elements are of different size. Consider for example a set of graphs that each have a varying amount of nodes. If we want to find whether this set contains a graph with some desired property, it may take us a different amount of time to compute the property for each graph. For such a case, we have the following theorem:

Theorem 2.2.4 (Variable Time Grover Search)**.** *Let $x \in \{0,1\}^n$ and suppose we can compute the value of x_i in time t_i. There exists a quantum algorithm such that the following hold: If there is some $i \in \{0,1\}^{\log n}$ such that $x_i = 1$, then the algorithm will output some $j \in \{0,1\}^{\log n}$ such that $x_j = 1$ with probability $\frac{2}{3}$. If there is no such $i \in \{0,1\}^{\log n}$, then the algorithm will output 0 with probability $\frac{2}{3}$. The algorithm takes $O(\sqrt{\sum_{i=1}^{n} t_i^2})$ time and makes $O(\sqrt{\sum_{i=1}^{n} t_i^2})$ queries to x.*

We refer to t_i in the above theorem as the 'checking time' for each element in the search space and abbreviate Variable Time Grover Search VTGS.

The algorithms from the theorems above are all polynomial speed-ups; the problems posed could already be solved in polynomial time by a randomized algorithm, but can now simply be solved in faster polynomial time. It would be very interesting to know whether there exists an algorithm that can solve an NP-complete problem in polynomial time, as this would imply that $\mathsf{NP} \subseteq \mathsf{BQP}$. While so far no such algorithm has been found, and it is unlikely that such an algorithm exists, we do have quantum speed-ups that are exponential: problems for which no polynomial time algorithm has been found that can be solved in polynomial time by a quantum computer. The main such example is Shor's algorithm for integer factorization. The best classical algorithm runs in sub-exponential time, roughly $\tilde{O}(2^{(\log n)^{\frac{1}{2}}})$, where n is the size of the composite number [12]. Shor's algorithm finds the prime factors of n in time roughly $O((\log n)^2)$[45]. Unfortunately integer factorization is not known to be NP-hard so even though our speed-up is exponential, there are no consequences for whether $\mathsf{NP} \subseteq \mathsf{BQP}$ holds.

2.3 Overview

A lot of our speed-ups in the quantum setting will be as a result of Grover Search, which allows for a polynomial speed-up. Combined with the fact that a lot of the lower and upper bounds in this work fall in the polynomial regime,

it is important that our machines have random-access memory, both in the quantum case and in the classical case.

Furthermore, we would do ourselves short leaving out the capacity of randomness in our quantum algorithms and since we want to make sure that our comparative analysis focuses on classical versus quantum effective differences, we incorporate randomness in our classical algorithms as well.

As a result, in the classical setting we end up with randomized RAM's to model our algorithms and in the remainder of this thesis whenever we mention what randomized algorithms are capable of, imagine them being executed on a randomized RAM.

Quantum random-access memory is not as conventional as classical random-access memory, but it makes for a natural extension of our classical model to also include the capacity for random-access memory in the quantum setting. As a result, we find quantum circuits with qRAG's to be the appropriate model for our quantum algorithms and whenever we mention quantum algorithms in the remainder of this thesis, assume that they can be executed on quantum circuits with qRAG's.

We defined the RAM somewhat loosely, and although we had a more rigorous mathematical definition of our quantum algorithms, using the Hilbert space formulation of quantum mechanics, in practice we will take liberties in describing our quantum algorithm to the same extent that we do classically. Large parts of our quantum algorithms will run the same as they would classically and as such, we describe them classically assuming that the algorithm can easily be executed on a quantum circuits with qRAG's. We apply the actual quantum parts of our algorithms, like Grover Search and VTGS, as subroutines. We do this by specifying the search set, the properties that mark elements and the time required to compute them. We can then find the complexity of our algorithms by combining the classical complexity for the classical part of our algorithm with the complexity stipulated by our quantum 'black boxes'.

Chapter 3

Known Fine-Grained Reductions

Fine-grained computational complexity started receiving notoriety over the past ten years as it's given us a very clear overview of comparable hardness between many computational problems. It is no surprise then that, just as many other classical frameworks, we want to see how these results hold up in the quantum case. In this chapter we define fine-grained reductions and elaborate on the classical framework. We also review classical reductions from three different popular problems and their associated popular hardness conjectures: k-SAT, 3SUM and APSP. In the end, we find two graph problems, Δ-Matching Triangles and Triangle Collection to which the three popular problems reduce. This result was found in [3] and allows us to prove the hardness of Δ-Matching Triangles and Triangle Collection based on the disjunction of the hardness conjectures for k-SAT, 3SUM and APSP. Since the hardness conjecture for each of these three problems is popular, conjecturing that at least one of the three problems is hard is extremely popular.

The goal is to repeat the hardness result based on an extremely popular conjecture in Chapter 4 for the quantum setting and formulate an extremely popular quantum conjecture. The relevant quantum reductions from 3SUM have already been reviewed in [13] and so the reductions that will be investigated in this chapter will be related to SAT, k-SAT and APSP and are taken from [3, 25, 37, 54, 53]. For the classical reductions from 3SUM we refer to [41] and [54].

In the first section we go over the required definitions and the ideas that helped develop them. In the second section there will be some graph related definitions and an overview of the reductions from APSP. In the third

section we discuss the reductions from SAT and k-SAT. We close off the chapter with an overview of all the reductions and proving the hardness of Δ-Matching Triangles and Triangle Collection based on extremely popular conjecture.

3.1 The Fine-Grained Model

Fine-grained reductions take the ideas from reducibility and hardness and re*fine* them, allowing for even more structure in the different levels of computational complexity to emerge. The concept of fine-grained reductions already existed for a while, with conditional hardness based on the implicit hardness of our popular problems, such as the quadratic hardness of many problems based on the hardness of 3SUM in [27]. This notion was further developed with the introduction of SETH in [33] and the exponential hardness of many NP problems based on SETH in [18].

Later on this framework was extended and formalized with the inclusion of hardness conjectures for 3SUM and APSP. An overview of many fine-grained reductions found so far can be studied in [51] and [52].

Many of the conditional lower bounds proven so far use deterministic algorithms and the hardness for our popular problems is in those cases conjectured for deterministic algorithms. However, as can be seen in the well-regarded overview in [52], it is not unconventional to conjecture the same hardness for the popular problems for randomized algorithms. Furthermore, important results such as the fine-grained reduction from 3SUM to Convolution-3Sum already made use of randomized reductions, relying on the quadratic hardness of 3SUM for randomized algorithms. We will extend the classical work reviewed in this chapter to the quantum framework and will therefore conjecture classical hardness of the popular problems for randomized algorithms.

3.1.1 Coarse-Grained Reductions

We've talked about the P versus NP problem in Chapter 2 and how the NP-completeness of k-SAT together with SETH makes a conditional distinction between polynomial-time problems and exponential-time problems. The hardness conjecture, in combination with polynomial-time reductions, allows us to structure different levels of computational complexity. Although the structure is based on conjecture, and therefore in a way completely hypothetical, the value of finding this structure is not diminished and does not depend on the validity of SETH. Regardless of the validity of SETH, working with such a hardness conjecture allows us to show that comparable hardness of other problems: SAT is equivalent in hardness as many other problems in NP. An important aspect of why this reasoning through reductions works, lies in the

definition of Turing reductions from the previous chapter; namely that they have to be polynomial time. If we'd let our reduction take more time, say exponential, having a polynomial time algorithm for a problem like k-SAT would not mean as much. If we had an efficient algorithm for k-SAT, we could not use exponential-time reductions to solve other NP problems in polynomial time.

Requiring that reductions take polynomial-time is the weakest time-requirement that lets us use reductions to show that a problem is in P by reducing it to other problems in P or showing that a problem is NP complete by reducing another NP-complete problem to it. To sum up, we can use them to show that problems can be solved in polynomial or *probably not* in polynomial time. It doesn't let us compare differences in complexity between different problems in P, for this, the polynomial-time requirement for the reductions is too coarse. We therefore refer to them as coarse-grained reductions.

3.1.2 Fine-Grained Reductions

Clearly the definition for coarse-grained reductions is too free to let us prove the hardness of problems up to precise asymptotic complexity. We refine the definition as follows:

Definition 3.1.1 (Fine-Grained Reductions)**.** Let $l(n)$ and $l'(n)$ be two non-decreasing functions of n. For two problems $L, L' \subseteq \{0,1\}^*$, we say that L is (l, l')-fine-grained reducible to L', denoted $L \leq_{l,l'} L$ if there exists a randomized algorithm $R^{L'}$ with access to oracle L' that solves L with probability at least $\frac{2}{3}$ such that for every $\epsilon > 0$ there exists $\delta > 0$ and:

- The algorithm $R^{L'}$ runs in time $O(l(n))^{1-\delta}$.
- For the sequence of instances $n_1, ..., n_k$ that $R^{L'}$ queries to oracle L' it holds that $\sum_{i=1}^{k} l'(n_i)^{1-\epsilon} = O(l(n))^{1-\delta}$.

Whenever $L \leq_{l,l'} L'$ and $L' \leq_{l',l} L$, then L and L' are fine-grained equivalent, denoted $L \equiv_{l,l'} L'$.

The above definition should make it clear why we started from Turing reductions as opposed to Karp-reductions in Chapter 2: We want to allow our algorithms to make multiple queries to the problem that we want to reduce to, as opposed to just outputting a single problem instance.

If we can show this type of fine-grained reducibility between two problems we can use the oracle machine from the above definition to solve L using a *simulated* oracle to L'. Suppose we have a $l'(n)^{1-\epsilon}$ time algorithm to L', we can then use the fine-grained reduction to solve L in $l(n)^{1-\delta}$ time. If we conjecture a lower bound in complexity to solving L' equal to

our best algorithm for L', finding a faster algorithm for solving L' would then also imply a smaller lower-bound to L through the fine-grained reduction.

The function $l(n)$ represents the known or conjectured lower bound for the problem that we are reducing, L. The bound $l'(n)$ is the lower bound we prove for the problem L', conditioned on the $l(n)$-hardness of problem L.

The first bullet-point in the definition for fine-grained reduction ensures that the algorithm takes no more than $O(l(n)^{1-\delta})$ time. Clearly this requirement is needed if we want to solve A using an oracle for L': if our reduction exceeds the run- time $l(n)^{1-\delta}$, finding a $l'(n)^{1-\epsilon}$ time algorithm for L' would not contradict the $l(n)$-hardness for L.

Lastly, we need to limit the use of the oracle L' by our algorithm. Solving the problem L through reduction would only work if we don't make too many queries to L' or our queried instances are small. If we'd find a $l'(n)^{1-\epsilon}$ time algorithm for L' but all the combined calls to this algorithm exceed the run time $l(n)^{1-\delta}$, we again don't contradict the $l(n)$-hardness for L. This requirement is set in the second bullet point.

Ideally we'd be able to use a single conjecture to show conditional lower bounds for many other problems through fine-grained reductions as we do in the coarse-grained case through SETH. Unfortunately we don't have this luxury, but with conjecturing hardness for just three popular problems we can already prove hardness of many computational problems.

3.1.3 Popular Hardness Conjectures

The three main popular problems for which hardness has been conjectured and from which fine-grained reductions have been found are k-SAT, 3SUM and APSP. For all three problems the hardness conjectures are natural in a way that no faster algorithm has been found in a long time or is ever expected to be found. Like mentioned in the first paragraph of this chapter, it is not really that important whether the conjecture is valid or not; fine-grained reductions let us know in any case that certain problems are at least as hard as others. The work done in the classical case on each of these three conjectures is extensive. We define the three problems in this section, starting with k-SAT.

The satisfiability problem is usually defined for a specific class of propositional-logic formulas: Conjunctive Normal Form (CNF) formulas.

Definition 3.1.2 (SAT and k-SAT)**.** Let ϕ be a propositional logic formula. We say that ϕ is CNF if it is a conjunction of clauses, $\phi := c_1 \wedge \cdots \wedge c_m$ for some $m > 0$. A clause is a disjunction of literals and a literal is a propositional variable or its negation.

- A formula ϕ is in SAT if it is CNF and there is a satisfying assignment to ϕ.
- A formula ϕ is in k-SAT if ϕ is CNF, every clause in ϕ is a disjunction of at most k literals and there is a satisfying assignment to ϕ.

For the satisfiability problem we assume that logical formulas are CNF since it makes them easier to work with. Furthermore, it can be shown that any logical formula can be translated in polynomial time into a CNF formula of the same asymptotic size that is equi-satisfiable [44, 47].

We often mention k-SAT and SAT separately when talking about hardness of satisfiability. The reason for this is that the two popular hardness conjectures for satisfiability are both formulated for k-SAT, and we can infer hardness for SAT from them. Any algorithm for SAT is also an algorithm for k-SAT and it is therefore easy to reduce k-SAT to SAT.

There are two main hardness conjectures for k-SAT, each of which can be stated in different equivalent ways. We chose common formulations that will be useful to our work. While the conjectures were first formulated in [33] for deterministic algorithms, it is not uncommon these days to extend the hardness to randomized algorithms [21, 52].

Conjecture 3.1.1 (Exponential-Time Hypothesis)**.** *For $k \geq 3$, let δ_k be the infimum of the set of constants c for which there exists a randomized algorithm that solves k-SAT in $O(m2^{cn})$ time, where n is the number of variables and m the number of clauses of the k-SAT instance. Then it holds that $\delta_k > 0$, for $k \geq 3$.*

From ETH it follows that for any $k \geq 3$ it holds that there is some δ_k such that k-SAT cannot be solved in $O(m2^{\delta_k})$ time. Since the sequence $\delta_3 \dots \delta_i \dots$ is non-decreasing, this implies that SAT cannot be solved in time $O(m2^{n\delta_3})$ and we find hardness for SAT.

Unfortunately ETH is not strong enough for proving conditional lower bounds based on SAT or k-SAT. We have the following stronger version:

Conjecture 3.1.2 (Strong Exponential-Time Hypothesis)**.** *For all $\epsilon > 0$ there exists $k \geq 3$ such that k-SAT cannot be solved in time $O(m2^{n(1-\epsilon)})$ by a randomized algorithm, where n is the number of variables and m the number of clauses of the k-SAT instance.*

If we were to have an $O(m2^{n(1-\epsilon)})$ time algorithm for SAT for some $\epsilon > 0$, then k-SAT for any k could be solved in $O(m2^{n(1-\epsilon)})$ time and SETH would be violated. As a result, we have that under SETH, SAT requires $O(m2^{n-o(1)})$ time.

The Exponential-Time Hypothesis (ETH) follows from the Strong Exponential-Time Hypothesis (SETH) by use of the sparsification lemma from [34] which is discussed towards the end of this chapter. Whether SETH is strictly stronger is unknown, it is possible that an implication from ETH to SETH can be found.

For the lower bounds in fine-grained complexity based on k-SAT we need to assume SETH and for lower bounds on SAT we need to assume the possibly weaker requirement that SAT requires at least $O(m2^{n-o(1)})$ time, which is why we make a clear distinction between hardness for the two problems.

Lastly, to sketch why ETH may not be strong enough, consider the following reduction scenario: Suppose we fine-grained reduce SAT to some polynomial time problem L, proving a conditional lower bound $O(N^c)$ to L in L's size parameter N and for some $c > 0$. Furthermore, assume that on a SAT input over n variables and m clauses our reduction makes $O(1)$ queries of instance size $O(2^{\frac{n}{c}})$ to oracle A, such that $O((2^{\frac{n}{c}})^c) = O(2^n)$. Finding a $O(N^{c-\epsilon})$ algorithm, with $\epsilon > 0$, for problem L would then result in a $O((2^{\frac{n}{c}})^{c-\epsilon}) = O(2^{n(1-\frac{\epsilon}{c})})$ time algorithm for SAT, which contradicts SETH but not ETH.

We can easily solve SAT in the time conjectured by SETH through simple brute force search and trying all possible variable assignments. For k-SAT there are faster than $O(m2^n)$ algorithms for all values of k as described in [42].

The next popular problem is the 3SUM problem.

Definition 3.1.3 (3SUM)**.** Given a set of integers S of size n from range $[-n^c, n^c]$, determine whether there exist three integers $x, y, z \in S$ such that $x + y = z$.

Naively we could solve 3SUM in n^3 time by computing the sum of every pair of numbers and searching S to determine whether S contains that sum.

We can solve 3SUM in n^2 by using a traversal. First we sort the set of integers in time $O(n \log(n))$. We then proceed by having an outer loop iterate over the integers k in ascending order and having a second loop traversing the order. This means that in the second loop, we take two integers i, j, starting with i as the lowest integer and j as the highest integer. We then compare their sum to the integer k from the outer loop.

If $k = -(i + j)$, we have found our sum and the algorithm terminates. If $k > -(i + j)$ then the traversal loop sets i to the next integer in ascending order. If $k < -(i + j)$ then the traversal loop sets j to the next integer in descending order.

Clearly if there are no integers that sum to 0 the algorithm will output a negative answer correctly. If there are three such integers, we know that

whenever our outer loop arrives at one of them, the traversal loop will find the other two. Since the traversal loop iterates over $O(n)$ integers, the algorithm takes $O(n^2)$ time.

Work has been done to improve the run-time of 3SUM and the best randomized algorithm solves it in $O(n^2 \frac{\log(\log(n))}{\log^2(n)})$ time[9]. It is an open question whether there exists a $O(n^{2-\epsilon})$ randomized algorithm for 3SUM for any $\epsilon > 0$ and it is therefore natural to conjecture the following.

Conjecture 3.1.3 (3SUM-hardness)**.** *There exists no randomized algorithm for* 3SUM *that solves it in time* $O(n^{2-\epsilon})$ *for any* $\epsilon > 0$.

We keep the discussion of 3SUM brief since we won't be reviewing direct reductions from 3SUM in this work.

The final popular problem for which we have a popular hardness conjecture is APSP.

Definition 3.1.4 (APSP)**.** Let $G = (V, E)$ be a weighted directed graph over n nodes with no negative cycles and with integer weights given by weight function $w : E \to [-n^c, n^c]$ for some constant c. For a pair of vertices, a sequence of connecting edges $p(a, b) := ((a, i_1); (i_1, i_2); \ldots; (i_{k-1}, b))$ is called a path over k edges from a to b. The length of a path $p(a, b)$ is $l(a, b) = \sum_{j=1}^{k} w(p(a, b)_j)$. An algorithm solves APSP if it outputs for every pair of vertices (a, b) the length of the shortest path from a to b and outputs ∞ if there is no path for a pair of nodes.

There are many versions of APSP. A common formulation of the problem requires an algorithm to output not just the length of the shortest paths, but the paths themselves. The popular conjecture in fine-grained literature is for the above distance formulation of the problem and many of the reductions work for the shortest distance as opposed to the shorted paths version. We can, of course, easily solve the shortest distance version by reading the shortest paths version, providing us with a reduction from the distance version to the paths version.

Other versions of the problem can be found by restricting the problem to a subclass of graphs such as APSP on undirected graphs, unweighted graphs, dense graphs, sparse graphs, etc. Common graph definitions that appear throughout the thesis can be found in Appendix A. In the above definition of APSP we have made one restriction: Our input graph has no negative cycles; paths from a node back to the original node that have a negative length. If there is a negative cycle in a graph, then any path from node a to node b that crosses a node on the negative cycle will have shortest distance $-\infty$, since our path can simply walk over the cycle an infinite number of times. This is problematic for our algorithms, since they need to terminate

in finite time and have a finite output. There are three ways of fixing this problem: We could reformulate our definition and force paths to be *simple*, meaning that they can only cross every node at most once. Second, we can have an algorithm that first detects whether the input contains a negative cycle and halts if it does. Or, lastly, we can simply restrict our input as we did in the current definition. The first option unfortunately increases the complexity of the problem [31, 56] and while the second option is acceptable, it does complicate our algorithms and reductions. We settle for the third.

We haven't specified a specific data structure of the input and output of the problem and algorithm and graph problems are usually presented in one of two different ways. A graph can be presented by either an adjacency list or an adjacency matrix. For our work we take the adjacency matrix representation and extend it to a weight matrix. We assume that our algorithms have random access to the nodes, edges and the edges weights, with the edge weights being encoded as entries of the weight matrix. The absence of an edge between two nodes is marked as an ∞ in the weight matrix.

The output for APSP is usually given in the form of a distance matrix D of the graph. For any pair of nodes i, j we have that the entry $D[ij]$ denotes exactly the length of the shortest path from i to j. If there is no path from i to j then $D[ij] = \infty$.

We work with ∞ in the problem formulation and our description of the weight and distance matrices which may present a problem since we would need infinite bits to store such a value. In practice, we can use any value larger than $n \cdot n^c$ instead. Since the longest shortest path in any graph has at most length n^{c+1}, we know that if we find a value in our distance or weight matrix that exceeds n^{c+1}, there is no edge or path between the nodes corresponding to the entry address. For ease of notation we will keep using ∞ in this work.

We won't provide matching upper bounds for most classical lower bounds presented in this chapter, as that is not the focus of this thesis. For arguing a conjectured classical, and later quantum, lower bound to APSP, it is however important to show an algorithm for solving APSP. Solving the distance version of APSP is rather straight-forward and is closely related to the matrix distance product and the fine-grained reduction from APSP to the distance product that will be discussed in the next section. Classically, there is no $O(n^{3-\epsilon})$ time algorithm for any $\epsilon > 0$, for both versions of APSP and since we will be the first to formally define a quantum hardness conjecture for APSP, it will be important to consider hardness for both versions of the problem. As such, we present here also a classical algorithm for solving the path version of APSP. The most popular classical algorithms for the path version of APSP are the Floyd-Warshall, Bellman-Ford and Johnson algorithms [26, 10, 35].

Johnson's algorithm is based on the single-source shortest paths algorithm by Dijkstra [22], which works for graphs with no negative weights and finds the shortest path from one node to all other nodes in $O(n^2)$ time. Johnson's algorithm uses a graph re-weighting trick from the Bellman-Ford algorithm to make all edges non-negative and then applies Dijkstra's algorithm to all nodes in the graph. Since we support our quantum hardness for APSP in Chapter 4 with a quantum single-source shortest paths algorithm, we provide Johnson's algorithm here.

Theorem 3.1.1 (Dijkstra's Algorithm)**.** *There exists an algorithm that, given a weighted directed graph $G = (V, E)$ over n nodes, weight matrix W with weights from $[0, n^c]$ and input node a outputs the shortest paths between a and all other nodes in G in time $O(n^2)$.*

Proof. Let $G = (V, E)$ be a weighted directed graph over n nodes with weights from $[0, n^c]$ and suppose we want to find all shortest paths from node a. We output our solution as a shortest path tree; a tree with a as its root node and if there is a path to another node b in our original graph, the unique path from a to b in the shortest path tree will be the shortest path from a to b in G. We enumerate the nodes in G and while the algorithm runs, we also keep track of an n-dimensional tentative distance vector d, where we keep track of all shortest distances from a to the other nodes in G. To start, we set the distance form a to a to 0 and all other values in d to ∞.

The algorithm proceeds as follows: Start by setting a as the *current* node. From the current node, compare the tentative distance from a to the current node plus the distance from the current node to one of its unvisited neighbors b and compare that distance to the distance from a to b stored in d. Update d so it contains the smallest of the two distances:

$$d[b] = \min(d[current] + w(current, b), d[b]).$$

Once all unvisited neighbors of the current node have been considered in this way, mark the current node as visited. The next current node will be the unvisited node that has the smallest tentative distance from a stored in d. Every time we move to a new current node, we can add it to our shortest path tree along the tentative shortest path from a in G.

We claim that whenever we move to a new current node b, the tentative shortest path from a to b is the shortest path from a to b. By induction on the number of visited nodes: It is trivially true for the base case from a to a. Assume we have visited $i - 1$ nodes, and we added each of them with their shortest path from a to our shortest path tree. Suppose our next current node is b. If there was a shorter path that crosses unvisited nodes from a

to b than the tentative path and the first unvisited node on that path is c, then the tentative distance from a to c would be smaller than the tentative distance from a to b, which is a contradiction. If there is a shorter path from a to b that crosses only visited nodes and the last node on that path before reaching b is c, then the tentative distance from a to b would be the tentative distance from a to c plus the distance from b to c, which is also a contradiction. It must be that the i'th visited node will be correctly added to our shortest path tree.

The algorithm terminates when every node has been visited or all the unvisited nodes have tentative distance ∞. Since each node will have $O(n)$ neighbors, the run time is $O(n^2)$. □

Dijkstra's algorithm works because the edges are all non-negative, so by iteratively adding the unvisited node with the least tentative distance to our shortest path tree, we know we will always be adding shortest paths. Johnson's algorithm uses a reweighing trick involving the Bellman-Ford algorithm to construct a graph with non-negative edges from the input graph that is shortest-path-equivalent. We can then apply Dijkstra's algorithm to each of the nodes in the non-negative graph.

Theorem 3.1.2 (Johnson's Algorithm)**.** *There exists an algorithm that solves* APSP *on weighted directed graphs $G = (V, E)$ over n nodes with weights from $[-n^c, n^c]$ in $O(n^3)$ time.*

Proof. Let $G = (V, E)$ be a weighted directed graph over n nodes with weights from $[-n^c, n^c]$ and no negative cycles.

First we enumerate all nodes in G and add a new node z to G that is connected to all other nodes by a 0-weight edge. We can, somewhat inefficiently in time $O(n^3)$, compute all shortest path lengths from node a using the Bellman-Ford algorithm as follows: We start with the same tentative-distance vector d from Dijkstra's algorithm and iterate n times, constantly relaxing the tentative distances. In iteration k loop over all edges (i, j) and check whether $d[i] + w(i, j) < d[j]$. If the check returns true, update the distance $d[j]$ to $d[i] + w(i, j)$.

We claim that after k updates, $d[a]$ will be ∞ if there is no path over k edges from z to a and $d[a]$ will be the length of the shortest path over k edges otherwise. By induction:

The base case where $k = 0$ is straight forward. Assume that for $k = i - 1$ it holds that $d[a]$ will be ∞ if there is no path over $i - 1$ edges from z to a and $d[a]$ will be the length of the shortest path over $i - 1$ edges otherwise. If there is no path over i edges from z to a, then there is no path over $i - 1$ edges reaching any of the neighbors of a and $d[a] = \infty$. Otherwise, assume that b is

the second to last node on the shortest path over i edges from z to a. By the induction hypothesis the shortest paths from z to all of a's neighbors will be stored in d and as a result, our update rule will correctly select the shortest path going through b.

Since there are no negative cycles in G, after n iterations d will contain the shortest distances from z.

We will now replace all edge-weights $w(i,j)$ in G by $w(i,j) + d[i] - d[j]$ and remove the node z to construct the graph G'. We then make the following two claims: G' contains only non-negative edge weights and the shortest path from a to b in G is the shortest path from a to b in G':

Let $p'(a,b)$ be a path of length k in G' from a to b and let $p(a,b)$ be the path crossing the same edges but in G. The length of the path $p'(a,b)$ will be:

$$\sum_i^k p'(a,b)_i = (w(a,i_1) + d[a] - d[i_1]) + (w(i_1,i_2) + d[i_1] - d[i_2]) + \cdots + (w(i_k,b) + d[i_k] - d[b])$$

$$= \sum_i^k p(a,b)_i + d[a] - d[b]$$

We see that any path from a to b in G' has the same value $d[a] - d[b]$ added to it and it follows that any shortest path in G will be the shortest path in G'. All the paths from z to other nodes a in the graph G would have the value $d[z] - d[a] = -d[a]$ added to them and as a result would have distance 0 from z. There can therefore be no negative edges in G': if there was a negative edge (i,j) in G' then the path going from z to i to j would have negative length in G, contradicting the previous statement.

It follows that G' has only non-negative edges and has the same shortest paths as G. We apply Dijkstra's algorithm to all nodes in G' to find all shortest paths in G in $O(n^3)$ time. □

The Bellman-Ford algorithm used in Johnson's algorithm finds single source shortest paths just like Dijkstra's algorithm does but works on graphs with negative edges. Unfortunately, Bellman-Ford has a worse run time of $O(n^3)$ for dense graphs, and we can therefore not use it on every node to solve APSP in $O(n^3)$ time. Johnson's algorithm cleverly combines the two algorithms to solve APSP in $O(n^3)$ time.

For unrestricted APSP, Johnson's algorithm is not necessarily the fastest algorithm out there but is rather well-known and intuitive. The best randomized algorithm to date solves APSP in $O(\frac{n^3}{e^{\sqrt{\log(n)}}})$ time and no truly faster than cubic algorithm for APSP has been found in many years [52]. We can

easily read the shortest distances from the shortest paths in $O(n^3)$ time and solving the shortest paths version is therefore at least as hard. We have the following conjecture.

Conjecture 3.1.4 (APSP-hardness)**.** *There exists no randomized algorithm that solves* APSP *in time* $O(n^{3-\epsilon})$ *for any* $\epsilon > 0$*.*

Although it is easier to work with the shortest distance version of APSP, we have the same hardness for both shortest distance and shortest paths between all pairs.

As explained earlier in this chapter, it is also not *that* important to give very strong arguments for why the conjectured hardness of our problems is believable. We can always use the conjecture to show comparable hardness for many other computational problems as we will see consequently. Finding a faster algorithm for any of the problems to which one of our popular problems reduces to will then simply let us reformulate our conjecture.

As a final remark before we jump into the classical known fine-grained reductions: While we defined the hardness of our popular problems for randomized algorithms, the classical reductions shown in this thesis are fully deterministic. As a consequence, we will simply be referring to 'algorithms' as opposed to 'randomized algorithms' in our theorem statements. Randomized fine-grained reductions do exist, as shown in [41], and for this work we need to randomize our hardness conjectures to make the comparative analysis valid.

3.2 Fine-Grained Reductions From APSP

Johnson's algorithm described in the previous section does more than we require for solving APSP since it also outputs the shortest paths themselves. The consensus in fine-grained literature is to stick to the shortest distance version and this is mostly due to the relation between APSP and the distance matrix product.

3.2.1 APSP and Matrix Multiplication

Since we can handily represent all the information of a graph using its adjacency- or weight matrix, there are very close relations between graph problems and matrix problems. Many problems can then come to depend on how fast we can perform matrix multiplication. It becomes important here to clarify which kind of matrix multiplication we are talking about. The standard matrix product is taken over the field $\mathbb{R}$ with standard addition and multiplication. This is the most common type of matrix multiplication and new techniques keep being developed to improve on its computational upper

bound. Since the best known upper bound is constantly improved upon, we say that the product of two $n \times n$ matrices takes time $O(n^\omega)$, with as of this moment $\omega < 2.3728596$ [4]. It is generally believed that no algorithm exists that performs matrix multiplication in $O(n^{2-\epsilon})$ time for $\epsilon > 0$, but a $O(n^2)$-time algorithm is not completely unthinkable. In fact, many algorithms that use matrix multiplication and have a time complexity parametrized by ω, turn out to have very 'nice' upper bounds when $\omega = 2$.

'Nice' in this context means nothing else than that the upper bound is something that we may have expected and seems natural to us like in the matrix multiplication case or because it matches a lower bound. We won't actually see much use of this kind of matrix multiplication for the reductions seen in this chapter, since it is more commonly used to show complexity upper bounds as opposed to lower bounds. We'll make use of standard matrix multiplication, henceforth simply referred to as matrix multiplication in Chapter 5, and will get back to this as it becomes relevant.

We can view a matrix as a 2-dimensional array for storing data and in doing so it is not hard to imagine performing matrix operations using other operations than multiplication and addition. More formerly, as opposed to matrix multiplication over the $(+, \times)$-ring, we can consider matrix multiplication over other ring or semi-ring structures. While standard matrix multiplication is believed to possibly go as fast as $O(n^2)$ for $n \times n$ matrices, matrix multiplication over an arbitrary semi-ring knows no $O(n^{3-\epsilon})$ time algorithm for any $\epsilon > 0$. This can be explained in part by the fact that clever algorithms for standard matrix multiplication make use of subtraction, an operation that is not always available in semi-rings. The alternative matrix multiplication that is relevant for our work is matrix multiplication over the $(\min, +)$-semi ring.

Definition 3.2.1 (Distance Product)**.** Matrix multiplication of $m \times l$ matrix M and $l \times n$ matrix N over the $(\min, +)$-semi ring is denoted $M \star N$ and defined by:

$$(M \star N)[ij] := \min_{k \in [l]} (M[ik] + N[kj])$$

Inspecting the operation a bit more closely, we can already see some connections to the APSP problem, we are computing the smallest element of a list of sums, while in APSP we want to compute the smallest edge-weight sum over all possible paths. For this reason we may also call this type of matrix multiplication the distance product of two matrices. The distance product allows us to formulate a corresponding computational problem:

Definition 3.2.2 ($(\min, +)$-Matrix Multiplication)**.** Given two $n \times n$ matrices M, N, with entries from $[-n^c, n^c]$ for some constant c, compute $M \star N$.

The relation between the distance product and APSP was first discussed in [25, 37], where Fischer and Munro discussed the transitive closure of graphs.

We can use the distance product to find the distance matrix of a graph by repeated squaring of the weight matrix of the graph.

Theorem 3.2.1 (APSP $\leq_{n^3,n^3}$ $(\min,+)$-Matrix Multiplication)**.** *If* $(\min,+)$-Matrix Multiplication *over* $n \times n$ *matrices with entries from* $[-n^c, n^c]$ *can be solved by an* $O(n^{3-\epsilon})$ *time algorithm for some* $\epsilon > 0$*, then* APSP *over weighted graphs with* n *nodes and entries from* $[-n^c, n^c]$ *can be solved in* $O(\log(n)n^{3-\epsilon})$ *time.*

Proof. Let $G = (V, E)$ be a weighted directed graph with no negative cycles and with weights from $[-n^c, n^c]$ and let W be its weight matrix where we set all diagonal entries to 0. For the distance product we define the k'th power of W to be $W^k := W \star W \star ... \star W$, with k copies of W in the sequence.

We claim that W^n is the shortest distance matrix of G. First we show by induction that $W^k[ij]$ is the length of the shortest path from i to j over at most k edges and $W^k[ij] = \infty$ if there is no such path. This is clearly true for the base case W. Assume that it is true for W^k, we show that it holds for W^{k+1}. We have that $W^{k+1} = W^k \star W$ and therefore

$$W^{k+1}[ij] = \min_{1 \leq l \leq n} (W^k[il] + W[lj])$$

We know that $W^k[il]$ holds the length of the shortest path from i to l for all $1 \leq l \leq n$ over k edges if such a path exists. We prove the inductive step by case distinction:

Suppose there is no path that crosses $k+1$ edges that is shorter than the shortest path over k edges between nodes i and j. Then W^k stores the length of that path, and since the diagonal entries are 0 we have that

$$\min_{1 \leq l \leq n} (W^k[il] + W[lj]) = W^k[ij] + W[jj]$$

is the shortest distance from i to j. If there was some edge l such that

$$W^k[il] + W[lj] < W^k[ij] + W[jj],$$

there would either be a shorter path from i to j over $k+1$ edges, which violates the case assumption, or there would be a shorter path from i to j over k edges, which violates the induction hypothesis.

In the next case, suppose there is a path from i to j that consists of $k+1$ edges and that is shorter than the shortest path over k edges. Let (l, j) be the last edge on that path over $k+1$ edges. It follows that the shortest path from i to l is over k edges and its distance will be $W^k[il]$. As a result

$$W^{k+1}[ij] = W^k[il] + W[lj],$$

which is the length of the shortest path over $k+1$ edges from i to j.

Lastly, assume that there exists no path over $k+1$ edges from i to j. Then there is no path over k edges from i to j. It must be that for all $1 \leq l \leq n$ it holds that $W^k[il] = \infty$ or $W[lj] = \infty$ and as a result $W^{k+1}[ij] = \infty$.

Since there are no negative cycles in our graph, we know that every shortest path will cross at most n edges, and it must be that W^n is the shortest distance matrix. We can compute W^n using $(\min, +)$-Matrix Multiplication and repeated squaring: $W^n = W^{\frac{n}{2}} \star W^{\frac{n}{2}}$. It takes $\log(n)$ calls to $(\min, +)$-Matrix Multiplication with $n \times n$ sized matrices. The theorem statement follows. □

We find that, in fact, $(\min, +)$-Matrix Multiplication is fine-grained equivalent to APSP.

Theorem 3.2.2 ($(\min, +)$-Matrix Multiplication $\leq_{n^3, n^3}$ APSP)**.** *If* APSP *over weighted graphs with n nodes and weights from $[-n^c, n^c]$ can be solved by an algorithm in time $O(n^{3-\epsilon})$ for $\epsilon > 0$, then* $(\min, +)$-Matrix Multiplication *over $n \times n$ matrices and entries from $[-n^c, n^c]$ can be solved in time $O(n^{3-\epsilon})$.*

Proof. Let M and N be two $n \times n$ matrices. We will construct a graph in a way that we can read the product $M \star N$ from its distance matrix D.

Let $G = (V, E)$ be the graph with vertex set $V = A \cup B \cup C$ such that $A = \{a_1, , , a_n\}$, $B = \{b_1, , , b_n\}$ and $C = \{c_1, , , c_n\}$. For every $M[ij]$ we add edge (a_i, b_j) with weight $M[ij]$ to E and for every $N[ij]$ we add edge (b_i, c_j) with weight $N[ij]$ to E. For a pair of nodes a_i, c_j it holds that the distance of its shortest path is

$$\min_{b_k \in B}(w(a_i, b_k) + w(b_k, c_j)) = \min_{1 \leq k \leq n}(M[ik] + N[kj]) = (M \star N)[ij].$$

Since $|V| = 3n$ and $|E| = 2n^2$, it takes $O(n^2)$ time to construct the graph G, furthermore it takes $O(n)$ time to read the correct entries from D. □

The equivalence follows immediately.

Corollary 3.2.1 (APSP $\equiv_{n^3, n^3}$ $(\min, +)$-Matrix Multiplication)**.** *For weighted graphs over n nodes and with weights from $[-n^c, n^c]$ there is an $O(n^{3-\epsilon})$ time algorithm for* APSP *for $\epsilon > 0$ if and only if an $\tilde{O}(n^{3-\epsilon})$ algorithm exists for* $(\min, +)$*-MM over $n \times n$ matrices with entries from $[-n^c, n^c]$.*

Proof. Follows from Theorem 3.2.1 and Theorem 3.2.2. □

The cubic hardness-equivalence between APSP and $(\min, +)$-Matrix Multiplication has been known for a while and seems natural to the extent that in both problems we are minimizing over

sums; summing path weights or summing matrix entries.

The next series of reductions from $(\min, +)$-Matrix Multiplication to our end goals of Δ-Matching Triangles and Triangle Collection are all triangle related graph problems.

3.2.2 Graph Triangle Finding

In the next series of problems, all our problem instances will be weighted graphs, and the instances can of course be represented fully by their weight matrices. More specifically, the graph problems we'll discuss in this section have to do with triangle finding problems.

Definition 3.2.3 (Graph Triangles)**.** Let $G(V, E)$ be a weighted graph. Nodes a, b, c form a triangle if $(a, b), (b, c), (c, a) \in E$.

- If nodes a, b, c form a triangle in a weighted graph G, then the weight of the triangle is $w(a, b) + w(b, c) + w(c, a)$.

In directed graphs, triangles are 3-cycles and in undirected graphs they are 3-cycles and 3-cliques.

Another important definition that we'll often encounter in graph-triangle problems is that of tripartiteness.

Definition 3.2.4 (Tripartite)**.** A graph $G = (V, E)$ is tripartite if there is a partition of $V = A \cup B \cup C$ for which it holds that there exists no $(i, j) \in E$ such that $i \wedge j \in A$ or $i \wedge j \in B$ or $i \wedge j \in C$.

Combining the tripartiteness definition with the triangle one from before, we can observe an interesting property about triangles in tripartite graphs: A triangle in a tripartite graph must contain one node in each of the three different graph parts.

Assuming that the graph input for a triangle problem is tripartite can simplify our reductions a great deal. Here we show that we can often make this assumption without loss of generality.

Theorem 3.2.3 (Tripartiteness Assumption)**.** *Let $G = (V, E)$ be a weighted graph over n nodes and with weights from $[-n^c, n^c]$ and let t be the number of triangles in G. We can construct a tripartite graph G' from G in $O(n^2)$ time such that the number of triangles in G' is $3t$.*

Proof. Let $G = (V, E)$ be a weighted graph with $V = \{v_1, , , v_n\}$ and t triangles. Define vertex set $V' = A \cup B \cup C$ such that $A = \{a_1, , , a_n\}$, $B = \{b_1, , , b_n\}$ and $C = \{c_1, , , c_n\}$. We define our new edge set E' as follows: For every edge $(v_i, v_j) \in E$, add edges (a_i, b_j), (b_i, c_j) and (c_i, a_j) to E'. It is

clear that $G' = (A \cup B \cup C, E')$ is tripartite, now we show that G' contains $3t$ triangles.

Let v_i, v_j, v_k be a triple of nodes in V. Suppose that the triple of nodes forms a triangle in G. It must be that $(v_i, v_j), (v_j, v_k), (v_k, v_i) \in E$ and as a result $(a_i, b_j), (b_j, c_k), (c_k, a_i) \in E'$. We then know that the triple of nodes a_i, b_j, c_k forms a triangle in G'. Similarly, the triples a_k, b_i, c_j and a_j, b_k, c_i form triangles in G'.

Assume a_i, b_j, c_k forms a triangle in G'. It must be that $(a_i, b_j), (b_j, c_k), (c_k, a_i) \in E'$ and by definition $(v_i, v_j), (v_j, v_k), (v_k, v_i) \in E$. It follows that v_i, v_j, v_k is a triangle in G.

Every triangle v_i, v_j, v_k in G gives rise to three triangles in G' indexed with i, j, k and every triangle in G' must be the result of a triangle in G. We can conclude that G' contains $3t$ triangles. Furthermore, since $O(|A| + |B| + |C|) = O(|V|)$ and $O(|E'|) = O(|E|)$, we can construct G' in $O(n^2)$ time. □

Assuming tripartiteness on graph inputs then works for graph problems where we need to determine the existence of triangles or make a triangle count.

The next series of reductions to triangle problems is of relevance due to their connection to the other two hardness conjectures. The reductions are all taken from [3, 54, 53].

To get to these problems from APSP, we depart from $(\min, +)$-Matrix Multiplication to the somewhat artificial problem All-Pairs Negative Triangle, which looks at pairs of nodes in a tripartite graph and determines whether they are part of a triangle that has total negative edge-weight.

Definition 3.2.5 (All-Pairs Negative Triangle)**.** Given a tripartite weighted graph $G = (A \cup B \cup C, E)$ over $O(n)$ nodes and with weights from $[-n^c, n^c]$ determine for every pair of nodes a, b such that $a \in A$ and $b \in B$ whether there exists $c \in C$ such that nodes a, b, c form a triangle of negative weight in G.

The All-Pairs Negative Triangle problem is mostly a stepping stone towards showing the reduction to the Negative Triangle problem, and so we can let tripartiteness be part of the problem definition. It is also not quite clear whether the tripartiteness could be assumed on a graph instance of All-Pairs Negative Triangle w.l.o.g.

Definition 3.2.6 (Negative Triangle)**.** Given a weighted graph G over n nodes with weights from $[-n^c, n^c]$, determine if there is a triangle in G with negative weight.

From Negative Triangle we go to 0-Weight Triangle, which is also the meeting point for 3SUM:

Definition 3.2.7 (0-Weight Triangle)**.** Given a weighted graph G over n nodes and with weights from $[-n^c, n^c]$, determine if there is a triangle in G with weight 0.

The final two problems are closely related and can be reduced to from 0-Weight Triangle. Through a separate route, also SAT and k-SAT reduce to these problems. The following two problems will be the main focus in the quantum setting in Chapter 4 and Chapter 5.

First we have Δ-Matching Triangles, a parameterized problem on colored graphs where we are tasked to find Δ triangles of the same color.

Definition 3.2.8 (Δ-Matching Triangles)**.** Given a colored graph $G = (V, E)$ over n nodes and with color function $\gamma : V \to \Gamma$, determine if there is a triple of colors $i, j, k \in \Gamma$ such that there are Δ triangles a, b, c for which $(\gamma(a), \gamma(b), \gamma(c)) = (i, j, k)$.

On colored graphs we can also ask whether there is a triangle for any possible color triple.

Definition 3.2.9 (Triangle Collection)**.** Given a colored graph $G = (V, E)$ over n nodes with color function $\gamma : V \to \Gamma$, determine if for every triple of colors $i, j, k \in \Gamma$ there is at least one triangle a, b, c for which $(\gamma(a), \gamma(b), \gamma(c)) = (i, j, k)$.

In the remainder of this section we present all the reductions from APSP to the above defined problems. The chain starts with the reduction from $(\min, +)$-Matrix Multiplication to All-Pairs Negative Triangle, from [53].

Theorem 3.2.4 ($(\min, +)$-Matrix Multiplication $\leq_{n^3, n^3}$ All-Pairs Negative Triangle)**.**
If All-Pairs Negative Triangle *over graphs of n nodes and weights from $[-n^c, n^c]$ can be solved by an $O(n^{3-\epsilon})$ time algorithm for $\epsilon > 0$, then* $(\min, +)$-Matrix Multiplication *over $n \times n$ matrices with entries from $[-n^c, n^c]$ can be solved in $O(\log(n) n^{3-\epsilon})$ time.*

Proof. Let M, N be two $n \times n$ matrices. We construct the tripartite graph $G = (A \cup B \cup C, E)$ with $A = \{a_1, , , a_n\}, B = \{b_1, , , b_n\}$ and $C = \{c_1, , , c_n\}$. Then for every pair $(b_j, c_k) \in B \times C$ we add an edge to E with weight $w(b_j, c_k) = N[jk]$ and for every pair $(c_k, a_i) \in C \times A$ we add an edge to E with weight $w(c_k, a_i) = M[ki]$.

For every pair a_i, b_j, there is a triangle a_i, b_j, c_k for each $k \in [n]$. We want to determine the largest weight that the edge (a_i, b_j) can have for which there

exists a negative triangle that contains a_i and b_j. In this specific scenario we are certain that the only negative triangle a_i and b_j are part of is the smallest triangle. It will also be true that $w(a_i, c_k) + w(c_k, a_i) = -w(a_i, b_j) - 1$ and we have that:

$$\begin{aligned} -w(a_i, b_j) - 1 &= w(b_j, c_k) + w(c_k, a_i) \\ &= \min_{k \in [n]} (w(b_j, c_k) + w(c_k, a_i)) \\ &= \min_{k \in [n]} (M[jk] + N[ki]) = (M \star N)[ji] \end{aligned}$$

We find these values $w(a_i, b_j)$ by binary searching the integer weight space and making repeated use of the All-Pairs Negative Triangle algorithm. First set the lower bound $l(a_i, b_j) = -n^c$ and the higher bound $h(a_i, b_j) = n^c$. Define $w(a_i, b_j) = \frac{l(a_i, b_j) + h(a_i, b_j)}{2}$. Now run All-Pairs Negative Triangle on the graph G. For every pair of nodes a_i, b_j, if they are part of a negative triangle, set $l(a_i, b_j) = w(a_i, b_j)$ for the next run and if they are not part of a negative triangle, set $h(a_i, b_j) = w(a_i, b_j)$ for the next run. We make $\log(2n^c)$ calls to All-Pairs Negative Triangle in this manner until $l(a_i, b_j) = h(a_i, b_j) = w(a_i, b_j)$. □

All-Pairs Negative Triangle is a somewhat artificial problem but does allow us to reduce APSP to the more interesting Negative Triangle, as done in [53].

Theorem 3.2.5 (All-Pairs Negative Triangle $\leq_{n^3, n^3}$ Negative Triangle)**.** *If* Negative Triangle *on weighted graphs of n nodes with weights from $[-n^c, n^c]$ can be solved by an $O(n^{3-\epsilon})$ time algorithm for $\epsilon > 0$, then* All-Pairs Negative Triangle *on weighted graphs of n nodes and weights from $[-n^c, n^c]$ can be solved in $O(\log(n) n^{3-\epsilon})$ time.*

Proof. Let $G = (A \cup B \cup C, E)$ be a tripartite, weighted graph with $|A| = |B| = |C| = n$. First we claim that upon detecting a negative triangle in G, we can find it efficiently. We do this as follows:

Suppose we query the negative triangle oracle to determine whether there is a negative triangle in G. If the oracle answers 'yes', we split the vertex set V into 4 equally sized subsets. We know that there must be at least one combination of three vertex sets that contains the nodes of a negative triangle. As soon as we've determined such a triple of three vertex sets using the negative triangle algorithm, we eliminate the remaining fourth of the vertex set. We repeat this step, eliminating a fourth of the remaining vertex sets at every iteration, until only three vertices remain. If our Negative Triangle oracle runs in time n^3, the running time of finding a negative triangle after detection will be $O(n + (\frac{3}{4}n)^3 + (\frac{9}{16}n)^3 + \dots) = O(\log(n) n^3)$.

We will use the above sub-algorithm to solve All-Pairs Negative Triangle on G. First we split A, B and C each into n^α graphs with $n^{1-\alpha}$ nodes. We can use these vertex subsets to create $n^{3\alpha}$ new tripartite sub-graphs of G, taking always a subset of A, a subset of B and a subset of C to make the vertex set of our new sub-graph. For each of these new tripartite graphs, we run the negative triangle detecting and finding algorithm to find a triangle a, b, c with $a \in A$, $b \in B$ and $c \in C$. After finding such a triangle report 'yes' for the pair (a, b) and remove the edge (a, b) from all the new sub-graphs. We repeat until no new triangles are detected in any of the $n^{3\alpha}$ sub-graphs.

The total number of sub-graphs is $n^{3\alpha}$ but the number of pairs for which we need to find a check whether they are part of a negative triangle is $O(n^2)$. Since the size of the Negative Triangle oracle input is $n^{\alpha-1}$ we find a complexity of $O(T(n^{1-\alpha}) \cdot (n^{3\alpha} + n^2))$. For $T(n) = O(n^3)$, this complexity is optimized at $\alpha = \frac{2}{3}$ and the theorem statement follows. □

In a nice result that we don't necessarily need for the results in this thesis, but that does give us some interesting insight later on in the quantum setting we can reduce Negative Triangle back to $(\min, +)$-Matrix Multiplication as done in [53].

Theorem 3.2.6 (Negative Triangle $\leq$ $(\min, +)$-Matrix Multiplication)**.** *If* $(\min, +)$-Matrix Multiplication *on* $n \times n$ *matrices with weights from* $[-n^c, n^c]$ *can be solved by an* $O(n^{3-\epsilon})$ *time algorithm for some* $\epsilon > 0$*, then* Negative Triangle *on graphs of* n *nodes and weights from* $[-n^c, n^c]$ *can be solved in* $O(n^{3-\epsilon})$ *time.*

Proof. Let $G = (V, E)$ be a weighted graph and W its weight matrix. Compute $M = W + (W \star W)^T$ and check if there are any entries (i, j) for which $M[ij] \leq 0$. Whenever this is the case we will have

$$W[ij] + (W \star W)[ji] = w(i, j) + \min_{1 \leq k \leq n} (w(j, k) + w(k, i)) < 0.$$

Which shows that for nodes i, j if we take k such that the triple of nodes i, j, k has the smallest combined edge-weight we will have a negative triangle. □

Since we already had fine-grained equivalence between APSP and $(\min, +)$-Matrix Multiplication, we now have fine-grained equivalence between APSP, $(\min, +)$-Matrix Multiplication, All-Pairs Negative Triangle and Negative Triangle. The cubic hardness equivalence of our reductions breaks down in the reduction from Negative Triangle to 0-Weight Triangle from [54].

For this reduction we will first need a lemma that lets us relate inequality to equality, also proven in [54]. We leave the proof out of this thesis since we can use the lemma as a black box, and we won't need its proof in Chapter 4.

Lemma 3.2.1. *For non-negative integers x, y, z we have that $x+y < z$ if and only if there exists a k such that either:*

$$\lfloor \frac{x}{2^k} \rfloor + \lfloor \frac{y}{2^k} \rfloor = \lfloor \frac{z}{2^k} \rfloor + 1$$

or it holds that both $\lfloor \frac{x}{2^{k-1}} \rfloor \mod 2 = \lfloor \frac{y}{2^{k-1}} \rfloor \mod 2 = 0$ and

$$\lfloor \frac{x}{2^k} \rfloor + \lfloor \frac{y}{2^k} \rfloor = \lfloor \frac{z}{2^k} \rfloor$$

Proof. See [54]. □

Since the weights of our graphs are always taken from a space polynomial in the size of our graph, the conditions in the lemma only need to be checked for $k = O(\log(n))$. We can then use the lemma to construct a logarithmic number of graphs to account for all the possible cases in the lemma. We show the idea in more detail in the reduction from Negative Triangle to 0-Weight Triangle below.

Theorem 3.2.7. *If* 0-Weight Triangle *on graphs of n nodes and with weights from $[-n^c, n^c]$ can be solved by an $O(n^{3-\epsilon})$ time algorithm for some $\epsilon > 0$, then* Negative Triangle *on graphs of n nodes and weights from $[-n^c, n^c]$ can be solved in $O(\log^2(n) n^{3-\epsilon})$ time.*

Proof. Let $G = (V, E)$ be a weighted graph with weight function w and assume w.l.o.g. that it is tripartite with partition $V = A \cup B \cup C$. We make $2\log(n^c)$ new copies $G_{i,j} = (A \cup B \cup C, E)$ of the graph G with the same vertex sets and edge sets, but with a new weight function $w_{i,j}$ for $1 \leq i \leq \log(n^c)$ and for $j = 0$ or $j = 1$.

For nodes $a \in A$ and $b \in B$, with edge-weight $w(a, b)$, we set $w_{i,j}(a, b) = \lfloor \frac{w(a,b)}{2^i} \rfloor$. For $b \in B$ and $c \in C$ we set $w_{i,j}(b, c) = \lfloor \frac{w(a,b)}{2^i} \rfloor$. For $c \in C$ and $a \in A$, we set $w_{i,j}(c, a) = -\lfloor \frac{w(c,a)}{2^i} \rfloor + j$.

For each of the graphs we run the 0-Weight Triangle algorithm. Whenever we detect a 0-weight triangle in a graph $G_{i,0}$ we know there must be a negative triangle in G. If we detect a 0-weight triangle a, b, c in a graph $G_{i,1}$ for some i we use the triangle finding algorithm from Theorem 3.2.5 to locate it and check whether $\lfloor \frac{w_{i,0}(a,b)}{2^{k-1}} \rfloor \mod 2 = \lfloor \frac{w_{i,0}(b,c)}{2^{k-1}} \rfloor \mod 2 = 0$ to determine whether there is a negative triangle in G. We run the 0-Weight Triangle algorithm $O(\log(n))$ times for $O(\log(n^c))$ of the graphs and the theorem statement follows. □

Before the reduction from Negative Triangle to 0-Weight Triangle was shown in [54] it was already shown in [41] that 3SUM reduces to 0-Weight Triangle as well. The next reductions from 0-Weight Triangle provide the meeting point for SAT and k-SAT

also.

The reduction from 0-Weight Triangle to Δ-Matching Triangles requires another lemma which was used in [2] and later reformulated in [3]. Here we state it without proof, since we can use the lemma as a black box in Chapter 4.

Lemma 3.2.2. *For integers $p, d, s, n, c \geq 1$ if $p \geq 3n^{\frac{c}{d}}$ and $s = 4^{d-1}$ there is a collection of functions $f_1, , , f_s : [-n^c, n^c] \to [-\frac{p}{3}, \frac{p}{3}]^d$ and target vectors $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_s \in [-p, p]^d$, computable in $O(\log(n))$ time, such that for all numbers $x, y, z \in [-n^c, n^c]$ it holds that*

$$x + y + z = 0 \text{ iff } \exists j \in [s] \text{ s.t. } f_j(x) + f_j(y) + f_j(z) = \boldsymbol{t}_j$$

Proof. See [2]. □

The reduction from 0-Weight Triangle to Δ-Matching Triangles and Triangle Collection requires us to turn a weighted graph instance into a colored graph instance. In order to encode the edge weights we use Lemma 3.2.2 to encode the weight space $[-n^c, n^c]$ of a graph into d-dimensional vectors and construct a node for each possible edge weight. We see it in more detail in the following reduction.

Theorem 3.2.8 (0-Weight Triangle $\leq_{n^3, n^3}$ Δ-Matching Triangles)**.** *If Δ-Matching Triangles on colored graphs of n nodes can be solved by an $O(n^{3-\epsilon})$ time algorithm for some $\epsilon > 0$ and $\omega(1) \leq \Delta(n) \leq o(\log(n))$, then 0-Weight Triangle on weighted graphs of n nodes and with weights from $[-n^c, n^c]$ can be solved in $O(\log(n)n^{3-\epsilon})$ time.*

Proof. Let $G(V, E)$ be a weighted graph with weight function $w : E \to [-n^c, n^c]$ and assume that it is tripartite with vertex partition $V = A \cup B \cup C$ and let $A = \{a_1, , , a_n\}$, $B = \{b_1, , , b_n\}$ and $C = \{c_1, , , c_n\}$.

We will use Lemma 3.2.2 setting $d = \Delta, p = O(n^{\frac{c}{\Delta}})$ and $n = n$ to construct $s = 2^{O(\Delta)}$ functions $f_i : [-n^c, n^c] \to [-n^{\frac{c}{\Delta}}, n^{\frac{c}{\Delta}}]^\Delta$. Now we use f_i to construct unweighted colored graphs $G_i = (A_i \cup B_i \cup C_i, E_i)$ for each $i \in [s]$.

For every $a \in A$ add Δ copies labelled a_j for $j \in [\Delta]$ to A_i and let their color be a. For each $b \in B$ add $\Delta \cdot 2n^{\frac{c}{\Delta}}$ copies to B_i labelled $b_{j,x}$ for $j \in [\Delta]$ and $x \in [-n^{\frac{c}{\Delta}}, n^{\frac{c}{\Delta}}]$ and with color b. Similarly, add $\Delta \cdot 2n^{\frac{c}{\Delta}}$ copies $c_{j,x}$ with color c for each $c \in C$ to C_i. That is, we add a node to B and C for every dimension up to Δ and every possible vector entry value in $[-n^{\frac{c}{\Delta}}, n^{\frac{c}{\Delta}}]$.

Now for the edges: For each $(a, b) \in A \times B$ add the edges $(a_j, b_{j,f_i(w(a,b))[j]})$ to E_i for every $j \in [\Delta]$. For each $(b, c) \in B \times C$ add the edges $(b_{j,x}, c_{j,x+f_i(w(b,c))[j]})$ to E_i for every $j \in [\Delta]$. Finally, for each

$(c, a) \in C \times A$, add edges $(c_{j,t_i[j]-f_i(w(c,a))[j]}, a_j)$ to E for every $j \in [\Delta]$. That is, we have edges going only from nodes in the same dimension, with the specific value for the b and c nodes given by our constructed function f_i. Every a_j has one outgoing edge and every b_j, x has one outgoing edge, even for the values of x that don't have an edge incoming from a_j. For every c only Δ of the $\Delta 2n^{\frac{c}{\Delta}}$ nodes $c_{j,x}$ has an outgoing edge.

We now claim that G has a 0-weight triangle if and only if there exists some $i \in [s]$ such that G_i has Δ matching triangles.

Suppose G has a 0-weight triangle a, b, c. It follows from Lemma 3.2.2 that there is some $i \in [s]$ such that $f_i(w(a, b)) + f_i(w(b, c)) + f_i(w(c, a)) = \boldsymbol{t}_i$. In the following set for ease of notation $\boldsymbol{x} = f_i(w(a, b)), \boldsymbol{y} = f_i(w(b, c))$ and $\boldsymbol{z} = f_i(w(c, a))$. We know that there exists an $i \in [s]$ such that

$$x[j] + y[j] + z[j] = t_i[j],$$

for each $j \in [\Delta]$. By our construction it always holds for a graph G_i that $(a_j, b_{j,x[j]})$, $(b_{j,x[j]}c_{j,(x+y)[j]})$, $(c_{j,-z[j]}, a_j) \in E_i$ for each $j \in [\Delta]$. Since $(x+y)[j] = (t_i - z)[j]$ by the previous observation, it follows that $c_{j,(x+y)[j]} = c_{j,t_i[j]-z[j]}$ and the triple of nodes $(a_j, b_{j,x[j]}, c_{j,-z[j]})$ forms a triangle for every $j \in [\Delta]$. Furthermore, since the triangle $(a_j, b_{j,x[j]}, c_{j,-z[j]})$ is the same color for every $j \in [\Delta]$, we have Δ matching triangles.

Conversely, suppose there are Δ matching triangles in G_i for some $i \in [s]$. We know from the construction of our graphs G_i that the colors must correspond to a triple of nodes $(a, b, c) \in A \times B \times C$ in the graph G. Every $a_j \in A_i$ has only one outgoing edge $(a_j, b_{j,x})$ and so it must be that $x = f_i(w(a, b))[j]$. The node $b_{j,x}$ also only has one outgoing edge $(b_{j,x}, c_{j,x+y})$ and it must be that $y = f_i(w(b, c))[j]$. Lastly, since a_j has only one incoming edge $(c_{j,t_i[j]-z}, a_j)$ it must be that $z = t_i[j] - f_i(w(c, a))[j]$ and as a result $f_i(w(a, b))[j] + f_i(w(b, c))[j] = t_i[j] - f_i(w(c, a))[j]$ for each $j \in [\Delta]$. It follows by Lemma 3.2.2 that $w(a, b) + w(b, c) + w(c, a) = 0$.

We can use the above algorithm to solve 0-Weight Triangle using a Δ-Matching Triangles oracle. We construct $2^{(O(\Delta))}$ graphs of $O(\Delta n \cdot n^{\frac{c}{\Delta}})$ nodes and $O(\Delta m n^{\frac{c}{\Delta}})$ edges each. To keep the number of graphs small enough, the algorithm only works for providing a sub-cubic algorithm for 0-Weight Triangle if $\Delta = o(\log n)$. On the other hand, to ensure that the individual graph sizes are small enough, we need to ensure that $\Delta = \omega(1)$. The theorem follows. □

A sub-cubic algorithm for $\omega(1) \leq \Delta \leq n^{o(1)}$ on graphs of n nodes provides us with a sub-cubic algorithm for 0-Weight Triangle. Since Δ-Matching Triangles can be solved in sub-cubic time for $\Delta = O(1)$,

it makes sense that the reduction does not work for that regime of Δ's. We can improve the upper bound on Δ using the following theorem:

Theorem 3.2.9. *If* Δ-Matching Triangles *on graphs with* n *nodes can be solved by a* $O(n^{3-\epsilon})$ *time algorithm for some* $\epsilon > 0$ *and* $\omega(1) \leq \Delta \leq n^{o(1)}$*, then* Δ'*-MT can be solved in* $\tilde{O}(n^{3-\epsilon})$ *time for* $\omega(1) \leq \Delta' \leq o(\log(n))$*.*

Proof. Let G be a colored graph over n nodes and suppose we want to know whether G contains Δ' matching triangles, but we have an algorithm for Δ-Matching Triangles for $\Delta \geq \Delta'$. We can add $\Delta' - \Delta$ nodes to G for every color in G. We add edges to turn the newly added nodes into a complete graph, adding $\Delta' - \Delta$ triangles of every color combination to G. We can run the Δ-Matching Triangles algorithm on the graph to solve Δ'-Matching Triangles. As long as $\Delta - \Delta' = n^{o(1)}$ the graph will have size $O(n)$. □

The reduction from 0-Weight Triangle to Triangle Collection uses the same Lemma 3.2.2 and construction from Theorem 3.2.8 for a specified value of Δ.

Theorem 3.2.10 (0-Weight Triangle $\leq_{n^3,n^3}$ Triangle Collection)**.** *If* Triangle Collection *on graphs over* n *nodes can be solved by an* $O(n^{3-\epsilon})$ *time algorithm for some* $\epsilon > 0$*, then* 0-Weight Triangle *can be solved in* $O(\log(n)n^{3-\epsilon})$ *time.*

Proof. We will use the same graph construction as in the proof of Theorem 3.2.8 but use it to construct new graphs $G'_i = (A'_i \cup B'_i \cup C'_i, E'_i)$ such that there is a 0-weight triangle in G if and only if there is a G'_i where we can't collect all color triples.

From each G_i from Theorem 3.2.8, we construct the graph G'_i by inverting all the edges in G_i. In the case that there is no 0-weight triangle in G, we want all color triples to be collected in every G'_i, which includes color triples with colors of the same parts in G, e.g. $(a, b, b') \in A \times B \times B$. To do so, we finish the constructions of the graphs G'_i by adding nodes and edges to G'_i.

For every $a \in A$, add two nodes a_B, a_C to A'_i with color a. For every $b \in B$ add nodes b_A, b_C to B'_i with color b. Lastly, for $c \in C$ add c_A, c_B to C'_i with color c.

For any pair of nodes $a, a' \in A$ we add edges (a_B, a'_B) and (a_C, a'_C) to E'_i. Similarly, for $b, b' \in B$ and $c, c' \in C$ we add edges $(b_A, b'_A), (b_C, b'_C), (c_A, c'_A), (c_B, c'_B)$ to E'_i. Finally, for pair $(a, b) \in A \times B$, we add edge (a_B, b_A) for pair $(b, c) \in B \times C$, we add (b_C, c_B) and for pair $(c, a) \in C \times A$ we add (c_A, a_C) to E'_i.

This finishes off our construction of graphs G'_i, now we prove that there is no 0-weight triangle in G if and only if there is no triangle collection in G'_i for some $i \in [2^{O(\Delta)}]$.

Suppose $(a, b, c) \in A \times B \times C$ forms a 0-weight triangle in G. Let G_i be the graph containing Δ-matching triangles from the proof of Theorem 3.2.8. We saw that every a_j in G_i can only be part of one triangle. Since there are Δ of each a_j, every such a_j must be part of a triangle in G_i if there is a 0-weight triangle in G. In our inverted graph G'_i, there can therefore be no triangle with color a. By adding the extra nodes and edges to G'_i, we have not made any new triangles that have a node in the three different parts of G'_i and there is therefore no triangle collection for graph G'_i.

Conversely, suppose there is no 0-weight triangle in G. Following the reasoning in the previous paragraph and the proof of Theorem 3.2.8, it must be that we collect every possible triple of colors $(a, b, c) \in A \times B \times C$ in every graph G'_i. We now show that we also collect every triple of colors that is not in $A \times B \times C$. Suppose we have three colors corresponding to three nodes from the same part in G, e.g. $a, a', a'' \in A$. Then nodes a_B, a'_B, a''_B from a triangle in G'_i for every i. If two colors come from one part of G and the third from another, e.g. $(a, a', b) \in A \times A \times B$, the nodes a_B, a'_B, b_a form a triangle in G'_i for every i.

Complexity-wise we have the bounds to Δ as in the proof of Theorem 3.2.8 and we can instantiate our construction by setting $\Delta = 2^{\sqrt{\log n}}$. The theorem statement follows. □

This concludes the main section of reductions from APSP. A clean overview can be seen in the graphic at the end of the chapter.

3.3 Reductions from 3SUM, SAT and k-SAT

We've formulated three different conjectures, because many problems can only be reduced to or from one of them. Ideally we'd be able to reduce the conjectures to each other so that we can reduce the amount of conjectures on which we base our hardness results. Unfortunately, none of our 3 hard problems reduces to another. The next best thing is to find problems that either reduce to all three of them or that all three reduce to. While problems that reduce to both APSP and 3SUM have been investigated in [11, 20], here we place the focus on two problems that all three hardness conjectures reduce to.

We ended our chain of reductions from APSP at the Δ-Matching Triangles and Triangle Collection problems. It turns out that both 3SUM and SAT reduce to these problems as well. 3SUM through a reduction to 0-Weight Triangle and SAT directly to Triangle Collection and through the sparsification lemma to Δ-Matching Triangles.

The 3SUM reduction to 0-Weight Triangle has already been investigated in both the classical setting in [41] and [54] and in the quantum setting in [13] so we cite the classical result here without proof.

Theorem 3.3.1 (3SUM $\leq_{n^2,n^3}$ 0-Weight Triangle)**.** *If* 0-Weight Triangle *on graphs over n nodes and with weights from $[-n^c, n^c]$ can be solved by an $O(n^{3-\epsilon})$ time algorithm for some $\epsilon > 0$, then* 3SUM *over sets of size n and with integers from $[-n^c, n^c]$ can be solved in $O(n^{2-\epsilon})$ time.*

Proof. See [41] and [54]. □

What remains is to investigate how the classical results from SAT to Δ-Matching Triangles and Triangle Collection hold up.

3.3.1 SAT and k-SAT

In the reductions in this section it will become clear why it is important to make the distinction between SAT and k-SAT. For some reductions we will need the sparsification lemma from [34] which only works for k-SAT. Ideally all our reductions would work from SAT, since it allows use to make a weaker hardness conjecture.

The algorithm for sparsifying a CNF-formula is not too complex, but the analysis of the correctness of the algorithm is rather involved and not necessary for the results in this thesis. Instead, we simply provide the algorithm here and refer to [34] for the proof of correctness and run-time of the algorithm.

Lemma 3.3.1 (Sparsification Lemma)**.** *Let ϕ be a k-CNF formula over n variables and with m clauses for $k \geq 3$. For any $\epsilon > 0$ there is an $O(2^{\epsilon n})$ time algorithm that produces $O(2^{\epsilon n})$ k-CNF formulas $\phi_1, \ldots, \phi_{O(2^{\epsilon n})}$ over n variables and cn clauses where $c = (\frac{k}{\epsilon})^{O(k)}$. It then holds that ϕ is in* SAT *if and only if there is a satisfying assignment to $\bigvee_{i=1}^{O(2^{\epsilon n})} \phi_i$.*

Proof. The intuition for sparsifying a k-CNF formula is as follows. Given a k-CNF instance over m clauses $\phi = c_1 \wedge \cdots \wedge c_m$, we try to find common *subclauses* to make *sunflowers*.

A clause $c' = l'_1 \vee \cdots \vee l'_{k'}$ is the subclause of a clause $c = l_1 \vee \cdots \vee l_k$ if every literal l part of the disjunction in c' is also part of the disjunction in c.

A sunflower is a disjunction of clauses that share a common subclause. The common subclause is called the *heart* of the sunflower and the clauses without the literals contained in the heart are called the *petals*.

Whenever we find a sunflower in a formula ϕ, we can remove all the clauses in the sunflower from ϕ and make two new copies, ϕ_p and ϕ_h. To ϕ_p we add all the petals of our sunflower as clauses and to ϕ_h we add the heart. If there is a variable assignment that satisfies all clauses in the sunflower, it must be that the heart or all the petals are satisfied by that assignment. If no such assignment exists, then no assignment exits that satisfies the heart and no assignment exists that satisfies the petals. As a result, ϕ_p and ϕ_h preserve the satisfiability of ϕ. We use ϕ_p and ϕ_h to find new sunflowers starting a tree-like structure, with ϕ as its root and the formulas $\phi_1, \ldots, \phi_{2^{\epsilon n}}$ described in the lemma statement on the leaves. The amount of clauses we need in our sunflower depends on the values k and ϵ.

To find the sunflowers, the algorithm proceeds as follows: Let ϕ be a k-CNF instance. Iterate first over $2 \leq i \leq k$ and then $1 \leq j \leq i-1$. For step i, j look for sunflowers of clauses over i literals with petals over j literals and a heart over $i-j$ literals. The amount of petals we look for in the sunflower is given by the function $C(i, \epsilon, k) = O((\frac{k}{\epsilon} 2^k)^{2^i - 1})$. Whenever we find a sunflower satisfying our restraints, halt the iteration and recurse on formulas ϕ_p and ϕ_h. If we complete the iteration over i and j without finding a sunflower terminate.

Lemma 3.3.2. *The sparsification algorithm ends in at most $2^{\epsilon n}$ steps.*

Proof. See [34]. □

Lemma 3.3.3. *The sparsification algorithm outputs $2^{\epsilon n}$ k-CNF formulas $\phi_1, \ldots, \phi_{2^{\epsilon n}}$ of at most cn clauses, where $c > \sum_{i=1}^{k} C(i, \epsilon, k)$. It holds that a assignment to the variables in ϕ satisfies ϕ if and only if it satisfies $\bigvee_{i=1}^{2^{\epsilon n}} \phi_i$.*

Proof. See [34]. □

The sparsification lemma follows from the two above claims. □

We consider it necessary to at least discuss the algorithm for sparsifying a k-CNF instance as opposed to using the sparsification lemma purely as a black box, to ensure that the algorithm also works in a quantum setting.

We are now ready to reduce SAT to Δ-Matching Triangles and Triangle Collection.

Theorem 3.3.2. *If Δ-Matching Triangles on graphs with N nodes can be solved by an $O(N^{3-\epsilon})$ time algorithm for some $\epsilon > 0$, then* SAT *over n variables and m clauses can be solved in $O((\Delta 2^{\frac{n}{3} + \frac{m}{3\Delta}})^{3-\epsilon})$ time.*

Proof. Let ϕ be a SAT instance over n variables and m clauses. Partition the set of variables into three sets N_1, N_2, N_3 of size $\frac{n}{3}$ each and index the $3 \cdot 2^{\frac{n}{3}}$ partial assignments with $i = 1, 2, 3$ depending on which of the three N_i they assign truth values to. Let C be the set of all clauses in ϕ, divide it into 3Δ

sets $C_1, \ldots, C_{3\Delta}$ of $\frac{m}{3\Delta}$ clauses each.

We make a tripartite graph $G = (V_1 \cup V_2 \cup V_3, E)$. For every partial assignment α_i to variables in V_i, clause set C_{3j+i} with $j \in [\Delta]$ and subset of clauses $s \subseteq C_{3j+i}$, add node $v_{\alpha,j,s}$ to V_i. We end up with $2^{\frac{n}{3}} \cdot \Delta \cdot 2^{\frac{m}{3\Delta}} = O(\Delta 2^{\frac{n}{3}+\frac{m}{3\Delta}})$ nodes in each V_i. The color of node $v_{\alpha,j,s}$ is α, i.e. nodes indexed by a different partial assignment have a different color.

In the following let $\alpha_4 = \alpha_1$ and $V_4 = V_1$ and let's describe the edges of G: For every pair of nodes $v_{\alpha_i,j,s}$ and $v_{\alpha_{i+1},j,s'}$ we add an edge between them if and only if α_{i+1} satisfies exactly the subset of clauses $s \subseteq C_{3j+i}$ and α_i together with α_{i+1} satisfy all clauses in $C_{3j+i+1} \backslash s'$.

We now claim that there are Δ matching triangles in G if and only if there is a satisfying assignment to ϕ.

Suppose there is a satisfying assignment to ϕ. Then there is a triple of partial assignments $\alpha_1, \alpha_2, \alpha_3$ that together satisfy all clauses in ϕ. Then for every $j \in [\Delta]$ let $s_1 \subseteq C_{3j+1}$ be exactly the subset of clauses satisfied by α_2 and similarly for $s_2 \subseteq C_{3j+2}$ and $s_3 \subseteq C_{3j+3}$. Then since the three partial assignment together must satisfy all clauses in $C_{3j+1} \cup C_{3j+2} \cup C_{3j+3}$, it must be that for α_i and α_{i+1} satisfies $C_{3j+i+1} \backslash s_{i+1}$. It follows that the triple of nodes $(v_{\alpha_1,j,s_1}, v_{\alpha_2,j,s_2}, v_{\alpha_3,j,s_3})$ forms a triangle, and we will have Δ matching triangles as a result.

Now suppose there are Δ matching triangles in G. Since edges are only drawn between nodes indexed by partial assignments to different variable parts and indexed by clause groups C_{3j+i} with the same value $j \in [\Delta]$, it must be that the triangles are of the form $(v_{\alpha_1,j,s_1}, v_{\alpha_2,j,s_2}, v_{\alpha_3,j,s_3})$ for some $s_1 \subseteq C_{3j+1}, s_2 \subseteq C_{3j+2}$ and $s_3 \subseteq C_{3j+3}$. An edge can only be drawn between v_{α_i,j,s_i} and $v_{\alpha_{i+1},j,s_{i+1}}$ if α_{i+1} satisfies s_i and α_i together with α_{i+1} satisfy $C_{3j+i+1} \backslash s_{i+1}$. It must be that together $\alpha_1, \alpha_2, \alpha_3$ satisfy all clauses in $C_{3j+1} \cup C_{3j+2} \cup C_{3j+3}$. Since each α_{i+1} satisfies exactly the set s_i there can be only one triangle in colors $(\alpha_1, \alpha_2, \alpha_3)$ for every $j \in [\Delta]$ and as a result, $(v_{\alpha_1,j,s_1}, v_{\alpha_2,j,s_2}, v_{\alpha_3,j,s_3})$ must be a triangle for each $j \in [\Delta]$ in order for Δ matching triangles to exist in colors $(\alpha_1, \alpha_2, \alpha_3)$. All clause groups are satisfied by α_1, α_2 and α_3 and as a result there is a satisfying assignment to ϕ.

Since G has $O(\Delta 2^{\frac{n}{3}+\frac{m}{3\Delta}})$ nodes, the theorem statement follows. □

We can use Theorem 3.3.2 to prove lower bounds for Δ-Matching Triangles conditioned on both k-SAT and SAT.

For SAT we find the following.

Theorem 3.3.3 (SAT $\leq_{2^n,n^3}$ Δ-Matching Triangles)**.** *If*

Δ-Matching Triangles *on graphs with N nodes can be solved by an $O(N^{3-\epsilon})$ time algorithm for some $\epsilon > 0$ and $\omega(\log(N)) \leq \Delta(N) \leq N^{o(1)}$, then* SAT *over n variables and m clauses can be solved in $O(2^{n(1-\epsilon)+o(1)})$ time.*

Proof. Suppose we have an $O(N^{3-\epsilon})$ algorithm for Δ-Matching Triangles over graphs of N nodes for some $\epsilon > 0$ and $\omega(\log(N)) \leq \Delta(N) \leq N^{o(1)}$ and let ϕ be a CNF instance over n variables and m clauses. We apply the reduction from Theorem 3.3.2 to decide ϕ in $O((\Delta 2^{\frac{n}{3}+\frac{m}{3\Delta}})^{3-\epsilon})$ time. Then since $N = O(\Delta 2^{\frac{n}{3}+\frac{m}{3\Delta}})$ and it must be that $m = \Omega(n^c)$ for some $c \in \mathbb{N}$, it follows that $\omega(m) \leq \Delta(n) \leq (2^{\frac{n}{3}+\frac{m}{3\Delta}})^{o(1)}$. For $\omega(m) \leq \Delta(n) \leq (2^{\frac{n}{3}+\frac{m}{3\Delta}})^{o(1)}$ we have that $O((\Delta 2^{\frac{n}{3}+\frac{m}{3\Delta}})^{3-\epsilon}) = O(2^{\frac{n}{3}+o(1)})^{3-\epsilon}) = O(2^{n(1-\frac{\epsilon}{3})+o(1)})$. □

To increase the range of Δ for which the reduction holds, we use the sparsification lemma. This means that this wider range of Δ only holds when basing hardness of Δ-Matching Triangles on k-SAT.

Theorem 3.3.4 (k-SAT $\leq_{2^{n\delta_k},n^3}$ Δ-Matching Triangles)**.** *If* Δ-Matching Triangles *on graphs with N can be solved by an $O(N^{3-\epsilon})$ time algorithm for some $\epsilon > 0$ and $\omega(1) \leq \Delta \leq N^{o(1)}$, then* k-SAT *over n variables and m clauses can be solved in $O(2^{n(1-\epsilon')+o(1)})$ time for some $0 < \epsilon' < \frac{\epsilon}{3}$.*

Proof. Let ϕ be a k-SAT instance over n variables and m clauses and suppose we have an $O(n^{3-\epsilon})$ algorithm for Δ-Matching Triangles over graphs with N nodes and $\omega(1) \leq \Delta \leq N^{o(1)}$. We apply the sparsification algorithm from Lemma 3.3.1 to ϕ to produce $O(2^{\epsilon' n})$ k-SAT instances $\phi_1, , , \phi_{O(2^{\epsilon' n})}$ over n variables and cn clauses for some constant c. To each ϕ_i we apply the reduction from Theorem 3.3.2 to decide whether ϕ_i is satisfiable in $O((\Delta 2^{\frac{n}{3}+\frac{cn}{3\Delta}})^{3-\epsilon})$ time. Since $\omega(1) \leq \Delta(N) \leq N^{o(1)}$ and $N = O(\Delta 2^{\frac{n}{3}+\frac{cn}{3\Delta}})$ this reduces to $O(2^{n(1-\frac{\epsilon}{3})+o(1)})$ time. We can then decide whether ϕ is satisfiable in $O(2^{\epsilon' n} 2^{n(1-\frac{\epsilon}{3})+o(1)}) = O(2^{n(1+\epsilon'-\frac{\epsilon}{3})+o(1)})$ time. For any $\epsilon' < \frac{\epsilon}{3}$ the theorem statement follows. □

The reduction to Triangle Collection works for SAT and therefore also for k-SAT, with no strings attached.

Theorem 3.3.5 (SAT $\leq_{2^n,n^3}$ Triangle Collection)**.** *If* Triangle Collection *on graphs with N nodes can be solved by an $O(N^{3-\epsilon})$ time algorithm for some $\epsilon > 0$, then* SAT *over m clauses and n variables can be solved in $\tilde{O}(2^{n(1-\frac{\epsilon}{3})})$ time.*

Proof. Let ϕ be a SAT instance over n variables and m clauses. Partition the set of variables into three sets N_A, N_B, N_C of size $\frac{n}{3}$ each and enumerate the clauses of ϕ.

We construct the graph $G = (A \cup B \cup C, E)$ as follows: For every partial assignment α to variables in N_A and for the i'th clause in ϕ, we add node a_i with color α to A. Similarly, we add nodes b_i and c_i to B and C respectively for the i'th clause and each partial assignments β and γ to N_B and N_C with colors determined by the corresponding partial assignment.

Now for the edges of G:

For each $(a_i, b_i) \in A \times B$ we add the edge (a_i, b_i) to E if neither of the corresponding partial assignments α or β satisfies the i'th clause in ϕ.

For each $(b_i, c_i) \in A \times C$ we add the edge (b_i, c_i) to E if γ does not satisfy the i'th clause.

For each $(c_i, a_i) \in C \times A$ we add an edge to E.

Now we claim that for every triple of colors (α, β, γ) coming from three partial assignments to the three different variable parts there is a triangle in G if and only if there is no satisfying variable assignment to ϕ.

First assume that the triple of partial assignments α, β, γ forms a satisfying assignment to ϕ. Any triangle in the colors (α, β, γ) will have to be of the form (a_i, b_i, c_i) for some $i \in [m]$. Since every clause is satisfied by the triple of partial assignments, each of the edges (a_i, b_i) will be missing by construction. There is no triangle that collects the color triple (α, β, γ).

Now assume that there is no satisfying assignment to ϕ. Then for every triple of partial assignments (α, β, γ) there must be some clause that is not satisfied by any of the partial assignments. Assume that it is the case for the i'th clause of ϕ. Then (a_i, b_i, c_i) is a triangle in G. Any triple of colors of the form (α, β, γ) is collected by a triangle in G.

To complete the reduction, we add edges and nodes to G to ensure that color triples that do not come from the different parts in G, e.g., of the form (α, α', β), are *always* collected. We do that using a similar construction as in the reduction from Theorem 3.2.10, adding dummy nodes $a_B, a_C, b_A, b_C, c_A, c_B$ to A, B, C respectively and for a, a', b, b', c, c' we add edges $(a_B, a'_B), (a_C, a'_C), (b_A, b'_A), (b_C, b'_C), (c_A, c'_A), (c_B, c'_B)$ to E.

It follows that there is no satisfying assignment to ϕ if and only if we have a triangle for every color triple in G.

The graph contains $O(m2^{\frac{n}{3}})$ nodes, and we can use a $O(N^{3-\epsilon})$ time Triangle Collection algorithm to solve SAT in $O((m2^{\frac{n}{3}})^{3-\epsilon}) = O(m^{3-\epsilon}2^{n(1-\frac{\epsilon}{3})})$ time. □

3.4 Extremely Popular Conjecture

We now have all the theorems at hand to prove the main result from [3] and the main result that we will try to replicate in the quantum case in Chapter 4.

Theorem 3.4.1. *If* Δ-Matching Triangles *with* $\omega(1) \leq \Delta \leq n^{o(1)}$ *or* Triangle Collection *can be solved by a* $O(n^{3-\epsilon})$ *time algorithm for some* $\epsilon > 0$*, then Conjecture 3.1.2, Conjecture 3.1.4 and Conjecture 3.1.3 must be false.*

Proof. We have that k-SAT reduces to Δ-Matching Triangles for $\omega(1) \leq \Delta \leq n^{o(1)}$ from Theorem 3.3.2 and Theorem 3.3.4 and reduces to Triangle Collection from Theorem 3.3.5. APSP reduces to 0-Weight Triangle from Theorem 3.2.2, Theorem 3.2.4, Theorem 3.2.5 and Theorem 3.2.7 and 3SUM reduces to 0-Weight Triangle from Theorem 3.3.1. We then found that 0-Weight Triangle reduces to Δ-Matching Triangles for $\omega(1) \leq \Delta \leq n^{o(1)}$ from Theorem 3.2.8 and Theorem 3.2.9 and to Triangle Collection from Theorem 3.2.10. □

An overview of all the reductions discussed in Chapter 3 can be found in the figure below.

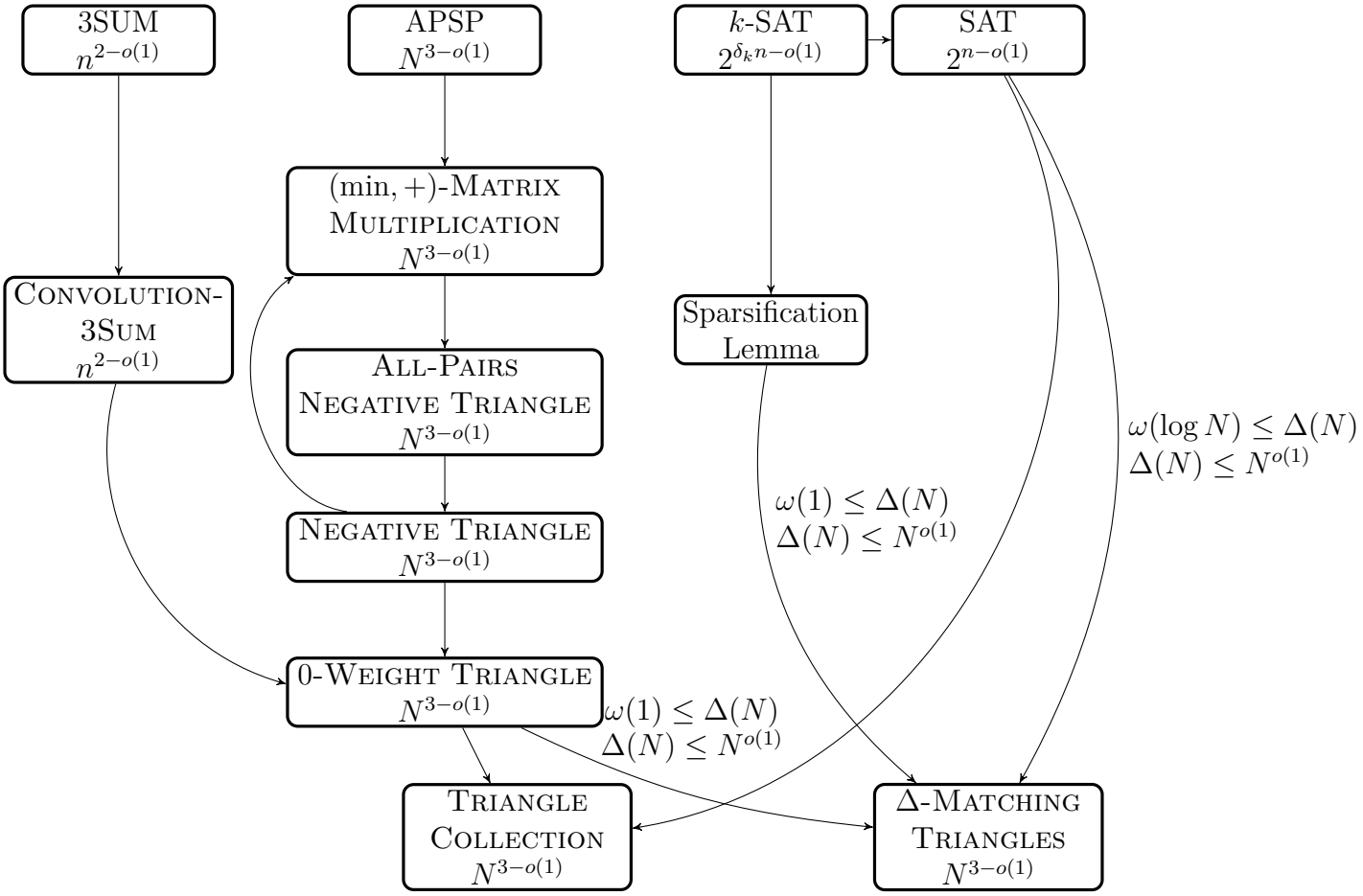

Figure 3.1: Classical reductions to Δ-Matching Triangles and Triangle Collection. The parameter n denotes the size of the sets in the case of 3SUM and Convolution-3Sum and the number of variables in the case of SAT and k-SAT. The parameter N denotes the number of nodes in a graph. In the case where multiple arrows arrive at one box, the lower bound in the second line of that box is the same for all reductions. Lastly, the lower bounds for Δ-Matching Triangles hold only for the Δ values on the incoming arrows.

Chapter 4

Quantum Fine-Grained Conditional Lower Bounds

In the classical setting a lot of work has been done in finding reductions from 3SUM, SAT and APSP, as seen in the overviews in [51] and [52].

Work in the quantum setting has only just started, with different formulations of quantum hardness conjectures for SAT and k-SAT and reductions from them being discussed in [1] and [15]. A quantum hardness conjecture for 3SUM and subsequent fine-grained reductions were researched in [13].

In this chapter we will review quantum fine-grained reductions from SAT, k-SAT, 3SUM and APSP to Δ-Matching Triangles and Triangle Collection to prove a quantum version of Theorem 3.4.1 and show a series of conditional quantum lower bounds for many computational problems in the process. Quantum upper bounds for APSP were discussed in [23] and for structured APSP instances in [39], but no formal quantum hardness conjecture has been made in current literature. We make sure to provide an extra argument for what makes a good quantum hardness conjecture for APSP. Furthermore, quantum fine-grained reductions from 3SUM to 0-Weight Triangle were already analyzed in [13], and since we will be reviewing the reduction from 0-Weight Triangle to Δ-Matching Triangles and Triangle Collection here, there is no need to review the quantum reductions from 3SUM. The focus of this chapter will therefore be on a formulation of a quantum hardness conjecture for APSP and reductions from APSP, SAT and k-SAT to Δ-Matching Triangles and Triangle Collection.

4.1 Hard Quantum Problems

From now on we will be working with quantum algorithms and the definition that we used for fine-grained reductions in the previous chapter needs to be

updated. We use a slightly modified version of the definition for quantum fine-grained reductions from [1].

Definition 4.1.1 (Quantum Fine-Grained Reductions)**.** Let $l(n)$ and $l'(n)$ be two non-decreasing functions in n. For two problems $L, L' \subseteq \{0,1\}^*$, we say that L is (l, l')-quantum fine-grained reducible to L', denoted $L \leq_{Q,l,l'} L'$ if there exists a quantum algorithm $Q^{L'}$ with query access to quantum oracle L' such that for every $\epsilon > 0$ there exists $\delta > 0$ and:

- For instance x, it holds that $Q^{L'}(x) = 1$ with probability at least $\frac{2}{3}$ if $x \in L$ and $Q^{L'}(x) = 0$ with probability at least $\frac{2}{3}$ if $x \notin L$.
- The quantum algorithm $Q^{L'}$ runs in time $O(l(n))^{1-\delta}$.
- For the sequence of instances $n_1, ..., n_k$ that $Q^{L'}$ queries to oracle L' it holds that $\sum_{i=1}^{k} l'(n_i)^{1-\epsilon} \leq O(l(n))^{1-\delta}$.

The idea remains the same as that of fine-grained reductions in the classical case, with the exceptions that we work with quantum algorithms and quantum oracles.

Working with quantum algorithms gives us access to the power of quantum computing but also provides us with some extra challenges. Since our hardness conjectures will be over smaller lower bounds, we will naturally find smaller lower bounds for many of the problems we reduce to. We may then run into the issue of the classical algorithm taking up too much time.

To illustrate: Graphs over n nodes can be of size $O(n^2)$ if they are dense. Using a reduction where we have to construct e.g. a graph over n nodes to prove a lower bound of $O(n^{2-\epsilon})$ for some $\epsilon > 0$ could be problematic, as our reduction itself would take more than $O(n^{2-\epsilon})$ time, violating the second bullet point in the definition of quantum fine-grained reductions. Luckily we don't need to compute the instances queried to our quantum oracle explicitly. Let's say instead of a quantum oracle to a problem L' we have an actual quantum algorithm to L' that we simulate or query using a quantum algorithm $Q^{L'}$. This quantum algorithm does not require the problem instances to L' explicitly, it needs random access to the values in the problem instance, e.g. the edge weights. For a problem instance of size n, we need $\log(n)$ qubits to encode the different memory addresses and since we can query the instance values in superposition, we can also compute these values in superposition 'on-the-fly'. We just need to ensure that computing the values takes time $\tilde{O}(1)$ in the input size.

We defined our quantum oracles to be probabilistic, and we will need to account for that in our algorithms. Furthermore, many of the quantum algorithms in the following sections will use probabilistic quantum algorithms as subroutines. These algorithms ensure us a correct output with probability at

least $\frac{2}{3}$. In order for the main algorithm to also have a correct output with probability at least $\frac{2}{3}$ we need to boost the probability of our subroutines and oracles somehow.

Suppose that our quantum algorithm uses m subroutines with error probabilities at most $\frac{1}{3}$ each. By the union bound we know that if we can reduce the error to $\frac{1}{3m}$, the probability that at least one of the m subroutines fails will be at most $\frac{1}{3}$. To boost the success probability of each of the individual subroutines, we need to apply each subroutine $\log(m)$ times and take the majority output if the algorithm outputs a yes/no answer. In the case of Grover Search algorithms, we can compute for each of the $\log(m)$ outputs whether the output is indeed a marked element and stop as soon as we find one. If none of the outputs is marked, we conclude that there exists no marked element. In the case of Grover Minimum finding, we can deterministically compute the minimum of all $\log(m)$ minima and use it as our output. By Hoeffding's inequality this will reduce the error probability of the individual algorithms to at most $\frac{1}{3m}$.

With the quantum fine-grained model at hand, we can discuss the hardness of the popular problems in the quantum setting.

We can easily solve SAT over n variables and m clauses with a quantum algorithm in time $O(m2^{\frac{n}{2}})$ by using Grover Search in an exhaustive search over all variable assignments. Furthermore, similarly to the classical setting, we need a quantum version of SETH, as ETH is not strong enough for providing the necessary bounds.

The different formulations of SETH easily generalize to a QSETH by applying the square root improvement provided by Grover Search. Our hardness conjecture for quantum SETH mostly resembles the one from [1], different versions of a quantum SETH were discussed in [15]. We have the following hardness for SAT and k-SAT, where we keep a distinction in hardness for the two problems.

Conjecture 4.1.1 (QSETH)**.** *Let ϕ be a k-SAT instance given over n variables and m clauses. For all $\epsilon > 0$ there exists a $k \geq 3$ such that there is no quantum algorithm that can solve k-SAT in time $O(m2^{n(\frac{1}{2}-\epsilon)})$.*

As in the classical case, we can use QSETH to prove that there is no $O(m2^{n(\frac{1}{2}-\epsilon)})$ time quantum algorithm for $\epsilon > 0$ for SAT: if there was such an algorithm we could use it to solve k-SAT in $O(m2^{n(\frac{1}{2}-\epsilon)})$ time for any k. In other words, under QSETH SAT is lower bounded by $\Omega(m2^{\frac{n}{2}-o(1)})$.

The hardness of 3SUM in the quantum setting was discussed in [13]. We saw that we can solve 3SUM in quadratic time classically by performing a structured search over the set of integers. In the outer loop we loop over the

(sorted) integers and in the inner loop we perform a traversal loop over at most $O(n)$ pairs to look for integers a, b, c such that $a + b + c = 0$. We can Groverize both loops to arrive at a linear run-time for 3SUM in the quantum model. Similar applications of Grover search to other quadratic algorithms for 3SUM have let to a conjectured linear hardness of 3SUM, first discussed in [7] and further researched in [13].

Conjecture 4.1.2 (3SUM Quantum Hardness)**.** *There exists no quantum algorithm that solves* 3SUM *in time* $O(n^{1-\epsilon})$ *for any* $\epsilon > 0$*.*

Again, since the reduction that we need from 3SUM to prove conditional hardness based on all three popular quantum conjectures was already shown in [13]; we leave the 3SUM discussion brief.

Finally, we turn our attention to a quantum hardness conjecture for APSP. In the classical setting, we conjectured that for graphs with n nodes, no $O(n^{3-\epsilon})$ time algorithm for $\epsilon > 0$ can exist. The question arises, what is a natural lower bound for APSP in the quantum setting for both the path and the distance version of APSP? In our formulation of APSP we only need to output the shortest distances to solve APSP and a speed-up for this version of the problem is natural. We can easily speed up $(\min, +)$-Matrix Multiplication by Groverizing the minimization step and using the Grover Minimum Finding algorithm from Chapter 2.

Theorem 4.1.1 ($(\min, +)$-Matrix Multiplication Upper Bound)**.** *For* $n \times n$ *matrices* M *and* N *with entries from* $[-n^c, n^c]$*, there exists a quantum algorithm that computes* $M \star N$ *in* $O(n^{2.5})$ *time.*

Proof. Let M, N be two $n \times n$ matrices with entries from $[-n^c, n^c]$. To compute the entry

$$(M \star N)[ij] = (\min_{k \in [n]}(M[ik] + N[kj])),$$

we simply use the Grover Minimum Finding Algorithm on the set $\{x | x = M[ik] + N[kj] \forall k \in [n]\}$. This takes $O(\sqrt{n})$ time. Since we need to compute n^2 entries, we need to boost the probability of finding the correct minimum by applying the Grover algorithm $O(\log(n))$ times. The distance product $M \star N$ can therefore be computed in $O(\log(n)n^{2.5})$ time. □

We can then compute the distance matrix of a graph by repeated squaring under the distance product of the weight matrix of a graph similar to the proof of Theorem 3.2.2.

Theorem 4.1.2 (APSP Upper Bound)**.** *Let* G *be a directed graph over* n *nodes and with weights from* $[-n^c, n^c]$*. There exists a quantum algorithm that computes the shortest distance matrix* D *of* G *in* $\tilde{O}(n^{2.5})$ *time.*

Proof. Given a directed graph G over n nodes, with weights from $[-n^c, n^c]$ and no negative cycles, we can run the reduction from the proof of Theorem 3.2.1 to compute the shortest distance matrix D of G. Instead of using a $(\min, +)$-Matrix Multiplication oracle, we simply apply the algorithm from the proof of Theorem 4.1.1. We compute D in $\tilde{O}(n^{2.5})$ time. □

The question remains whether there is also a $O(n^{2.5})$ time algorithm for the paths version of APSP. The common classical Floyd-Warshall algorithm does not allow for a very obvious speed-up, but a quantum single-source shortest paths algorithm was found by Dürr, Heiligman, Høyer and Mhalla in [23].

The algorithm works somewhat similarly to Dijkstra's algorithm as it iteratively constructs a shortest paths tree from a single source by adding the node closest to the source that is not yet in the tree to the shortest path tree. The algorithm finds these nodes by using a Grover Search for multiple minima finding, also proven in [23].

We state the minima finding lemma here without proof.

Lemma 4.1.1 (Grover Minima Finding)**.** *Let $f : [n] \to \mathbb{N} \cup \{\infty\}$ be a function indexing a set of numbers and suppose we have query access to f. There exists a quantum algorithm that outputs the set of d smallest elements in the image of f with probability at least $\frac{2}{3}$ in time $O(\sqrt{dn})$ and making $O(\sqrt{dn})$ queries to f.*

Proof. See [23]. □

We then find the following single-source shortest paths algorithm.

Theorem 4.1.3. *There exists a quantum algorithm that, given a weighted directed graph $G = (V, E)$ over n nodes, weight matrix W with weights from $[0, n^c]$ and input node a, computes the shortest paths between a and all other nodes in G in time $\tilde{O}(n^{1.5})$.*

Proof. Let $G = (V, E)$ be a weighted directed graph over n nodes, weight matrix W with weights from $[0, n^c]$ and suppose we want to find all shortest paths from node $a \in V$.

We want to use the minimum finding algorithm from Lemma 4.1.1, but our search space is very specific: we want to search over edges (i, j) such that i is already in our shortest path tree and j is not. Furthermore, for each vertex outside our shortest path tree, there may be multiple incoming edges from vertices inside our shortest path tree, and we would have to compare distances between all of them. We can't simply use the Grover Minimum Finding algorithm to find the unvisited node that is closest to our tentative shortest path tree: We would either have to search the set of all $O(n)$ nodes $O(n)$ times, and for each node compare its distance to a through each of the $O(n)$ nodes in the tree, or search the set of $O(n^2)$ edges $O(n)$ times. Even with

a Grover speed-up that would take $O(n)$ time at every step. Luckily, we don't need to actually compute the closest neighbors of all our tree nodes at every step. Instead, we add nodes to the tree iteratively and partition our search set to look for multiple minima during each step that we add a node to the tree.

Start by keeping a counter k, set $k = 1$ and let $S_1 = \{a\}$. At each iteration we look for the $|S_k|$ closest edges to a with sources in S_k. We use these edges to construct small-edge set U_k and add the closest edge to a from $\bigcup_k U_k$ to our shortest paths tree. We set $k = k + 1$ and construct a new source set S_k from the target vertex of the edge we just added. We repeat this procedure and whenever $|S_k| = |S_{k-1}|$ we set $S_{k-1} = S_{k-1} \cup S_k$ and $k = k - 1$.

We repeat $O(n)$ times or until all vertices not in the tree have infinite distance from a.

The source sets S_k will have size some power of 2 with $S_k < S_{k-1}$ and since we have that $\sum_{i=1^{k-1}} 2^i = 2^k - 1$, any S_k will be bigger than the union of all the following sets.

There are $\log(n)$ different set sizes that S_k can have and for each size $|S_k|$ there are at most $\frac{n}{|S_k|}$ such sets. The time required to process all sets of size S_k is $\sum_{i=1}^{\frac{n}{|S_k|}} \sqrt{|S_k|n^2} = O(\sqrt{n^3})$. Since there are $\log(n)$ different set sizes the total time required is $O(\log(n)n^{1.5})$. Furthermore, our iteration runs $O(n)$ loops so in order to boost the success rate of our Grover Minima Finding algorithm at each iteration, we need to apply it $\log(n)$ times, adding another log factor to the complexity. □

We end up with a $\tilde{O}(n^{1.5})$ time algorithm for finding all shortest paths from a single source on graphs with n nodes and no negative edge weights. Unfortunately the reweighting trick at the beginning of Johnson's algorithm from Theorem 3.1.2 takes $O(n^3)$ time, so we cannot use it to find a $\tilde{O}(n^{2.5})$ time algorithm for APSP with integer edge weights.

Remember that the Bellman-Ford algorithm that we used to reweight our graph in the proof of Theorem 3.1.2 only needs to output the shortest distances between all paths, so we can simply replace the call to Bellman-Ford in Johnson's algorithm by the Groverized repeated distance-product squaring of the extended weight matrix described in the above paragraph to reweight the graph in $\tilde{O}(n^{2.5})$ time. We then replace the calls to Dijkstra's algorithm with calls to the $\tilde{O}(n^{1.5})$ time single-source shortest path algorithm from [23] to arrive at a $\tilde{O}(n^{2.5})$ time algorithm for the paths version of APSP.

Theorem 4.1.4. *There is a quantum algorithm that solves the paths version of* APSP *over n nodes and with weights from $[-n^c, n^c]$ in $\tilde{O}(n^{2.5})$ time with probability at least $\frac{2}{3}$.*

Proof. Let G be a directed graph over n nodes with no negative cycles and weights from $[-n^c, n^c]$. We start the algorithm in the same way that Johnson's algorithm starts, by adding a node z to G that has a 0-weight edge connected to all other nodes in G. We compute the shortest distance matrix D of G in $O(\log(n)n^{2.5})$ time by repeated squaring of the weight matrix under the distance product. At every squaring step we use a Grover Minimum Finding algorithm to compute the distance product in $O(n^{2.5})$ time. Since we use $O(\log(n))$ Grover algorithms, we need to boost the success rate of each Grover Search by applying it $O(\log(\log(n)))$ times.

To any edge (i, j) with weight $W[ij]$ we then add $D[zi]-D[zj]$ to construct graph G' that has no negative weights and that is shortest-path equivalent to G. The verity of these claims follows from the proof of Theorem 3.1.2.

We use the quantum single-source shortest paths algorithm from [23] to compute n shortest path trees in $\tilde{O}(n^{2.5})$ time. □

We arrive at a likely $n^{2.5-o(1)}$ time quantum hardness conjecture for APSP. We reduce the distance version of APSP to many other problems for which we will in Chapter 5 find matching upper bounds. This will further reinforce the likelihood of $n^{2.5-o(1)}$ being a correct quantum lower bound to APSP. Then since the distance version of APSP easily reduces to the paths version of APSP, we have a conditional lower bound of $n^{2.5-o(1)}$ for the paths version of APSP through fine-grained reduction. We could of course have used this reduction to solve the shortest distance version of APSP without using the distance product, but the algorithm from Theorem 4.1.2 is so simple that it is worth providing in any case.

Technically we are solving APSP through a reduction to $(\min, +)$-Matrix Multiplication and in many cases we'd ideally give a straight forward algorithm for providing an upper bound to the complexity of the problems discussed in this thesis, without using one of the described reductions, since it can lead to unnecessary logarithmic and combinatorial overhead. If we used the reductions we'd have to give an algorithm only for the last problem reduced to, and this would in turn solve all previously reduced from problems as well. In the particular case of APSP however, the fact that we are using a reduction is not that obvious, we simply take the weight matrix of a graph and use Grover Search to compute the distance product over it swiftly, we don't need to change the mathematical structure of our problem instance at all, or construct specific problem instances to query to an oracle. We will leave the other upper bounds for the next chapter.

Conjecture 4.1.3 (APSP Quantum Hardness)**.** *There exists no quantum algorithm for* APSP *that solves it in time* $O(n^{2.5-\epsilon})$ *for any* $\epsilon > 0$.

Having formulated the hardness of our popular computational problems, we can start proving conditional quantum lower bounds for the computational problems from Chapter 3.

4.2 Quantum Fine-Grained Reductions from APSP

The first couple of results are quite straight-forward, we can use the classical reductions with occasional on-the-fly methods, and we find many quantum lower bounds based on the quantum hardness of APSP. First we retain the hardness-equivalence with $(\min,+)$-Matrix Multiplication.

Theorem 4.2.1 (APSP $\leq_{Q,n^{2.5},n^{2.5}}$ $(\min,+)$-Matrix Multiplication)**.** *If* $(\min,+)$-Matrix Multiplication *over* $n \times n$ *matrices with entries from* $[-n^c, n^c]$ *can be solved by an* $O(n^{2.5-\epsilon})$ *time algorithm for some* $\epsilon > 0$*, then* APSP *over weighted graphs with* n *nodes and entries from* $[-n^c, n^c]$ *can be solved in* $O(\log(n)\log(\log(n))n^{2.5-\epsilon})$ *time.*

Proof. We use the same reduction as was used in the proof of Theorem 3.2.1: Let $G = (V, E)$ be a weighted directed graph with weight matrix W and with n nodes. We use a $(\min,+)$-Matrix Multiplication oracle to compute $W^{\star n}$ by making $\log(n)$ calls to the oracle through repeated squaring. To boost the success probability of each of the $\log(n)$ oracle calls we need to repeat them $O(\log(\log(n)))$ times. We don't need to use on-the-fly methods, since our instance size of $O(n^2)$ is smaller than the lower bound of $O(n^{2.5})$. □

Since the above reduction does not use on-the-fly methods or quantum algorithms, Theorem 4.2.1 also holds on classical probabilistic RAM's.

Theorem 4.2.2 ($(\min,+)$-Matrix Multiplication $\leq_{Q,n^{2.5},n^{2.5}}$ APSP)**.** *If* APSP *for weighted graphs over* n *nodes and with weights from* $[-n^c, n^c]$ *can be solved by an* $O(n^{2.5-\epsilon})$ *quantum algorithm for* $\epsilon > 0$*, then* $(\min,+)$-Matrix Multiplication *over* $n \times n$ *matrices and entries from* $[-n^c, n^c]$ *can be solved in time* $O(n^{2.5-\epsilon})$*.*

Proof. We use the same reduction as was used in the proof of Theorem 3.2.2: Let M, N be two $n \times n$ matrices. We construct the graph G from Theorem 3.2.2. We can query the values of its distance matrix to find the values of $M \star N$. Constructing G takes $O(n^2)$ time and we can construct it explicitly. □

Again, we don't need any quantum operations for the above reduction to work.

The hardness equivalence follows.

Corollary 4.2.1 (APSP $\equiv_{Q,n^{2.5},n^{2.5}}$ $(\min,+)$-Matrix Multiplication)**.** *For weighted graphs over n nodes and with weights from $[-n^c, n^c]$ there is a $\tilde{O}(n^{2.5-\epsilon})$ time algorithm for* APSP *and $\epsilon > 0$ if and only if a $\tilde{O}(n^{2.5-\epsilon})$ algorithm exists for* $(\min,+)$*-MM over $n \times n$ matrices with entries from $[-n^c, n^c]$.*

Proof. Follows from Theorem 4.2.1 and Theorem 4.2.2. □

From $(\min,+)$-Matrix Multiplication we follow the classical chain of reductions, first finding a quantum conditional lower bound for All-Pairs Negative Triangle:

Theorem 4.2.3 ($(\min,+)$-Matrix Multiplication $\leq_{Q,n^{2.5},n^{2.5}}$ All-Pairs Negative Triangle)**.** *If* All-Pairs Negative Triangle *over weighted graphs of n nodes and with weights from $[-n^c, n^c]$ can be solved by an $O(n^{2.5-\epsilon})$ time quantum algorithm for $\epsilon > 0$, then* $(\min,+)$-Matrix Multiplication *over $n \times n$ matrices with entries from $[-n^c, n^c]$ can be solved in $O(\log(n)\log(\log(n))n^{2.5-\epsilon})$ time.*

Proof. Given two $n \times n$ matrices M and N with entries from $[-n^c, n^c]$, we apply the reduction from Theorem 3.2.4, constructing a graph of $O(n)$ nodes and perform a binary search over $[-n^c, n^c]$ to find the shortest paths using $\log(n)$ applications of an All-Pairs Negative Triangle oracle. To boost the individual All-Pairs Negative Triangle calls, we apply each one of them $\log(\log(n))$ times and take the output of each individual pair of nodes. □

We again don't need to run this reduction on a quantum computer, since we don't require on-the-fly methods or other quantum operations.

We find an interesting divergence in conditional lower bounds between our quantum and classical model in the reduction from All-Pairs Negative Triangle to Negative Triangle.

Theorem 4.2.4 (All-Pairs Negative Triangle $\leq_{Q,n^{2.5},n^{1.5}}$ Negative Triangle)**.** *If* Negative Triangle *can be solved in $O(n^{1.5-\epsilon})$ time by a quantum algorithm on weighted graphs over n nodes with weights from $[-n^c, n^c]$ and for $\epsilon > 0$, then* All-Pairs Negative Triangle *on weighted graphs over n nodes and with weights from $[-n^c, n^c]$ can be solved in $O(\log(n)\log(\log(n))n^{2.5-\epsilon})$ time.*

Proof. Given a weighted graph G over n nodes and with weights from $[-n^c, n^c]$, we apply the reduction from Theorem 3.2.5 to construct $n^{1-\alpha}$ graphs of size n^α for $0 \leq \alpha \leq 1$. If All-Pairs Negative Triangle can be solved in time $T(n)$, then we can solve Negative Triangle in $O(T(n^{1-\alpha})(n^{3\alpha} + n^2))$ time. For $T(n) = n^{1.5}$, this is optimised at $\alpha = \frac{2}{3}$. For each graph we need to make $\log(n)$ calls to the Negative Triangle oracle and to boost the success probability of each call, we apply each one of them $\log(\log(n))$ times. We can solve All-Pairs Negative Triangle in $O(\log(n)\log(\log(n))n^{2.5})$ time. □

Although we prove a lower bound of $n^{1.5-o(1)}$ to Negative Triangle, we use the above algorithm to solve All-Pairs Negative Triangle in $O(n^{2.5})$ time and on-the-fly methods are therefore still not necessary, making the above reduction classically viable.

Where in the classical setting we had cubic lower bounds for all problems from APSP to Δ-Matching Triangles and Triangle Collection, we find a $n^{1.5-o(1)}$ lower bound for Negative Triangle for graphs with n nodes. A lower bound of $n^{1.5-o(1)}$ does not imply that a better lower bound of e.g. $n^{2.5-o(1)}$ does not exist for Negative Triangle. However, a simple Grover Search of all triples of nodes results in a matching upper bound, which *does* imply that a larger lower bound would be inconsistent.

One could argue that it may come as more of a surprise that Negative Triangle is as hard as All-Pairs Negative Triangle or APSP classically than that we find a gap in computational complexity in the quantum case. We look for a single triangle in Negative Triangle and for n^2 potential triangles in All-Pairs Negative Triangle. The quantum model highlights these gaps in difficulty of the problems in a way that the classical model could not. In this sense working in a quantum model on itself is already providing us with useful insights on the complexity of these problems, without requiring that we have a physical quantum algorithm to implement our reductions. An unfortunate consequence of this gap in lower bound complexities is that we lose the reduction from Negative Triangle to $(\min, +)$-Matrix Multiplication, since this reduction 'as-is' does not let us go *up* in complexity. In fact, the definition for fine-grained reductions makes it very challenging to prove computational lower bounds conditioned on smaller computational lower bounds, especially when the instances are similarly structured. To illustrate:

Proving a $n^{2.5-o(1)}$ lower bound for $(\min, +)$-Matrix Multiplication conditioned on a $n^{1.5-o(1)}$ lower bound for Negative Triangle using (quantum) fine-grained reductions would mean that our algorithm cannot query $(\min, +)$-Matrix Multiplication instances of more than $O(n^{.6})$ rows and columns, since $O((n^{.6})^{2.5}) = O(n^{1.5})$. It will be difficult to encode dense graphs into matrices of this size.

We do classically go from a quadratic lower bound for Convolution-3SUM to a cubic lower bound for 0-Weight Triangle, where quadratic is in the number of integers N in the Convolution 3SUM instance and cubic in the number of nodes n for 0-Weight Triangle. By the definition of fine-grained reductions, we can't prove a useful lower bound to 0-Weight Triangle by constructing graphs for our 0-Weight Triangle oracle that have the same number of nodes as there are integers in the Convolution-3SUM instance. The reduction circumvents the obstacle

by constructing $\sqrt{N}$ graphs over $\sqrt{N}$ nodes each. If a graph instance has $\sqrt{n}$ nodes, we can have up to n edges and entries in our weight matrix. Pushing the lower bound up through fine-grained reduction can in this case be explained by the change in parameters in which we measure the complexity.

Less surprising, we find a similar lower bound of $n^{1.5-o(1)}$ for the 0-Weight Triangle problem.

Theorem 4.2.5 (Negative Triangle $\leq_{n^{1.5},n^{1.5}}$ 0-Weight Triangle)**.** *If* 0-Weight Triangle *on graphs over n nodes and with weights from $[-n^c, n^c]$ can be solved in $O(n^{1.5-\epsilon})$ time by a quantum algorithm for some $\epsilon > 0$, then* Negative Triangle *on graphs of n nodes and weights from $[-n^c, n^c]$ can be solved in $O(\log(\log(n))n^{1.5-\epsilon})$ time.*

Proof. Given a weighted directed graph G over n nodes with weights from $[-n^c, n^c]$, we apply the reduction from Theorem 3.2.7 to construct $\log(n^c)$ graphs G_i over n nodes. Since we work in the adjacency matrix model of graphs, am Negative Triangle algorithm has to look over n^2 matrix entries to decide whether a negative triangle exists in one of the graphs. This reduction therefore needs to be on-the-fly. The entries of the adjacency matrix of a graph G_i can be computed in $O(1)$ time. Since we need to boost the individual calls to the 0-Weight Triangle oracle, we can detect a negative triangle using $O(\log(\log(n)))$ calls to a 0-Weight Triangle algorithm. □

Just like in the classical case, there are no reductions from 0-Weight Triangle up the chain of reductions towards APSP or 3SUM. Since 0-Weight Triangle is the meeting point for 3SUM and APSP, having a reduction from 0-Weight Triangle to APSP or 3SUM would imply a reduction between 3SUM and APSP.

The reduction from 0-Weight Triangle to Δ-Matching Triangles highlights some interesting aspects about the complexity of Δ-Matching Triangles.

Theorem 4.2.6 (0-Weight Triangle $\leq_{Q,n^{1.5},n^{1.5}}$ Δ-Matching Triangles)**.** *If* Δ-Matching Triangles *on coloured graphs of n nodes can be solved in $O(n^{1.5-\epsilon})$ time by a quantum algorithm for some $\epsilon > 0$ and $\omega(1) \leq \Delta(n) \leq o(\log(n))$, then* 0-Weight Triangle *on weighted graphs of n nodes and with weights from $[-n^c, n^c]$ can be solved in $O(\log(n)\log\log(n)n^{1.5-\epsilon})$ time by a quantum algorithm.*

Proof. Given a weighted directed graph G over n nodes and using the reduction from the proof of Theorem 3.2.8 we can compute the adjacency

matrices of $2^{O(\Delta)}$ graphs G_i over $O(\Delta n \cdot n^{\frac{c}{\Delta}})$ nodes on-the-fly. We apply a Δ-Matching Triangles algorithm to each of these graphs using on-the-fly computation. Computing the edges of our graphs requires functions computed using Lemma 3.2.2. The functions are computable in $O(2^{\Delta})$ time. It follows that for $\omega(1) \leq \Delta \leq \log(n)$ we can use a Δ-Matching Triangles algorithm to solve 0-Weight Triangle. We boost the $\log(n)$ calls to the Δ-Matching Triangles oracle and arrive at a complexity of $O(\log(n) \log\log(n) n^{1.5})$. □

Corollary 4.2.2. *If* Δ-Matching Triangles *on coloured graphs of* n *nodes can be solved in* $O(n^{1.5-\epsilon})$ *time for some* $\epsilon > 0$ *and* $\omega(1) \leq \Delta(n) \leq o(n^{o(1)})$, *then* 0-Weight Triangle *on weighted graphs of* n *nodes and with weights from* $[-n^c, n^c]$ *can be solved in* $\tilde{O}(n^{1.5-\epsilon})$ *time.*

Proof. Follows from Theorem 4.2.6 and Theorem 3.2.9. □

Here it is less clear whether the lower bound of $O(n^{1.5})$ is the best lower bound we can find for Δ-Matching Triangles. On an intuitive level, Δ-Matching Triangles definitely seems more complex than the single triangle finding problems of Negative Triangle and 0-Weight Triangle. It is important to note that the above reduction only holds for the specified values of Δ: $\omega(1) \leq \Delta(n) \leq n^{o(1)}$. We will see in Chapter 5 that for these ranges of Δ, there is indeed a matching upper bound. This leaves open the question of whether we can find a reduction from 0-Weight Triangle to Δ-Matching Triangles for ranges of Δ that are polynomial or constant in the number of nodes in the graph. For polynomial values of Δ we are faced with the same challenge as in reducing Negative Triangle to $(\min, +)$-Matrix Multiplication in the quantum case: we would be trying to increase the complexity of the lower bound through fine-grained reduction, going from $O(n^{1.5})$ to potentially $O(n^{2.5})$, depending on values of Δ. In Chapter 5 we will see why $O(n^{2.5})$ could be a reasonable quantum lower bound for Δ-Matching Triangles for unrestricted values of Δ.

The reduction from 0-Weight Triangle to Triangle Collection makes use of the construction from the reduction from 0-Weight Triangle to Δ-Matching Triangles for Δ values of $2^{O(\sqrt{\log n})}$, which is in the regime of Δ where we found an $O(n^{1.5})$ lower bound for Δ-Matching Triangles. As a consequence, we get a similar result for Triangle Collection .

Theorem 4.2.7 (0-Weight Triangle $\leq_{Q,n^{1.5},n^{1.5}}$ Triangle Collection)**.** *If* Triangle Collection *can be solved in* $O(n^{1.5-\epsilon})$ *time by a quantum algorithm for some* $\epsilon > 0$, *then* 0-Weight Triangle *can be solved in* $\tilde{O}(\log(n) \log\log(n) n^{1.5-\epsilon})$ *time by a quantum algorithm.*

Proof. Given a weighted directed graph G over n nodes, using the reduction from the proof of Theorem 3.2.10 we can compute the adjacency matrices of $2^{O(\Delta)}$ graphs G_i over $O(\Delta n \cdot n^{\frac{c}{\Delta}})$ nodes on-the-fly. We set $\Delta = 2^{\sqrt{\log(n)}}$ and use a Triangle Collection algorithm to solve 0-Weight Triangle, boosting each of the individual Triangle Collection calls. □

Here the question now really comes down to whether we can find a $O(n^{1.5})$ matching upper bound for Triangle Collection, which we do in Chapter 5.

4.3 Quantum Fine-Grained Reductions from SAT and k-SAT

Through the results from the previous section, and the quantum reductions from 3SUM to 0-Weight Triangle from [13] we now have lower bounds on Δ-Matching Triangles and Triangle Collection from the disjunction of hardness conjectures for APSP and 3SUM. To complete the result, we need to verify the reductions from SAT and k-SAT.

For k-SAT we again go through the sparsification lemma, arriving at the same lower bound we found through 0-Weight Triangle for the same ranges of Δ-Matching Triangles.

Theorem 4.3.1 (k-SAT $\leq_{Q,2^{\delta_k \frac{n}{2}},n^{1.5}}$ Δ-Matching Triangles)**.** *If* Δ-Matching Triangles *can be solved in* $O(n^{1.5-\epsilon})$ *time by a quantum algorithm for some* $\epsilon > 0$ *and* $\omega(1) \leq \Delta(N) \leq N^{O(1)}$*, then* k-SAT *can be solved in* $\tilde{O}(2^{\frac{n}{2}(1-\epsilon')})$ *time for some* $0 < \epsilon' < \frac{2}{3}\epsilon$*.*

Proof. Suppose we have a $O(N^{\frac{3}{2}-\epsilon})$ time Δ-Matching Triangles algorithm for graphs with N nodes and $\omega(1) \leq \Delta(N) \leq N^{o(1)}$ and let ϕ be a k-CNF instance over n variables and m clauses. We apply the sparsification lemma, Lemma 3.3.1, to compute $2^{\epsilon' n}$ k-CNF formulas ϕ_i with cn clauses for some constant value c in $2^{\epsilon' n}$ time. We then apply the reduction from Theorem 3.3.4 to decide whether ϕ_i is satisfiable for each $i \in [2^{\epsilon' n}]$ in $O((\Delta 2^{\frac{n}{3}+\frac{nc}{3\Delta}})^{\frac{3}{2}-\epsilon})$ time. Since $\omega(1) \leq \Delta(N) \leq N^{o(1)}$ it follows that $O((\Delta 2^{\frac{n}{3}+\frac{nc}{3\Delta}})^{\frac{3}{2}-\epsilon}) = O(2^{n(\frac{1}{2}-\frac{\epsilon}{3})+o(1)})$. The total time to evaluate all sparse formulas will be $O(2^{n(\frac{1}{2}-\frac{\epsilon}{3}+\epsilon')+o(1)}) = O(2^{\frac{n}{2}(1-\frac{2}{3}\epsilon+2\epsilon')+o(1)})$. The theorem statement follows for $0 < \epsilon' < \frac{\epsilon}{6}$. For these ranges of ϵ', the number of sparse formulas and the time required to compute them will not exceed the time necessary to solve k-SAT faster than conjectured.

□

For any $\epsilon' < \frac{1}{2}$, the sparsification lemma can easily be applied for quantum reductions from k-SAT. Furthermore, we don't need to explicitly

construct all graphs for our sparse formulas as we did in the proof above. Instead, we can Grover search the set of sparse formulas, applying our Δ-Matching Triangles reduction on-the-fly. In this case, the total run time can be improved to $O(2^{\frac{n}{2}(1-\frac{2\epsilon}{3}+\epsilon')})$, allowing for a wider range of $\epsilon' < \frac{\epsilon}{3}$.

Also the reduction from SAT to Triangle Collection gives us the same lower bound as we found from 0-Weight Triangle.

Theorem 4.3.2 (SAT $\leq_{Q,2^{\delta_k}\frac{n}{2},n^{1.5}}$ Triangle Collection)**.** *If* Triangle Collection *can be solved in* $O(n^{\frac{3}{2}-\epsilon})$ *time by a quantum algorithm for some* $\epsilon > 0$*, then* SAT *can be solved in* $\tilde{O}(2^{\frac{n}{2}(1-\frac{\epsilon}{3})})$ *time.*

Proof. Suppose we have a $O(N^{1.5-\epsilon})$ time algorithm for Triangle Collection on graphs of N nodes. Let ϕ be a CNF formula over n variables and m clauses. We apply the reduction from Theorem 3.3.5 to produce a graph over $O(2^{\frac{n}{3}}m)$ nodes. We can then use our Triangle Collection algorithm to solve SAT in $\tilde{O}(2^{\frac{n}{2}(1-\frac{\epsilon}{3})})$ time. □

Since we find the same lower bounds from 0-Weight Triangle and k SAT to Δ-Matching Triangles and Triangle Collection for the same ranges of Δ, we can now state the quantum equivalent of Theorem 3.4.1.

Theorem 4.3.3. *If* Δ-Matching Triangles *with* $\omega(1) \leq \Delta \leq n^{o(1)}$ *or* Triangle Collection *can be solved in time* $O(n^{1.5-\epsilon})$ *for some* $\epsilon > 0$*, then Conjecture 4.1.1, Conjecture 4.1.3 and Conjecture 4.1.2 must be false.*

Proof. We know that k-SAT reduces to Δ-Matching Triangles for $\omega(1) \leq \Delta \leq n^{o(1)}$ from Theorem 4.3.1 and reduces to Triangle Collection from Theorem 4.3.2. APSP reduces to 0-Weight Triangle from Theorem 4.2.2, Theorem 4.2.3, Theorem 4.2.4 and Theorem 4.2.5 and 3SUM reduces to 0-Weight Triangle as shown in [13]. We then found that 0-Weight Triangle reduces to Δ-Matching Triangles for $\omega(1) \leq \Delta \leq n^{o(1)}$ from Corollary 4.2.2 and to Triangle Collection from Theorem 4.2.7. □

This concludes the main quantum fine-grained reduction result from this thesis. We find all discussed quantum lower bounds alongside their classical counterpart in the figure below.

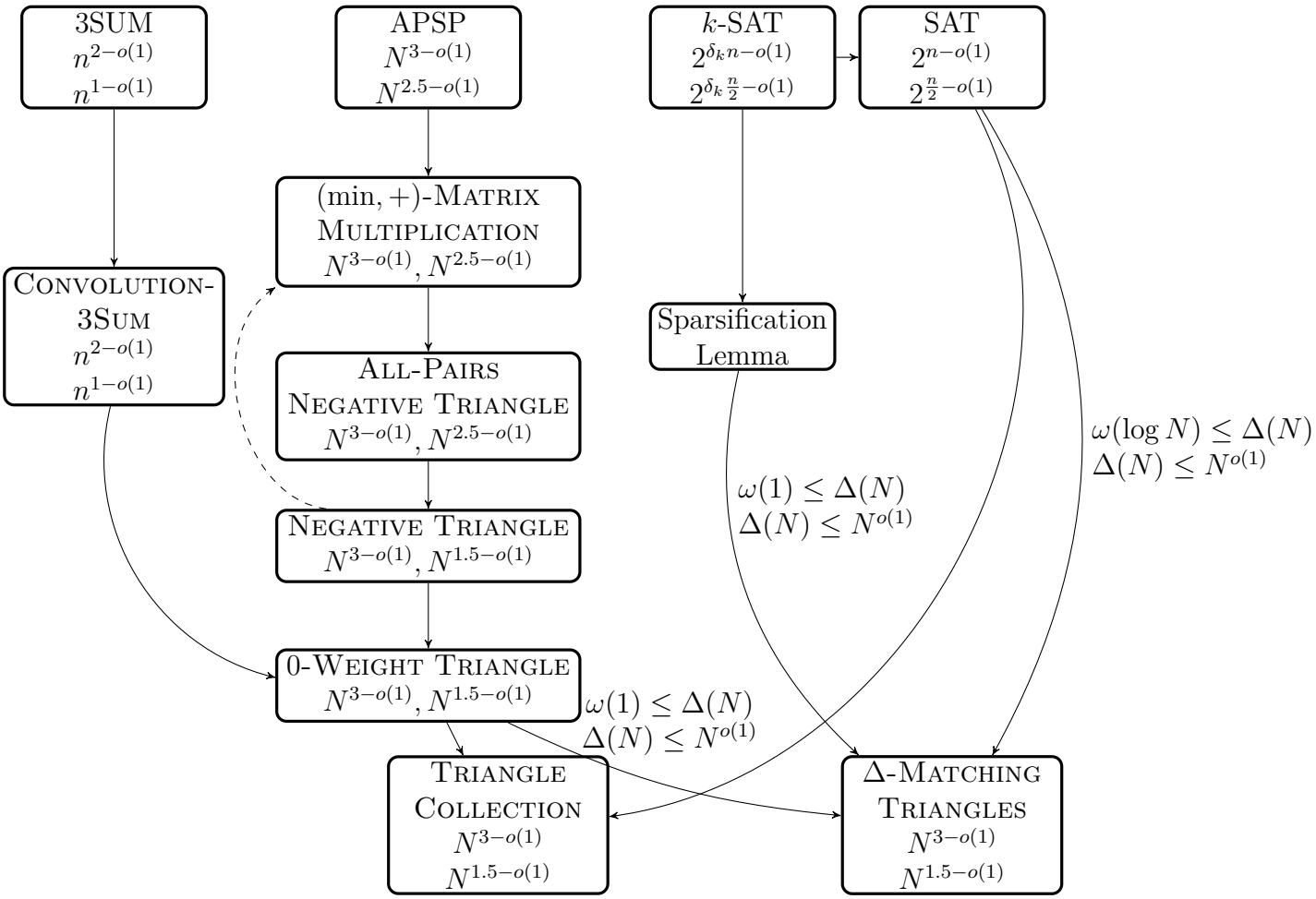

Figure 4.1: Quantum fine-grained reductions to Δ-Matching Triangles and Triangle Collection. The parameter n denotes the size of the sets in the case of 3SUM and CONV-3SUM and the number of variables in the case of SAT and k-SAT. The parameter N denotes the number of nodes in a graph. The first lower bound in each node denotes the known classical lower bound and second lower bound denotes the new quantum lower bound. In the case where multiple edges arrive at one node, the lower bound in the second line of the node is the same for all reductions. For the dashed edge we only know of a classical reduction. Lastly, the lower bounds for Δ-Matching Triangles hold only for the Δ values denoted by the labels on the incoming edges.

Chapter 5

Quantum Upper Bounds

In Chapter 4 we saw a series of lower bounds, conditioned on our quantum hardness conjectures. In this chapter we will provide upper bounds that match these lower bounds by constructing quantum algorithms. We could be done swiftly by constructing two algorithms: One for Δ-Matching Triangles and one for Triangle Collection. If we have an algorithm for these problems that solves it in a time matching their lower bounds, we can apply our reductions to solve inputs from any of the other problems mentioned in the previous chapter.

It can still be useful to construct algorithms in a more direct manner for several reasons. Foremost, many of the reductions used induce logarithmic overheads: we have to binary search integers spaces, evaluate a logarithmic amount of graphs, and boost probabilistic algorithms. Asymptotically, these factors can be largely ignored, but practically they can be very impactful, especially if we use many reductions on one instance, as when we'd want to solve an APSP instance using a Triangle Collection algorithm.

A lot of the direct algorithms are more 'natural' ways of solving a problem than is done through reductions. It is more intuitive to solve a problem directly and this can help us get a better understanding of the problem at hand.

In Chapter 4 we already saw a matching algorithm for APSP and $(\min, +)$-Matrix Multiplication through an application of the Grover Minimum Finding algorithm. For All-Pairs Negative Triangle, Negative Triangle and 0-Weight Triangle we find matching upper bounds through application of Grover Search as well. Since the algorithms are quite straight-forward, we briefly sketch them here.

For All-Pairs Negative Triangle we simply run a Grover Search for each pair of nodes for which we need to check for a negative triangle, requiring $O(n^2)$ applications of a $O(\sqrt{n})$ time Grover Search. We boost each of the $O(n^2)$ Grover Search subroutines to solve All-Pairs Negative Triangle in $O(\log^2(n)n^{2.5})$ time.

For Negative Triangle and 0-Weight Triangle we can simply Grover Search the set of all triples of nodes. Since the checking time to determine whether a given triple of nodes is the right type of triangle we're looking for takes $O(1)$ time due to our RAM capacity, both algorithms require $O(\sqrt{n^{1.5}})$ time in the quantum setting.

5.1 Delta Matching Triangles and Triangle Collection

The algorithms for solving Δ-Matching Triangles and Triangle Collection are not as straight forward as those from the previous section. We are not looking for a single triangle, yet here we find quite a large speed-up, going from $O(n^3)$ to $O(n^{1.5})$. Both algorithms rely heavily on the Variable Time Grover Search algorithm from Theorem 2.2.4 in Chapter 2, as suggested by Ambainis [5].

Theorem 5.1.1. *There exists a quantum algorithm that solves* Δ-Matching Triangles *on graphs of* n *nodes in* $O(\min\{n^{1.5+\frac{\alpha}{2}}, n^{1.5+\omega-\frac{\alpha}{2}}\})$ *time for* $\Delta = n^\alpha$.

Proof. Let $G = (V, E)$ be a colored graph with $|V| = n$ and colors given by $\gamma : E \to \Gamma$. We want to search the set of all possible color triples using Variable Time Grover Search. To determine the time it takes for checking whether a single triple of colors contains Δ triangles, we will use a combination of two different approaches. First, let $\Delta = n^\alpha$ for $0 \leq \alpha \leq 3$. We apply VTGS to the set of color triples Γ^3 and to determine the VTGS 'checking time' for a single color triple, we use one of two different techniques, dependent on the value of α. For small α we will want to use regular Grover Search, while for large α, we will use matrix multiplication. Given a color i, let $V_i \subseteq V$ be the subset containing only nodes in that color and let $|V_i| = \Gamma_i$ be the color size.

Small α: We use VTGS on the set Γ^3. Given a triple of colors $(i, j, k) \in \Gamma^3$, let the time to determine whether G contains n^α triangles in that color be $t_{i,j,k}$. Using Grover Search we have $t_{i,j,k} = \sqrt{n^\alpha \Gamma_i \Gamma_j \Gamma_k}$. Since VTGS takes time $T(n) = O(\sqrt{\sum_{i,j,k \in \Gamma} t_{i,j,k}^2})$ and $\sum_{i \in \Gamma} \Gamma_i = n$ we find the following:

$$\begin{aligned} T(n) &= O(\sqrt{\sum_{i,j,k\in\Gamma} t_{i,j,k}^2}) \\ &= O(\sqrt{\sum_{i,j,k\in\Gamma} n^\alpha \Gamma_i\Gamma_j\Gamma_k}) \\ &= O(\sqrt{n^\alpha \sum_{i\in\Gamma}\Gamma_i \sum_{j\in\Gamma}\Gamma_j \sum_{k\in\Gamma}\Gamma_k}) \\ &= O(\sqrt{n^\alpha n^3}) = O(n^{1.5+\frac{\alpha}{2}}). \end{aligned}$$

Large α: First we notice that as α increases, the amount of color triples for which there are even enough nodes in the graph to form n^α triangles decreases. Before performing VTGS, we filter out all colors triples (i,j,k) such that $\Gamma_i\Gamma_j\Gamma_k < \Delta$. We need to be able to this efficiently, and we achieve that in the following way: First enumerate all nodes and all colors in G. Then construct the $n \times n$ matrix M such that $M[uv] = 1$ if the color of the v'th node is u and $M[uv] = 0$ otherwise. This will take time n^2. Now we can Grover Search the set of color triples Γ^3 to filter out all the small triples. Checking whether a triple is large enough takes constant time and there are at most $\frac{n^3}{n^\alpha}$ such triples. Grover Search will therefore take at most $O(\sqrt{n^{3-\alpha}n^3}) = O(n^{4.5-1.5\alpha})$ time.

Let $\Gamma' \subseteq \Gamma^3$ denote the set of all large enough color triples. To check whether there are n^α triangles in G for color triple (i,j,k) we apply matrix multiplication to the adjacency matrix of the sub-graph induced by $V_i \cup V_j \cup V_k$. First we'd have to construct this matrix, which takes time$(\Gamma_i + \Gamma_j + \Gamma_k)^2$. Let $M_{i,j,k}$ be the adjacency matrix of the induced sub-graph. Then to count the number of triangles, we compute $Tr[M^3]$ in $O((\Gamma_i + \Gamma_j + \Gamma_k)^\omega)$ time.

For VTGS we then find:

$$\begin{aligned} T(n) &= O(\sqrt{\sum_{i,j,k\in\Gamma} t_{i,j,k}^2}) \\ &= O(\sqrt{\sum_{i,j,k\in\Gamma} (\Gamma_i + \Gamma_j + \Gamma_k)^{2\omega}}) \\ &\leq O(\sqrt{\sum_{i,j,k\in\Gamma} n^{2\omega}}) \\ &\leq O(\sqrt{n^{3-\alpha}n^{2\omega}}) = O(n^{1.5+\omega-\frac{\alpha}{2}}). \end{aligned}$$

The complexity for large α will then be $O(n^{1.5+\omega-\frac{\alpha}{2}} + n^{4.5-1.5\alpha})$. Then from

$$1.5 + \omega - \frac{\alpha}{2} = 4.5 - 1.5\alpha$$

we have that whenever $\alpha \geq 3 - \omega$ the complexity becomes $O(n^{1.5+\omega-\frac{\alpha}{2}})$.

Combined approach: Given an instance of Δ-Matching Triangles, we first compute which of the two approaches is faster based on $\Delta = n^\alpha$ and then apply that approach. We find a complexity of:

$$T(n) = O(\min\{n^{1.5+\frac{\alpha}{2}}, n^{1.5+\omega-\frac{\alpha}{2}} + n^{4.5-1.5\alpha}\})$$

Setting

$$1.5 + \frac{\alpha}{2} = 1.5 + \omega - \frac{\alpha}{2}$$

we see that for $\alpha < \omega$ the small α algorithm will be faster while for $\alpha > \omega$ our large algorithm is faster. Combined with the fact that in the regime where $\alpha > \omega$ and assuming that $\omega \geq 2$, we have that the complexity for large α is $O(n^{1.5+\omega-\frac{\alpha}{2}})$. We find a final complexity of

$$O(\min\{n^{1.5+\frac{\alpha}{2}}, n^{1.5+\omega-\frac{\alpha}{2}}\})$$

□

From Theorem 5.1.1 we get a worst case corollary and a corollary for the range of Δ for which we found reductions in Chapter 4.

Corollary 5.1.1. *There exists a quantum algorithm that solves* Δ-Matching Triangles *on graphs of n nodes in $O(n^{1.5+\frac{\omega}{2}})$ time for any Δ.*

Corollary 5.1.2. *There exists a quantum algorithm that solves* Δ-Matching Triangles *on graphs of n nodes in $O(n^{1.5})$ time for $\omega(1) \leq \Delta \leq n^{o(1)}$.*

The commonly conjectured lower bound for ω is 2, in which case we have a worst case complexity for Δ-Matching Triangles of $O(n^{2.5})$, matching the quantum complexity of other problems encountered in our reductions from APSP. That is not to say that a faster algorithm is not possible of course. Pushing the $O(n^{1.5})$ lower bound for Δ-Matching Triangles up for polynomial Δ is challenging with current techniques and a matching upper bound of $O(n^{1.5})$ for unrestricted Δ is not impossible, albeit unlikely.

Using a similar strategy as for the case of small Δ in the proof of Theorem 5.1.1 we also find a matching upper bound for Triangle Collection.

Theorem 5.1.2. *There exists a quantum algorithm that solves* Triangle Collection *in $O(n^{1.5})$ time.*

Proof. Let $G = (V, E)$ be a colored graph with $|V| = n$ and colors given by $\gamma : E \to \Gamma$. We apply the Variable-Time Grover Search algorithm to the set of all color triples Γ^3, to determine whether there is a color triple for which there is no triangle in G. Let (i, j, k) be a triple of colors and let the time it takes to check whether G contains a triangle in these colors be $t_{i,j,k}$. Then VTGS takes $T(n) = O(\sqrt{\sum_{i,j,k\in\Gamma} t_{i,j,k}^2})$ time. Given a color i, let $V_i \subseteq V$ be the subset containing only nodes in that color and let $|V_i| = \Gamma_i$ be the color size. For any triple of colors (i, j, k), we can Grover Search the set $V_i \times V_j \times V_k$ to determine whether there is a triangle in the induced sub-graph in $\sqrt{\Gamma_i\Gamma_j\Gamma_k}$ time, and we have that $t_{i,j,k} = \Gamma_i\Gamma_j\Gamma_k$. Since it holds that $\sum_{i\Gamma} \Gamma_i = n$ it follows that

$$\begin{aligned} T(n) &= O(\sqrt{\sum_{i,j,k\in\Gamma} t_{i,j,k}^2}) \\ &= O(\sqrt{\sum_{i,j,k\in\Gamma} \Gamma_i\Gamma_j\Gamma_k}) \\ &= O(\sqrt{\sum_{i\in\Gamma} \Gamma_i \sum_{j\in\Gamma} \Gamma_j \sum_{k\in\Gamma} \Gamma_k}) \\ &= O(\sqrt{n^3}) = O(n^{1.5}). \end{aligned}$$

We can solve Triangle Collection in $O(n^{1.5})$ time. □

Both the reductions to Triangle Collection and the direct algorithm for solving Triangle Collection decide Triangle Collection by solving the negation of the problem: In the reductions we accept a 0-Weight Triangle or SAT instance using a Triangle Collection oracle if and only if the Triangle Collection oracle rejects. In the algorithm from Theorem 5.1.2 we solve a Triangle Collection instance by looking for a triple of colors for which there is no triangle. The Triangle Collection problem comes down to a single-triangle finding problem, which may be what explains the low computational complexity of the problem.

Chapter 6

Conclusion

6.1 Discussion

In this thesis we have contributed a natural continuation to the recent work done in quantum fine-grained computational complexity by formulating a quantum hardness conjecture for APSP. There are many classical fine-grained reductions from APSP, but in order to connect to previous work done in quantum fine-grained complexity, we decided to investigate reductions connected to the other quantum hardness conjectures. In doing so, we found many conditional lower bounds and were able to make some interesting observations.

Classically, it was surprising to find in Chapter 3 that more seemingly simple problems such as Negative Triangle and 0-Weight Triangle are at least as hard as more seemingly complex problems like APSP and $(\min, +)$-Matrix Multiplication. In the quantum framework in Chapter 4 we found that our intuition regarding the difference in complexity of these problems is to a reasonable degree well-founded: In the quantum setting we found a quadratic gap in complexity in both the lower and the upper bounds of these problems. A consequence of this gap in complexity was the loss of the reduction from Negative Triangle back to $(\min, +)$-Matrix Multiplication.

In Chapter 5 we found that many other computational problems find natural speed-ups through Grover's Search algorithm, with Δ-Matching Triangles and Triangle Collection finding speed-ups through Variable-Time Grover Search. In a result that ties in well with the lower bound results, we have a $O(n^{1.5})$ upper bound for Δ-Matching Triangles that works for those exact ranges of Δ for which the reduction works from all three hardness conjectures.

Overviews of all the results are nicely presented in Table 1.1 and Figure 4.1, with the complexity gaps clearly visible.

6.2 Future Work

There are many opportunities for future research in quantum fine-grained computational complexity.

To pick up where this work has left off would be to face the challenges of finding reductions that prove a lower bound that is higher than the lower bound on which it is conditioned, as in the case of quantum reducing Negative Triangle to $(\min, +)$-Matrix Multiplication.

The Δ-Matching Triangles problem presents another series of worthwhile challenges. Due to the dependency of the upper bound on ω, faster algorithms for Δ-Matching Triangles can definitely be found, either by improving ω, or by finding an algorithm with no or lower dependency on ω. The question remains then whether there exists quantum algorithms that have a better than $O(n^{2.5})$ worst case Δ complexity, which would be the worst case Δ complexity if $\omega = 2$. To reinforce the likelihood of $O(n^{2.5})$ being the best worst case Δ upper bound for Δ-Matching Triangles, we could look for reductions to Δ-Matching Triangles that work for polynomial ranges of Δ. In the classical setting, we saw in [3] that for constant ranges of Δ, Δ-Matching Triangles permits a faster than cubic algorithm and it leads us to wonder whether a faster than $O(n^{1.5})$ algorithm exists for Δ-Matching Triangles with constant Δ in the quantum case. The Δ-Matching Triangles and Triangle Collection are of course of special interest due to their connection to all three popular hardness conjectures.

Aside from picking up where this work leaves off, much work remains to be done in quantum fine-grained complexity as a whole. An easy starting point is the overview of classical reductions from [52], which points to many classical reductions that can be reviewed in the quantum case, in the effort to construct a quantum complexity web of reductions that mirrors the classical complexity web.

In [15] many computational lower bound were proven using truth table properties of computational problems. Some reductions preserve or alter in some computable way properties such as the parity or count of solutions. These properties provide us with an additional tool for proving lower bounds, possibly letting us overcome the challenge of increasing lower bounds through reductions as mentioned in the first paragraph of this section.

Appendix A

Graph Definitions

Here we collect all the definitions relating to graphs that occur in the thesis.

Definition A.0.1 (Graphs)**.** A graph G is an ordered pair (V, E) with a vertex set V of vertices (or nodes) and an edge set E of unordered pairs of vertices called edges.

- The size of a graph comes in two parameters, with $n = |V|$ and $m = |E|$.
- A graph is sparse if $O(|V|) = O(|E|)$.
- A graph is dense if $O(|V|^2) = O(|E|)$.
- A directed graph is a graph where the edge set consists of ordered pairs of vertices.
- A weighted (directed) graph is a triple (V, E, w) such that (V, E) is a (directed) graph and $w : E \to W$ a weight function for some set of weights W.
- A coloured graph is a triple (V, E, c) such that (V, E) is a (directed) graph and $c : V \to L$ is a function to a set of colours or labels L.

As opposed to an ordered triple, the information in a graph can also be stored using adjacency and weight matrices.

Definition A.0.2 (Adjacency Matrix)**.** Let $G(V, E)$ be a graph, with $V = [|V|]$.

- The adjacency matrix A of G is a $|V| \times |V|$ matrix such that $A_{ij} = 1$ if $(i, j) \in E$ and $A_{ij} = 0$ else.
- If G is a weighted graph and w is its weight function, the weight matrix W of G is a $|V| \times |V|$ matrix such that $W_{ij} = w((i, j))$ if $(, ij) \in E$ and $W_{ij} = \infty$ else.

Definition A.0.3 (Trees)**.** Let $T(V, E)$ be a graph.

- The graph T is a tree if for any two vertices in T there is exactly one path.
- The graph T is a rooted tree if T is a directed tree and there is a node, called the root, that has only outgoing edges.

Bibliography

[1] S. Aaronson, N.-H. Chia, H.-H. Lin, C. Wang, and R. Zhang. "On the quantum complexity of closest pair and related problems". In: *Proceedings of the 35th Computational Complexity Conference*. CCC '20. July 2020, pp. 1–43.

[2] A. Abboud, K. Lewi, and R. Williams. "Losing weight by gaining edges". In: *Algorithms - ESA 2014*. Ed. by A. S. Schulz and D. Wagner. Lecture Notes in Computer Science. 2014, pp. 1–12.

[3] A. Abboud, V. V. Williams, and H. Yu. "Matching triangles and basing hardness on an extremely popular conjecture". In: *Siam journal on computing* 47.3 (Jan. 2018), pp. 1098–1122.

[4] J. Alman and V. V. Williams. "A refined laser method and faster matrix multiplication". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2021, pp. 522–539.

[5] A. Ambainis. personal communication. 2022.

[6] A. Ambainis. "Quantum search with variable times". In: *Theory of computing systems* 47.3 (Oct. 2010), pp. 786–807.

[7] A. Ambainis and N. Larka. "Quantum algorithms for computational geometry problems". In: *15th conference on the theory of quantum computation, communication and cryptography (tqc 2020)*. Ed. by S. T. Flammia. Vol. 158. Leibniz International Proceedings in Informatics (LIPIcs). 2020, 9:1–9:10.

[8] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. 2009.

[9] I. Baran, E. D. Demaine, and M. Pătraşcu. "Subquadratic algorithms for 3SUM". In: *Algorithmica* 50.4 (Apr. 2008), pp. 584–596.

[10] R. Bellman. "On a routing problem". In: *Quarterly of applied mathematics* 16.1 (1958), pp. 87–90.

[11] D. Bremner, T. M. Chan, E. D. Demaine, J. Erickson, F. Hurtado, J. Iacono, S. Langerman, M. Pătraşcu, and P. Taslakian. "Necklaces, convolutions, and X+Y". In: *Algorithmica* 69.2 (June 2014), pp. 294–314.

[12] J. Buhler, H. Lenstra, and C. Pomerance. "Factoring integers with the number field sieve". In: *The development of the number field sieve (edited with A. K. Lenstra), Lecture Notes in Mathematics 1554*. Nov. 2006, pp. 50–94.

[13] H. Buhrman, B. Loff, S. Patro, and F. Speelman. "Limits of quantum speed-ups for computational geometry and other problems: fine-grained complexity via quantum walks". In: *13th innovations in theoretical computer science conference (itcs 2022)*. Ed. by M. Braverman. Vol. 215. Leibniz International Proceedings in Informatics (LIPIcs). 2022, 31:1–31:12.

[14] H. Buhrman, B. Loff, S. Patro, and F. Speelman. "Memory compression with quantum random-access gates". In: *Arxiv:2203.05599 [quant-ph]* (Mar. 2022). arXiv: 2203.05599.

[15] H. Buhrman, S. Patro, and F. Speelman. "A framework of quantum strong exponential-time hypotheses". In: *38th international symposium on theoretical aspects of computer science (stacs 2021)*. Ed. by M. Bläser and B. Monmege. Vol. 187. Leibniz International Proceedings in Informatics (LIPIcs). 2021, 19:1–19:19.

[16] T. M. Chan. "More logarithmic-factor speedups for 3SUM, (median,+)-convolution, and some geometric 3SUM-hard problems". In: *Acm transactions on algorithms* 16.1 (Nov. 2019), 7:1–7:23.

[17] S. A. Cook and R. A. Reckhow. "Time bounded random access machines". In: *Journal of computer and system sciences* (1973), pp. 354–375.

[18] M. Cygan, H. Dell, D. Lokshtanov, D. Marx, J. Nederlof, Y. Okamoto, R. Paturi, S. Saurabh, and M. Wahlström. "On problems as hard as CNF-SAT". In: *Acm transactions on algorithms* 12.3 (May 2016), 41:1–41:24.

[19] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*. 2016, p. 629.

[20] M. Cygan, M. Mucha, K. Węgrzycki, and M. Włodarczyk. "On problems equivalent to (min,+)-convolution". In: *Acm transactions on algorithms* 15.1 (Jan. 2019), pp. 1–25.

[21] H. Dell, T. Husfeldt, D. Marx, N. Taslaman, and M. Wáhlen. "Exponential time complexity of the permanent and the Tutte polynomial". In: *Acm transactions on algorithms* 10.4 (Aug. 2014), pp. 1–32.

[22] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (Dec. 1959), pp. 269–271.

[23] C. Dürr, M. Heiligman, P. Hoyer, and M. Mhalla. "Quantum query complexity of some graph problems". In: *Siam journal on computing* 35.6 (Jan. 2006), pp. 1310–1328.

[24] C. Dürr and P. Hoyer. "A quantum algorithm for finding the minimum". In: *Corr* quant-ph/9607014 (July 1996).

[25] M. J. Fischer and A. R. Meyer. "Boolean matrix multiplication and transitive closure". In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. ISSN: 0272-4847. Oct. 1971, pp. 129–131.

[26] R. W. Floyd. "Algorithm 97: shortest path". In: *Communications of the acm* 5.6 (June 1962), p. 345.

[27] A. Gajentaan and M. H. Overmars. "On a class of $O(n^2)$ problems in computational geometry". In: *Computational geometry* 5.3 (Oct. 1995), pp. 165–185.

[28] V. Giovannetti, S. Lloyd, and L. Maccone. "Quantum random access memory". In: *Phys. rev. lett.* 100 (16 2008), p. 160501.

[29] L. K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*. 1996, pp. 212–219.

[30] A. Grønlund and S. Pettie. "Threesomes, degenerates, and love triangles". In: *Journal of the acm* 65.4 (Apr. 2018), 22:1–22:25.

[31] J. Hershberger, M. Maxel, and S. Suri. "Finding the k shortest simple paths: A new algorithm and its implementation". In: *Acm transactions on algorithms* 3.4 (Nov. 2007), p. 45.

[32] R. Impagliazzo. "A personal view of average-case complexity". In: *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*. 1995, pp. 134–147.

[33] R. Impagliazzo and R. Paturi. "On the complexity of k-SAT". In: *Journal of computer and system sciences* 62.2 (Mar. 2001), pp. 367–375.

[34] R. Impagliazzo, R. Paturi, and F. Zane. "Which problems have strongly exponential complexity?" In: *Journal of computer and system sciences* 63.4 (Dec. 2001), pp. 512–530.

[35] D. B. Johnson. "Efficient algorithms for shortest paths in sparse networks". In: *Journal of the acm* 24.1 (Jan. 1977), pp. 1–13.

[36] V. Kabanets. "Derandomization: A brief overview". In: *In Electronic Colloquium on Computational Complexity, technical report, TR*. 2002, pp. 02–008.

[37] I. Munroe. "Efficient determination of the transitive closure of a directed graph". In: *Information processing letters* 1 (1971). ISSN: 0020-0190, pp. 56–58.

[38] M. Naya-Plasencia and A. Schrottenloher. "Optimal merging in quantum k-xor and k-sum algorithms". In: *Advances in Cryptology – EUROCRYPT 2020*. Ed. by A. Canteaut and Y. Ishai. Lecture Notes in Computer Science. 2020, pp. 311–340.

[39] A. Nayebi and V. V. Williams. "Quantum algorithms for shortest paths problems in structured instances". In: *Arxiv:1410.6220 [quant-ph]* (Oct. 2014). arXiv: 1410.6220.

[40] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. 10th anniversary ed. 2010.

[41] M. Patrascu. "Towards polynomial lower bounds for dynamic problems". In: *Proceedings of the annual acm symposium on theory of computing* (June 2010), pp. 603–610.

[42] R. Paturi, P. Pudlak, M. E. Saks, and F. Zane. "An improved exponential-time algorithm for k-SAT". In: *Journal of the acm* (2005), p. 28.

[43] L. Roditty and U. Zwick. "On dynamic shortest paths problems". In: *Algorithmica* 61.2 (Oct. 2011), pp. 389–401.

[44] D. Sheridan. "The optimality of a fast CNF conversion and its use with sat". In: *The seventh international conference on theory and applications of satisfiability testing* (2004), p. 6.

[45] P. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Nov. 1994, pp. 124–134.

[46] S. S. Skiena. *The algorithm design manual*. 2nd. 2018.

[47] T. B. de la Tour. "An optimality result for clause form translation". In: *Journal of symbolic computation* 14 (4 1992), pp. 283–301.

[48] S. Warshall. "A theorem on boolean matrices". In: *Journal of the acm* 9.1 (Jan. 1962), pp. 11–12.

[49] R. R. Williams. "Faster All-Pairs Shortest Paths via circuit complexity". In: *Siam journal on computing* 47.5 (Jan. 2018), pp. 1965–1985.

[50] R. Williams. "A new algorithm for optimal 2-constraint satisfaction and its implications". In: *Theoretical computer science*. Automata, Languages and Programming: Algorithms and Complexity 348.2 (Dec. 2005), pp. 357–365.

[51] V. V. Williams. "Hardness of easy problems: basing hardness on popular conjectures such as the strong exponential time hypothesis". In: *The international symposium on parameterized and exact computation* (2018), p. 14.

[52] V. V. Williams. "On some fine-grained questions in algorithms and complexity". In: *Proceedings of the International Congress of Mathematicians (ICM 2018)*. May 2019, pp. 3447–3487.

[53] V. V. Williams and R. R. Williams. "Subcubic Equivalences Between Path, Matrix, and Triangle Problems". In: *Journal of the acm* 65.5 (Sept. 2018), pp. 1–38.

[54] V. V. Williams and R. Williams. "Finding, minimizing, and counting weighted subgraphs". In: vol. 42. Jan. 2009, pp. 455–464.

[55] R. de Wolf. *Quantum Computing: Lecture Notes*. arXiv: 1907.09415. Jan. 2022.

[56] J. Y. Yen. "Finding the K shortest loopless paths in a network". In: *Management science* 17.11 (1971), pp. 712–716.