COMPLEXITY BOUNDS ON SOME FUNDAMENTAL COMPUTATIONAL PROBLEMS FOR QUANTUM BRANCHING PROGRAMS

Dissertation

zur

erlangung des doktorgrades (dr. rer. nat.)

der

mathematisch-naturwissenschaftlichen fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Airat Khasianov

aus Kazan

Bonn, Mai 2005

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Erstgutachter (Betreuer): Prof. Dr. Marek Karpinski
Zweitgutachter: Prof. Dr. Farid Ablayev

Tag der Promotion: Juli 14, 2005

Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn *http://hss.ulb.uni-bonn.de/diss_online* elektronisch publiziert, 2005.

Preface

Quantum computing was not invented at once all of a sudden. Disciplines that split and specialized last hundred years converge again today. That was how computer scientists came about to think physically, or, perhaps, physicists – computationally. Hopefully, this common tendency of the contemporary science will lead to a better and simpler description of the world we live in.

In 1994, Peter Shor discovered polynomial time quantum algorithms for factoring integers and for computing discrete logarithms [Sho94]. It was this discovery that put the field of *quantum computations* in the public spotlight. Later, in 1997, Lov Grover introduced a quantum search algorithm, that could find a "needle" in a "haystack" of size n in only square root of n queries [Gro97]. Clearly, even a randomized algorithm would need to make at least $n/2$ queries in order to achieve reasonably high probability to find the "needle".

The hearts of computer scientists were won over. They massively read physics textbooks, and spend their spare time performing Fourier transforms, involved in the Shor's algorithm. However, the quantum algorithms demanded people to "think different", as Apple™ has been suggesting in their promotion campaign. Not only was it difficult to design efficient quantum algorithms, but also engineering of quantum devices proved hard. The most elaborate quantum computer designed so far operates only five qubits of memory. Using this computer, the IBM™ Almaden Research Center group lead by Isaac Chuang successfully factored 15 into 5×3.

The intrinsic hardness of quantum computers caused the research efforts shift to more restricted models of computation. Those that would be less demanding in formal implementation. Ambainis and Freivalds already in 1998 speculated that first

quantum computers would perhaps have relatively simple *quantum mechanical* part connected to a full-functional *classical* computer [AF98].

In this work we consider one of the most popular restricted computational model called a *quantum branching program.*

The branching program model has been around for half a century. But it was not before 1986, that R. Bryant [Bry86] improved the model to introduce what is now called the *Oblivious Ordered Read-once Branching Program*, or more recognizably OBDD (*Ordered Binary Decision Diagram*). Apart from that it is a computational paradigm, OBDD turned out to be a very useful type of *boolean functions* representation. There are several reasons for that.

1. Standardized OBDD provide a canonical representation of Boolean functions;
2. Logical operations can be efficiently performed over Boolean functions represented by reduced OBDD;
3. Most of practically useful Boolean function have concise OBDD representations.

No surprise that OBDD proved extremely useful in CAD/CAM (Computer Aided Design/Manufacturing) applications. Nowadays there can be no VLSI (Very Large Scale Integration) production imagined, where OBDD based technology would not be applied. The book of Ch. Meinel and Th. Theobald [MT98] is dedicated solely to the OBDD role in the VLSI design. Branching programs are presented as an established mathematical subject in the monograph of I. Wegener [Weg00].

In 1996 F. Ablayev and M. Karpinski introduced *Randomized Branching Programs* [AK96]. They constructed a function computed by a polynomial size *Randomized Branching Program*, such that no polynomial size *Deterministic Branching Program* existed for the function. Five years later, after *Randomized Branching Programs* become a classical paradigm, the same authors and A. Gainutdinova introduced *Quantum Ordered Binary Decision Diagram* [AGK01]. They demonstrated that the quantum model can be exponentially more efficient than its classical counterpart for an explicit function.

We chose the model of *quantum* OBDD for our research, because

- Quantum branching programs describes actual physical processes;
- The model is also more adequate than other approaches to the current state-of-the-art in the fabrication of quantum computatinal devices;
- The quantum system used for computation is allowed to consist of only a *sub-linear* number of qubits, which is impossible for *quantum circuits*;
- This model allows construction of efficient algorithms;
- It is one of the most important classes of restricted computational models. Comparing its power to that of OBDD we better understand advantages over conventional approaches that quantum mechanics offers.

In the context of quantum branching programs, we study several computational problems. We start by presenting an algorithm for the *Equality* function. Then we show how to extend our results, gradually advancing towards the algorithm that computes the *decision* version of the *hidden subgroup problem.* Our proofs use the *fingerprinting technique* that dates back to 1979 [Fre79]. It was used for *quantum automata* in the 1998 paper by A. Ambainis and R. Freivalds [AF98].

We prove lower bounds for all the problems we consider. The lower and the upper bounds match for all of the functions except for the *hidden subgroup test.* In the latter case the lower bounds asymptotically equal the upper bounds in the worst-case choice of the function parameters. Performance of the algorithm we present, in turn, does not depend on the internal structure of the considered group. Our lower bound proofs are based on the *communication complexity* approach and results of Ablayev, Gainutdinova and Karpinski [AGK01].

The choice of the problems was motivated by the fact that the *factoring* and *discrete logarithm* problems, we mentioned earlier, can be formulated in terms of the hidden subgroup problem. All those problems are efficiently solved by *quantum computers* but no *efficient* classical, even randomized, algorithm is known for them so far. Thus, these problems are the best candidates for the witnesses of quantum computers superiority over the classical counterparts. As Scot Aaronson put it in his theses [Aar04a]:

> Either the Extended Church-Turing Thesis is false, or quantum mechanics must be modified, or the factoring problem is solvable in classical polynomial time. All three possibilities seem like wild, crackpot speculations but at least one of them is true!

The thesis given by Aaronson provides one more motive to study quantum computers. Whatever ultimately will be discovered concerning quantum computers, it will exert tremendous influence on the whole Science!
Our little research falls short in making that ultimate discovery. However, it is novel in several ways.

- None of the problems we study was considered in the context of *quantum branching programs* before.
- We prove original lower and upper bounds for all of the considered computational problems.
- Additionally a tight communication complexity lower bound is proved for the *hidden subgroup problem test* function.
- This research unveils an interesting connection of a simple function, like *Equality*, to the more elaborate *hidden subgroup problem*.
- All algorithms of our research are shown to work with relatively *"simple"* quantum states, that have already been demonstrated experimentally.
- We consider unrestricted *non-abelian* version of the *hidden subgroup problem*, thus, we also solve the *graph isomorphism* within the same complexity bounds.
- This research also offers rich opportunities for generalization of the obtained results by means of the reductions that we discuss.

Let us briefly review the structure of this thesis, and give a list of the chapters.

Introduction to computer science

In this chapter we tell a story of emergence and evolution of what we now call an *algorithm*. Then we introduce basic notions and concepts that belong to the computer science "common speech".

Classical models of computations

The Turing machine is the most fundamental, and maybe famous, mathematical definition of an algorithm. We use it to define basic *complexity measures*. We also present the *linear speedup theorem* that is crucial to how we treat the *complexity measures* in this research. Later we use the development of the Turing machine to show how other classical computational models evolved into their non-classical variants.
Done with the deterministic Turing machine, we define the *branching program*, the computational model of central interest in this thesis. We define its most important subclasses and introduce relevant complexity measures. Finally, we consider the *communication model*. We apply the *one-way communication complexity* techniques to prove the lower bounds later in the thesis.

Nondeterministic and randomized models

We describe how to formalize a computation that makes errors with a certain probability. That is, *probabilistic models of computation.* We first demonstrate the approach on the *Turing machines*. There, we introduce important probabilistic techniques. We also define *probabilistic complexity classes*. Eventually we apply the approach to the *branching programs.* We obtain different classes of *randomized branching programs* and define corresponding complexity classes. That prepares us to the "main" computational model of this research – the *quantum branching programs.*

Quantum computations

First we take a detour to make a glance at the fascinating history of the *quantum mechanics* creation. Then we introduce the *postulates* of quantum mechanics and

basic techniques. At last, we present the *quantum branching programs*, the main tool of this research.

The hidden subgroup related problems

This chapter tells about results we obtained for several fundamental functions, on our quest to identify the complexity of the *hidden subgroup problem* for the *quantum branching programs*. We start with *Equality function* and proof linear upper and lower bounds on the width of the *read-once quantum branching programs* that represent this function. We extend this result to *Periodicity* and simplified *Simon* function. Gradually generalizing the technique in order to apply it to the *hidden subgroup test*, the decision version of the *hidden subgroup problem*, in the next chapter.

The hidden subgroup problem

At length, we are able to tackle the main motivation of this research – *the hidden subgroup problem*. First we prove linear upper bound on the width of *read-once quantum branching programs* that represent the *hidden subgroup test*. Then we prove that this upper bound is almost tight. Apart from the quantum OBDD lower bound, a one-way communication complexity lower bound is also proved. The multiple lower bounds from this chapter are intended to provide possibly complete picture of the hidden subgroup test complexity.

We conclude this chapter by showing that our algorithms use only the quantum states that have already been demonstrated in a laboratory. This contrasts, for example, to the original Shor's algorithm, where quantum states are a matter of controversy concerning feasibility of the algorithm.

Reducibility theory

In this chapter we give several possibilities to generalize the results obtained in this thesis. We consider two different reduction concepts: *rectangular reductions* and *polynomial projections* suitable for that purpose.

Appendices

Here, the most frequent notation is presented. Also one can find a note on Chernoff bound met more than once in the thesis. A standard illustration of the complexity classes relations is also given.

Acknowledgement

I would like to thank Prof. Marek Karpinski for fruitful conversations on the Hidden Subgroup Problem. The upper bound theorem is an outcome of this collaboration. I would like to thank Prof. Farid Ablayev for an exceptional collaboration that lead to the lower bound results proven in this dissertation. I was lucky to be a member of the Bonn International Graduate School in Mathematics, Physics and Astronomy (BIGS–MPA). I am in a great debt to the school for all the different ways that they helped me to make this work done. I am personally grateful to Prof. Carl-Friedrich Bödighemier, the person who did so much for the graduate school to be such a wonderful experience. This thesis owes the people who read the early versions of this work many a correction made. These people are Dr. Yakov Nekrich, Natalie Palina, Girisan Venugopal, and Claus Viehmann. I thank Christiane Andrade who ensured compliance of this thesis to the formal regulations of the faculty. I am also very grateful to Prof. Sergio Albeverio and Prof. Joachim K. Anlauf, who, despite of their busy schedule, found time to read and grade my thesis.

Thank you, my dear mother, Farida Khasianova, and my lovely grandma, Minnegoel Shafigullina, for your loving support. I owe a great debt to you and your selfless love!

I thank all my friends who distracted me from this work when I badly needed to take a break, and get back to work fresh with a new inspiration.

I want to dedicate this work to my grandfather Shahimardan Shafigullin who, to my great sadness, did not live to rejoice the results of my work.

Contents

Preface iii

Acknowledgement xi

1 Computer Science Essentials 1

1.1 Introduction 1

1.2 Describing a problem to a computer 6

2 Classical Models of Computation 11

2.1 Turing machine 11

2.2 Turing machine with multiple strings 17

2.3 Branching programs 27

2.4 Communication model 33

2.4.1 Complexity measure 37

2.4.2 Properties of communication function 40

2.4.3 One-way communications 42

2.4.4 Lower bounds 44

3 Nondeterministic and Randomized Models 47

3.1 Nondeterministic Turing machines 47

3.2 Randomized Turing machines 50

3.3 Unrealistic probabilistic Turing machines 53

3.4 Realistic yet probabilistic Turing machines 55

3.5 Randomized branching programs 57

4 Quantum Computations **63**
4.1 Invention of quantum mechanics . . . 63
4.2 Introduction to quantum mechanics . . . 68
4.3 Quantum branching programs . . . 77
4.4 Quantum branching programs complexity . . . 84
4.5 Occam's Razor and quantum computers . . . 91

5 Connections to the Hidden Subgroup Problem **93**
5.1 Introductions . . . 93
5.2 Missing an important function . . . 95
5.3 The upper bound for the equality function . . . 97
5.4 The upper bound for the Periodicity function . . . 104
5.5 The upper bound for the Semi-Simon function . . . 108
5.6 The lower bounds . . . 109

6 The Hidden Subgroup Problem **113**
6.1 Introduction . . . 113
6.2 The upper bound for the hidden subgroup test . . . 117
6.3 The lower bound for the hidden subgroup test . . . 122
6.4 Sure states remark . . . 136

7 Reducibility Theory **139**
7.1 Introduction . . . 139
7.2 Rectangular reduction . . . 140
7.3 Polynomial projections . . . 141

8 Conclusion **145**

A The list of notation **149**

B Additional material **153**
B.1 On Chernoff bound . . . 153
B.2 Complexity classes . . . 154

B.3 **NP**-Intermediate problems 154

Bibliography **157**

List of Tables

4.1 Elements of linear algebra in quantum mechanics 69

A.1 List of most frequently used notation 149

List of Figures

1.1 An imaginary device that decides the PROBLEM. 9
1.2 An imaginary device that computes a function $f(x)$. 10

2.1 A Turing machine. 14
2.2 Turing machine and example of computation for O_2 15
2.3 Turing machine and example of computation for a string function . . 16
2.4 Two-string Turing machine that computes bitwise addition modulo two. 20
2.5 Ordered Binary Decision Diagrams of addition modulo two. 28
2.6 A two-party communication protocol. 35
2.7 A one-way two-party communication protocol. 42

3.1 A nondeterministic branching program for $g(x)$. 58

4.1 The state space of the qubit 70
4.2 Communicating two classical bits by sending one qubit 75
4.3 Quantum branching program 79
4.4 Quantum linear program 82
4.5 OBDD complexity classes hierarchy 89

6.1 Hidden subgroup problem 115
6.2 Communication protocol simulating OBDD 126

B.1 Plethora of complexity classes. 155
B.2 Class NP-Intermediate 156

Chapter 1

Computer Science Essentials

> Computer Science is no more about computers than astronomy is about telescopes.
>
> E. W. Dijkstra

1.1 Introduction

A central notion of *computer science* is *Algorithm*. The word is taken from the name of 9th century Arabian mathematician called Muhammad ibn Musa abu Abdallah *al-Khorezmi* al-Madjusi al-Qutrubulli. Medieval Europeans learnt arithmetics from his books. However, the idea of algorithm preceded him at for millennia. The *Euclid's algorithm* for finding *largest common divisor* dates back to 300 BC.

Despite of its long history the notion eluded rigorous mathematical definition until 1936 when Alan Turing introduced a model that later was given his name – the *Turing machine* [Tur36]. Around the same time, another famous mathematician, Alonzo Church proposed the notion of *recursive functions* [Chu36] for capturing the essence of what an algorithm is. A thesis later dubbed as *Church-Turing Principle* was proposed.

> A Turing Machine (recursive function) concept captures what an algorithm performed by any physical device is.

It was this statement that established the foundations of computer science as a rigorous mathematical discipline. Yet it was to be taken as an axiom, although one generally agreed upon. There could be no proof of its validity. A whole new field of *theory of computability* branched out starting off with *Church-Turing Principle*. But something was missing. Computer scientists began to recognize that it was not sufficient for a problem to have a solution, in order to be actually solved. The solution had to be *feasible*. Initially, problems shown to be *intractable* fell short of practical value. In mid 1960's Hartmanis and Stearns [HS65] provided first artificially constructed *decidable* intractible problems. That is problems that being decidable in principle (*Turing machine decidable*) had no algorithm that would decide them in *reasonable time*. The reasonable time was proposed to be a polynomial of the input length. Only in early 1970's Meyer and Stockmeyer [MS72], Fischer and Rabin [FR74], and later others finally provided "natural" examples of intractable problems that were decidable. Namely the "intractability" here essentially meant the problems could not be decided by a *nondeterministic Turing machine* in polynomial time, although they could be decided given more time resources. Those problems came mostly from the automata theory, the theory of formal languages and mathematical logic.
However great that leap from *undecidability* to *intractability* was, the concept failed to catch hardness of numerous practical problems that had polynomial nondeterministic Turing-machine algorithms. But a polynomial time real-world computer solutions had ever eluded researchers. Those problems had to be regarded "intractable" just as well! The time has come to bring up a new kind of thinking. The idea was already floating in the air. In 1971 Stephen Cook [Coo71] introduced the concept of **NP**-*completeness.*
The class **NP** of problems solvable in polynomial time by the nondeterministic Turing machines had been known before. Cook proved that a particular problem in **NP**, the Boolean formula satisfiability, had a property that every problem in **NP** could be *reduced* to it. That is, for every problem in **NP** there is a polynomial algorithm for a deterministic Turing machine that transforms instances of that problem to those of

the *Satisfiability*. Moreover, the *reduction* f_n is such that that X is a solution of the initial problem if and only if, $f_n(X)$ is a solution for the satisfiability.
Later, in 1972 Richard Karp showed that the decision problem versions of many well known combinatorial problems, like the *traveling salesman problem*, are just as hard as *satisfiability* [Kar72]. The set of all these "hardest" problems in **NP** has been given a name: **NPC** - the class of **NP**-complete problems.
Importance of this approach to intractability was later nicely illustrated by Garey and Johnson [GJ79]. A puzzled software engineer that fails to find both an efficient algorithm and a proof that no polynomial-time algorithm is possible for some problem. But he manages to show that a problem from **NPC** is reduced to the problem he's been struggling to find an efficient algorithm for. Given a concept of **NP**-completeness, now the engineer can boldly claim:

> I can't find an efficient algorithm, but neither can all these famous people.

It is still an open question whether the class of problems solvable in polynomial time on a *deterministic* Turing machine called **P** equals **NP**. A million of dollars is promised for the answer on $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ [fM]. Whatsoever, we can tell with certainty that **NPC** problems are the "hardest" in **NP**.
The progress in understanding "intractability" and "efficiency" did not leave unchanged even the corner stone of the computer science. The Church-Turing thesis, that earlier gave raise to the *commputability theory*, failed to provide a rigorous ground to study *complexity*. In the time when **NPC** was introduced, problems that could be solved "efficiently" were also noticed to be solved efficiently by a Turing machine. The observation was fixed in the *strong* Church-Turing thesis.

> A Turing machine can simulate *efficiently* any physical process of computations.

That day of early 1970s the *Complexity Theory* was born. Since then it's been growing tall and thick. Set out with **NP**-completeness, the complexity classes compendium

now accounts over 300 classes [Aar]. Experts still argue about which classes are "natural" and deserve their existence and which are just mathematical "monsters" that must be given to the Occam's razor[1]. There are several well-written, now considered classical, textbooks on the subject [BDG88, BDG90, Pap94, GJ79].
However, even the "strong" Church-Turing thesis did not look strong enough anymore when Robert Solovay and Volker Strassen came up with a polynomial-time *probabilistic* algorithm for primality test that after a few repetitions gave correct answer with nearly certainty. It was only in 2002 that Agrawal, Kayal and Saxena discovered a determinstic polynomial time algorithm for primality testing [AKS02]. In 1976, the Solovay-Strassen algorithm implied that a computer with an access to a random number generator could outperform deterministic Turing machines. That indefiniteness put under controversy the very basis of complexity theory – the strong Church-Turing thesis. It was hastily amended to meet the challenge.

> A *probabilistic* Turing machine can simulate *efficiently* any physical process of computations.

From then on, the class **BPP** – "a Probabilistic Bounded error Polynomial time", the probabilistic analog of P, established itself as the formalization for the class of "efficiently" solvable problems.
After the parade of the Church-Turing principles, followed on by the parade of the complexity classes, as it was rightfully noted in Nielsen and Chuang [NC00],

> This *ad hoc* modification of the strong Church-Turing thesis should leave you feeling rather queasy. Might it not turn out at some later date that yet another model of computation allows one to efficiently solve problems that are not efficiently soluble within Turing model of computation?

Indeed, in 1985 David Deutsch, motivated by the very same question, came up with a natural idea that surprisingly enough had been escaping mathematicians. If we look at the second part of the Church-Turing principle, we can see there: "any physical

[1]William of Ockham (Occam) English scholastic philosopher and assumed author of Occam's Razor (1285-1349)

process of computation". In the same time, physics had been left out of consideration for nearly half a century, since the first version of the principle was proposed! Instead of adopting new intuitive hypothesis, Deutsch attempted to define a *model of computation* that would be capable of simulating efficiently any arbitrary physical theory. Thus, a *quantum mechanical* analogue of the Turing machine was proposed. Deutsch also considered a simple example suggesting that *quantum computers* could have stronger computational power than their classical counterparts. This intuition was further strengthened by later results of Grover [Gro97] and Shor [Sho94, Sho97]. Let's name the following conjecture "the *quantum* Church-Turing thesis".

> A *quantum* Turing machine can simulate *efficiently* any physical process of computations.

We don't know whether quantum Turing machine is a more powerful device than its classical analogue. However, discovery of the quantum computations has far going implications. First of all, the quantum version of the Church-Turing principle is a *provable conjecture* that depends on the validity of the given physical theory, of *quantum mechanics*. This is a better situation than building up a theory based on the pure intuition only very indirectly related to the physical reality!

It is yet an open question if class QBP (Quantum Bounded-error Polynomial) should be taken for the rigorous definition of "efficient". But what is apparent is that quantum computational models, and subsequently *quantum complexity* do not represent revolution in computer science. Instead they turn out to be logical consequences of the long evolutionary development of the understanding what an algorithm is and what makes the algorithm "efficient". In fact, as the open question concerning QBP exhibits it, the evolution is not yet over. It is still under way. I believe, thus, it would be appropriate to begin this work on quantum computers with an introduction to classical complexity.

1.2 Describing a problem to a computer

It is not trivial to describe a problem to a human, neither it is trivial to describe a problem to a machine. There must be some common speech, we formulate the problems in, that the "solver" would understand. What is more, we first should agree what we call a *computational problem*! In this subsection we describe the common speech mentioned above, and define the notion of a computational problem.
We start with the basics of the computer science "vocabulary" listed below.

1. *Alphabets* $\big($e.g. $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{a, b, c, d, e, f, g\}\big)$;

2. *Words* $\big($e.g. $123210, 23145, abba, fade, dgdg\big)$;

3. *Languages* $\big($e.g. $A = \{123, 1234, 12345, 123456, \ldots\}$, $B = \{abba, fade\}$, $C = \{000, 111, 222, 333, 444\}\big)$;

4. *Classes* $\big($e.g. {A, B}, {C}$\big)$;

The definitions we are going to present are mostly standart across *computational complexity*, *mathematical logic* and *formal languages* theory literature. They were adopted by complexity theorists as well as the formulation of a computational problem was borrowed from the *computability theory*.

Definition 1.2.1. An alphabet is any non-empty, finite set. We shall use upper case Greek letters to denote alphabets. The cardinality of alphabet Σ is denoted $|\Sigma|$. The elements of Σ are assumed to be indivisible symbols.

Definition 1.2.2. A word (or a chain, a string) over Σ is a finite sequence of symbols from Σ.

Let Σ^* denote the set of all finite-length strings over Σ, and Σ^n denote the set of all n-length strings over Σ. Let's define the operation of *concatenation*.

Definition 1.2.3. For any $x, y \in \Sigma^*$, result of the binary operation of concatenation is the new word xy.

Example 1. Let $\Sigma = \{0, 1\}$ and $x = 010, y = 1$ then $xy = 0101$ and $yx = 1010$

Definition 1.2.4. We shall need a special symbol, which is denoted λ. It is called the empty symbol.

Next proposition will clarify significance of the *empty symbol*.

Proposition 1.2.1 ([Har78]). *For any given alphabet Σ, the set Σ^* is a monoid under concatenation and λ is the identity element.*

The definition of a language as it is given here probably will be not precise enough for the theory of formal languages. But it is good enough for our purposes.

Definition 1.2.5. Given an alphabet Σ, a language over Σ is a subset of Σ^*.

There is a traditional notation of operations over languages. It is found in all subfields across the scope of the computer science. That's why it is worth to mention it here.

Definition 1.2.6 ([Hro97]). Let A be a language over an alphabet Σ, and B be a language over an alphabet Γ. We define

1. For any homomorphism $h : \Sigma^* \to \Gamma^*$

$$h(A) := \{h(w)|w \in A\}$$

2. We define the *complement of the language* A according to the alphabet Σ as

$$A^{c\Sigma} := \Sigma^* - A.$$

   If the alphabet $\Sigma = \{0, 1\}$, we use the conventional complement notation A^c.

3. We define the *concatenation of the languages* A and B as

$$AB = A \cdot B := \{w|w = xy, x \in A, y \in B\},$$

   where xy means concatenation of words.

4. We further generalize the concept of concatenation of languages.

$$\begin{aligned} B^0 &:= \{\lambda\}, \\ B^{i+1} &:= B \cdot B^i \text{ for any } i \in \mathbb{N}, \\ B^+ &:= \cup_{i=1}^{\infty} B^i, \\ B^* &:= \cup_{i \in \mathbb{N}} B^i = B^+ \cup \{\lambda\}. \end{aligned}$$

5. Finally, we define the *level* n of the language B as

$$B[n] := B \cap \Gamma^n = \{x \in B | \ |x| = n\} \text{ for any } n \in \mathbb{N}.$$

A language is a set. Being so, all set operations, like union, intersection, complementation etc., are well defined for languages. We shall use the term *class* to denote a set whose elements are also sets. Thus, naturally introducing *classes of languages*. We'll use calligraphic capital latin letters do denote classes of languages ($\mathcal{A}, \mathcal{B}, \mathcal{C}$). The notion of *class of complements* is one of the most widely used in structural complexity theory.

Definition 1.2.7. Given a class $\mathcal{C}$, we define its class of complements denoted by $co\mathcal{C}$:

$$co\mathcal{C} = \{L | \bar{L} \in \mathcal{C}\}.$$

Let's proof a very simple "folklore" lemma from computer science. This lemma usually is not even mentioned by structural complexity researchers.

Lemma 1. *Given classes of languages $\mathcal{C}_1$ and $\mathcal{C}_2$ over Σ^*, $\mathcal{C}_1 \subseteq \mathcal{C}_2$ if and only if $co\mathcal{C}_1 \subseteq co\mathcal{C}_2$. In particular, $\mathcal{C}_1 \subseteq co\mathcal{C}_1$ if and only if $\mathcal{C}_1 = co\mathcal{C}_1$.*

Proof. Let $\mathcal{C}_1 \subset \mathcal{C}_2$ and $A \in co\mathcal{C}_1$. This implies that $\bar{A} \in \mathcal{C}_1$. Subsequently, $\bar{A} \in \mathcal{C}_2$. Finally, $A \in co\mathcal{C}_2$. The convers follows from the fact that $co(co\mathcal{C}) = \mathcal{C}$. □

We have expected to understand the way a computational problem is rigorously defined in computer science. Instead we have plunged into the field of abstract languages. Surprisingly, at least at the first glance, a *language* is exactly the notion used

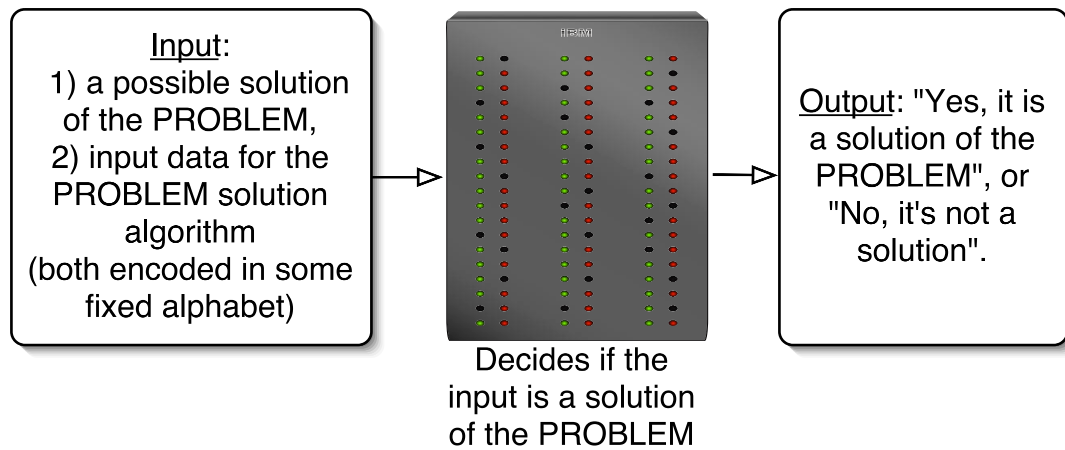

Figure 1.1: An imaginary device that decides the PROBLEM.

to capture what a computational problem is.
Let us imagine a device that knows how to solve a given problem. It would expect a possible solution, an *instance*, of this problem at the input. Received the input, the device would let us know if the solution we gave was correct or wrong (See fig. 1.1). Assume that the problem instances at the input are encoded in some alphabet Σ, thus, the instances are simply words in Σ^*. Let's take all the words that represent correct solutions of the problem, and define a set of words $PROBLEM \subset \Sigma^*$:

$$PROBLEM = \{w | w \in \Sigma, w \text{is a correct solution of the problem}\}. \tag{1.1}$$

The set $PROBLEM$ is a language, a subset of Σ^*. This is exactly the most fundamental way a *computational problem* concept is captured in computer science. This kind of problems is called *decision problems*. Any computational problem can be formulated as a decision problem, there will be a language (over some alphabet) corresponding to it.

Example 2 (PRIMES). Consider the problem of *primality test*. The question of the computational problem is:

For a given number N, decide is it a prime number?

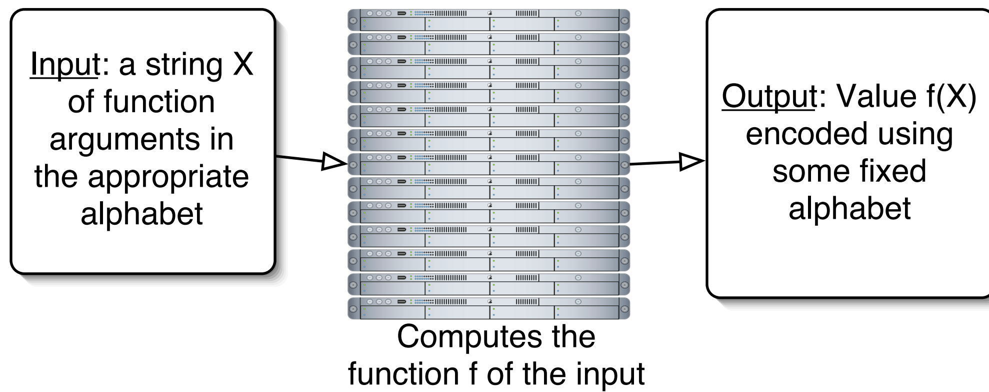

Figure 1.2: An imaginary device that computes a function $f(x)$.

Let's fix $\Sigma = \{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Define language $PRIMES = \{w | w \in \Sigma^*, w \text{ is a prime number}\}$.

Apart form decision problems, we may like to consider *functions* (See fig. 1.2), i.e. problems that require computing a *solution*, rather than just a "YES/NO" sort of answer.
This new implementation of the computational problem concept does not encompass any more computational problems that can be studied using decision problems paradigm. However, it does allow us examine more of different computational models. It is models of computation that we shall be concerned with in the next chapter.

Chapter 2

Classical Models of Computation

> I do not fear computers, I fear the lack of them.
>
> — Isaac Asimov

So far we presented only the mathematical abstraction for the concept of computational problem. Although, we have mentioned mysterious "imaginary devices", we still don't have a mathematical definition for a computer! According to the widely accepted *strengthened Church-Turing thesis*, the mathematical notion of the *Turing machine* captures substantial features of any conceivable real-world computer (See the introduction section, or [Tur36]). Let's define the Turing machine first.

2.1 Turing machine

Definition 2.1.1 ([Pap94]). Formally, a Turing machine is a quadruple $M = (K, \Sigma, \delta, s)$.

1. Here K is a finite set of states, $s \in K$ is the *initial* state;

2. Σ is the alphabet of M. We assume that K and Σ are disjoint sets. Σ always contains the special symbols $\sqcup$ and $\triangleright$: the *blanc* and the *first* symbol.

3. Finally, δ is a *program*, or the *transition function*, which maps $K \times \Sigma$ to $(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that h (the *halting state*), "yes" (the *accepting state*), "no" (the *rejecting state*), and the *cursor directions* $\leftarrow$ for "left", $\rightarrow$ for "right", and $-$ for "stay", are not in $K \cup \Sigma$.

A Turing machine works in *discrete time*. On each step we can consider a complete description of the current state of the machine, including all information contained in the string. The string, naturally, will be finite in any given finite moment of time. We intuitively described a *configuration* of a Turing machine. Let's give its formal definition.

Definition 2.1.2. A configuration of a Turing machine M is a triple (q, w, u), where $q \in K$ is a state, and w, u – strings in Σ^*, w is a string to the left of the cursor, including the symbol scanned by the cursor, and u is the string to the right of the cursor, possibly empty, q is the current state. We shall use ϵ to denote the empty string. We shall also use the notation $(q, w, u) \overset{M^t}{\rightarrow} (q', w', u')$, if there are configurations $(q_1, w_1, u_1), \ldots, (q_{t-1}, w_{t-1}, u_{t-1})$, such that the transitions $(q, w, u) \rightarrow (q_1, w_1, u_1) \rightarrow \ldots \rightarrow (q_{t-1}, w_{t-1}, u_{t-1})$ are legitimate according to the program δ. We say that the configuration (q, w, u) *yields* the configuration (q', w', u') in t steps. If $(q, w, u) \overset{M^t}{\rightarrow} (q', w', u')$ for some t, we simply say (q, w, u) *yields* (q', w', u').

Working time of every Turing machine always has a beginning. It is a natural assumption, since we don't have perpetual computers, that could had been working infinitely long prior to a fixed moment of time. We already defined a special state $s \in K$ that is called a *starting state* (See **Definition 2.1.1**). But only fixing a state is not enough to describe a Turing machine completely in the beginning of the computation.

Definition 2.1.3. *Initial configuration* of a Turing machine is always $(s, \triangleright, x)$, where x is the input, possibly empty.

As we mentioned earlier, a Turing machine is an abstraction of a computational device. So far we have had only the device, but not the rigorous notion of a computation it performs. Now, that we have notions of *configuration* and *initial configuration*, we can define the notion of *computation* of a Turing machine.

Definition 2.1.4 ([BDG88]). Given a machine $M = (K, \Sigma, \delta, s)$ and an input string x, a *partial computation* of M on x is a (finite or infinite) sequence of configurations of M, in which each step from a configuration to the next obeys the transition function δ. A *computation* is a partial computation which starts with the initial configuration of M on x, and ends in a configuration in which no more steps can be performed.

Remark 1. Sometimes a transition function δ is allowed to be a partial function. Then any configuration for which δ is not defined is assumed equivalent to a configuration with "no" (or h if a function is computed) state of the final state control.

A Turing machine transits from one configuration to another, until it reaches one of the halting states from the set $\{h, \text{"yes"}, \text{"no"}\}$. If it halts in the state h, then we write $M(x) = y$, where y is the result of the computation contained on the string. More precisely, before it halted M worked finite amount of time, thus, it managed to fill the string with a finite number of symbols. We take y to be the sequence right to $\triangleright$ whose last symbol is not a $\sqcup$, possibly followed by a string of $\sqcup$. If M halts in the "yes" or the "no" state we write $M(x) = \text{"yes"}$ or $M(x) = \text{"no"}$ respectively. However, for some input x a machine M may never halt, we denote this case by writing $M(x) = \nearrow$.

Remark 2. We require that if for some states p and q $\delta(q, \triangleright) = (p, \rho, D)$, then $\rho = \triangleright$ and $D = \rightarrow$. In other words we don't allow the cursor to fall off the left end of the string (the symbol $\triangleright$). However, the cursor may move right without restriction. We agree that right of the input the string contains empty symbols $\sqcup$, that can be, of course, overwritten.

Allowing the head move left we don't add any computational power, rather we introduce some unnecessary complexity to the algorithm. For example, having deleted the left endmark, we have to remember where was the input on the string: to the left from the cursor, or to right. Otherwise, we'll have to traverse the string in both directions!

Figure 2.1 demonstrates a graphical representation of a Turing machine. The picture contains three structural elements:

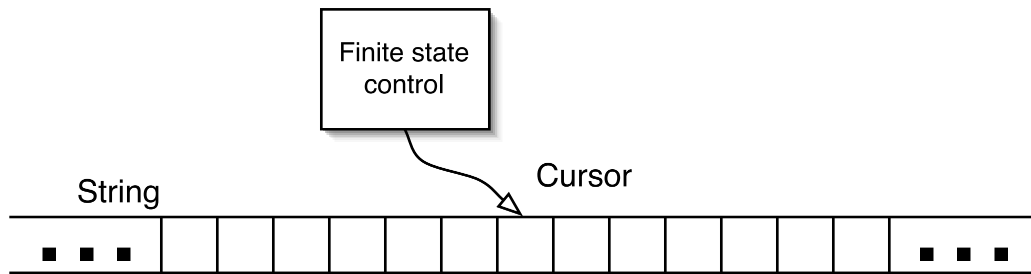

Figure 2.1: A Turing machine.

- A *finite state control* unit, that resides in any of the valid states from the set K, the states are changed according to the transition function δ;
- A *string* of symbols from the alphabet Σ, the machine can read and write symbols of that string;
- A *cursor* that points to a symbol of the string the machine currently observes, at any time it can observe only one symbol, initial conditions (see the Definition 2.1.1) and the transition function δ define which symbol is currently scanned and where the cursor moves next.

The Turing machine is our first mathematical abstraction of a *computational device*. Computers solve problems, and in our world problems are represented by *languages*. What does it mean for a Turing machine to *decide a language*?

Definition 2.1.5 ([Pap94]). Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language. Let M be a Turing machine such that , for any string $x \in \{\Sigma - \{\sqcup\}\}^*$ if $x \in L$ then $M(x) =$ "yes" and, if $x \notin L$ then $M(x) =$ "no". Then we say that M *decides* L.
If L is *decided* by some Turing machine M, then L is called *recursive language*.
We say that M simply *accepts* L whenever, for any string $x \in (\Sigma - \{\sqcup\})^*$, if $x \in L$ then $M(x) =$ "yes"; however, if $x \notin L$ then $M(x) = \nearrow$.
If L is accepted by some Turing machine M, L is called *recursively enumerable*.

Let's define a language, and try to see if it is recursive, or at least recursively enumerable. In other words, we shall try to construct a Turing machine, that decides, or

at least accepts the language.

Definition 2.1.6. We define a language O_2 over alphabet $\{1, 0\}^*$ as a set of words that contain equal number of zeros and ones.

It turns out that O_2, also known as the *language of balanced words*, is recursive!

Example 3 (A Turing machine that decides O_2). We define the Turing machine $M = (K, \Sigma', \delta, s)$

1. The finite control states: $K = \{s, q_0, q_1, q\}$
2. The alphabet: $\Sigma = \{\triangleright, \sqcup, 0, 1, \diamond\}$
3. The transition function δ is defined in the Figure 2.2.

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
$s,$	$\triangleright,$	$(s, \triangleright, \rightarrow)$
$s,$	$0,$	$(q_0, \diamond, \rightarrow)$
$s,$	$1,$	$(q_1, \diamond, \rightarrow)$
$s,$	$\diamond,$	$(s, \diamond, \rightarrow)$
$q_0,$	$0,$	$(q_0, 0, \rightarrow)$
$q_0,$	$1,$	$(q, \diamond, \leftarrow)$
$q_0,$	$\diamond,$	$(q_0, \diamond, \rightarrow)$
$q_1,$	$1,$	$(q_1, 1, \rightarrow)$
$q_1,$	$0,$	$(q_1, \diamond, \leftarrow)$
$q_1,$	$\diamond,$	$(q, \diamond, \rightarrow)$
$q,$	$0,$	$(q, 0, \leftarrow)$
$q,$	$1,$	$(q, 1, \leftarrow)$
$q,$	$\diamond,$	$(q, \diamond, \leftarrow)$
$q,$	$\triangleright,$	$(s, \triangleright, \rightarrow)$
$s,$	$\sqcup,$	$(\text{"yes"}, \sqcup, \rightarrow)$
$q_0,$	$\sqcup,$	$(\text{"no"}, \sqcup, \leftarrow)$
$q_1,$	$\sqcup,$	$(\text{"no"}, \sqcup, \leftarrow)$

1.	$s,$	$\underline{\triangleright}011\sqcup$
2.	$s,$	$\triangleright\underline{0}11\sqcup$
3.	$q_0,$	$\triangleright \diamond \underline{1}1\sqcup$
4.	$q,$	$\triangleright\underline{\diamond} \diamond 1\sqcup$
5.	$q,$	$\underline{\triangleright} \diamond \diamond 1\sqcup$
6.	$s,$	$\triangleright\underline{\diamond} \diamond 1\sqcup$
7.	$s,$	$\triangleright \diamond \underline{\diamond}1\sqcup$
8.	$s,$	$\triangleright \diamond \diamond\underline{1}\sqcup$
9.	$q_1,$	$\triangleright \diamond \diamond \diamond \underline{\sqcup}$
10.	"no",	$\triangleright \diamond \diamond\underline{\diamond}\sqcup$

Figure 2.2: Turing machine and example of computation for O_2

Example 3 of the Turing machine deciding O_2 demonstrates the use of all concepts we have introduced so far. It also exhibits importance of the Remark 2, that demanded

a Turing machine head to never cross the left margine mark $\triangleright$ of the string. Indeed, the algorithm could have been less elegant, had we ignored that "rule of thumb"! Apart from deciding languages, Turing machines can compute *string functions*.

Definition 2.1.7 ([Pap94]). Suppose that f is a function from $(\Sigma - \{\sqcup\})^*$ to Σ^*, and let M be a Turing machine with alphabet Σ. We say that M *computes* f if, for any string $x \in (\Sigma - \{\sqcup\})^*, M(x) = f(x)$. If such an M exists, f is called a *recursive function*.

Note that for a Turing machines computing a string function, its output defined by the string content rather than by the state it halted with (see page 13). Next example demonstrates how Turing machine can compute a string function.

Example 4 (A Turing machine that computes a string function). For a binary string $x \in \{0,1\}^*$, let $n = |x|$ be the length of the string. Define a function

$$f(x) = 2^n - 1 - x,$$

where x is interpreted as a number in binary. A short thought shows that this function is simply a *bitwise negation* of x. We define the Turing machine $M = (K, \Sigma, \delta, s)$

1. The finite control states: $K = \{s\}$
2. The alphabet: $\Sigma = \{\triangleright, \sqcup, 0, 1, \}$
3. The transition function δ is defined in the Figure 2.3.

$p \in K$	$\sigma \in \Sigma$	$\delta(p,\sigma)$
$s,$	$\triangleright,$	$(s, \triangleright, \rightarrow)$
$s,$	$0,$	$(s, 1, \rightarrow)$
$s,$	$1,$	$(s, 0, \rightarrow)$
$s,$	$\sqcup,$	$(h, \sqcup, \leftarrow)$

1. $s,\ \underline{\triangleright}011\sqcup$
2. $s,\ \triangleright\underline{0}11\sqcup$
3. $s,\ \triangleright1\underline{1}1\sqcup$
4. $s,\ \triangleright10\underline{1}\sqcup$
5. $s,\ \triangleright100\underline{\sqcup}$
6. $s,\ \triangleright10\underline{0}\sqcup$

Figure 2.3: Turing machine and example of computation for a string function

We have introduced mathematical abstractions for both *decider* and *transducer* types of computers. That means we have now got everything we need to study computational complexity of different problems. But why should we take a Turing machine as the ultimate mathematical tool to represent a computational device? We know that it was not the only model proposed along with a plenty of others. Though, they were all equivalent in respect of the *computability* of problems. In the following sections, we shall introduce some different computational paradigms that will help us to see more complexity "faces" of computational problems.

2.2 Turing machine with multiple strings

First natural improvement could be adding more tapes to a Turing machine. What advantage would we then gain? Let's address this question.

Definition 2.2.1 ([Pap94]). A k-string Turing Machine, where $k \geq 1$ is an integer, is a quadruple $M = (K, \Sigma, \delta, s)$, where K, Σ, and s are exactly as in ordinary Turing machines. Similarly, δ defines program of the machine. Formally, δ is a function from $K \times \Sigma^k$ to $(K \cup \{h, ''yes'', ''no''\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$. Initially all strings start with a $\triangleright$ symbol read. The input is contained on the first string left to the symbol $\triangleright$. If the k-string Turing machine computes a string function the output is read from the last kth string when the machine halts.

Semantically, $\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$ means that:

- From the state q with k cursors (of the respective strings) scanning symbols $\sigma_1, \sigma_2, \ldots, \sigma_k$ respectively, the machine transites into state p;
- Then all symbols $\sigma_1, \sigma_2, \ldots, \sigma_k$ are overwritten with $\rho_1, \rho_2, \ldots, \rho_k$ respectively;
- Finally, all k cursors are moved according to the respective directions $D_1, D_2, \ldots, D_k$.

Remark 3. Similarly to the ordinary Turing machines we require the $\triangleright$ symbol still can not be overwritten or passed on the left! If $\sigma_i = \triangleright$, then $\rho_i = \triangleright$, and $D_i = \rightarrow$.

Notions of *configuration, initial configuration* are simply natural extensions of those defined for conventional Turing machines.

Definition 2.2.2 ([Pap94]). A configuration of a k-string Turing machine M is a $(2k+1)$-tuple $(q, w_1, u_1, \ldots, w_k, u_k,)$, where $q \in K$ is a state, the ith string reads $w_i u_i$, and the last symbol of w_i is holding the ith cursor.

Definition 2.2.3. *Initial configuration* of a k-string Turing machine is always $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \epsilon)$, where x is the input, possibly empty, and ϵ denotes the *empty string.*

The notion of computation of a k-string Turing machine is defined exactly as it was done for the one-string Turing machines.
In the next example a two-string Turing machine elegantly computes *bitwise addition modulo two.*

Example 5 (Addition modulo two). Let input be defined as $x \oplus y$, where $x, y \in \{0,1\}^n$, n is an arbitrary natural number. We want to compute a string function $f(x \oplus y) = x \oplus y$, where x and y are interpreted as numbers represented in binary encoding.
Let us define the two-string Turing machine $M = (K, \Sigma, \delta, s)$ that computes f.

1. The finite control states: $K = \{s, l\}$
2. The alphabet: $\Sigma = \{\triangleright, \oplus, \sqcup, 0, 1, \}$
3. The partial transition function δ is defined in the **Figure 2.4.**

Note, that the additional string allowed the Turing machine save some computational steps in the example above. Perhaps, it is a good time to introduce our first complexity measure. It will be the number of steps a Turing machine expends on the computation. In other words, our first complexity measure corresponds to the most natural resource utilized by computers (and wasted by their users), to the *time.*

Definition 2.2.4 ([Pap94]). If for a k-string Turing machine M and input x we have $(s, \rhd, x\rhd, \epsilon, \ldots, \rhd, \epsilon) \xrightarrow{M^t} H, w_1, u_1, \ldots, w_k, u_k$ for some $H \in \{h, \text{"yes"}, \text{"no"}\}$, then the *time* required by M on input x is t. That is, the *time* is simply the number of steps to halting. If $M(x) = \nearrow$, then the time required by M is thought to be ∞.

$p \in K$	$\sigma_1 \in \Sigma$	$\sigma_2 \in \Sigma$	$\delta(p, \sigma_1, \sigma_2)$
$s,$	$\triangleright,$	$\triangleright,$	$(s, \triangleright, \rightarrow, \triangleright, \rightarrow)$
$s,$	$0,$	$\sqcup,$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s,$	$1,$	$\sqcup,$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s,$	$\oplus,$	$\sqcup,$	$(l, \oplus, -, \sqcup, \leftarrow)$
$l,$	$\oplus,$	$0,$	$(l, \oplus, -, 0, \leftarrow)$
$l,$	$\oplus,$	$1,$	$(l, \oplus, -, 1, \leftarrow)$
$l,$	$\oplus,$	$\triangleright,$	$(s, \oplus, \rightarrow, \triangleright, \rightarrow)$
$s,$	$0,$	$0,$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s,$	$0,$	$1,$	$(s, 0, \rightarrow, 1, \rightarrow)$
$s,$	$1,$	$0,$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s,$	$1,$	$1,$	$(s, 1, \rightarrow, 0, \rightarrow)$
$s,$	$\sqcup,$	$\sqcup,$	$(h, \sqcup, -, \sqcup, -)$

1.	$s,$	$\underline{\triangleright}011 \oplus 100\sqcup$ $\underline{\triangleright}\sqcup$
2.	$s,$	$\triangleright\underline{0}11 \oplus 100\sqcup$ $\triangleright\underline{\sqcup}$
3.	$s,$	$\triangleright0\underline{1}1 \oplus 100\sqcup$ $\triangleright0\underline{\sqcup}$
4.	$s,$	$\triangleright01\underline{1} \oplus 100\sqcup$ $\triangleright01\underline{\sqcup}$
5.	$s,$	$\triangleright011\underline{\oplus}100\sqcup$ $\triangleright011\underline{\sqcup}$
6.	$l,$	$\triangleright011\underline{\oplus}100\sqcup$ $\triangleright01\underline{1}\sqcup$
7.	$l,$	$\triangleright011\underline{\oplus}100\sqcup$ $\triangleright0\underline{1}1\sqcup$
8.	$l,$	$\triangleright011\underline{\oplus}100\sqcup$ $\triangleright0\underline{1}1\sqcup$
8.	$l,$	$\triangleright011\underline{\oplus}100\sqcup$ $\triangleright\underline{0}11\sqcup$
9.	$l,$	$\triangleright011\underline{\oplus}100\sqcup$ $\underline{\triangleright}011\sqcup$
10.	$s,$	$\triangleright011 \oplus \underline{1}00\sqcup$ $\triangleright\underline{0}11\sqcup$
11.	$s,$	$\triangleright011 \oplus 1\underline{0}0\sqcup$ $\triangleright1\underline{1}1\sqcup$
12.	$s,$	$\triangleright011 \oplus 10\underline{0}\sqcup$ $\triangleright11\underline{1}\sqcup$
13.	$s,$	$\triangleright011 \oplus 100\underline{\sqcup}$ $\triangleright111\underline{\sqcup}$
13.	$h,$	$\triangleright011 \oplus 100\underline{\sqcup}$ $\triangleright111\underline{\sqcup}$

Figure 2.4: Two-string Turing machine that computes bitwise addition modulo two.

However, in reality we need a measure defined on the problems, in other words, on *languages*. The definition above provides a complexity measure defined only for specific instances of problems, that is, on the pairs consisting of a *word* and a *Turing machine*. In order to abstract from particular instances, next we define the time measure on Turing machines alone.

Definition 2.2.5 ([Pap94]). Let f be a function from the nonnegative integers to the nonnegative integers. We say that a Turing machine M *operates within time* $f(n)$, if for any input string x, the time required by M on x is at most $f(|x|)$, where $|x|$ is the length of the string x. We call $f(n)$ *the time bound* for M.

Finally, we abstract even from a specific Turing machine. What left is the language, characterized according the complexity of the problem it corresponds to.

Definition 2.2.6 ([Pap94]). Suppose that a language $L \subset (\Sigma \setminus \{\sqcup\})^*$ is decided by a multistring Turing machine operating in time $f(n)$. We say that $L \in \mathbf{TIME}(f(n))$. That is, $\mathbf{TIME}(f(n))$ is a set of languages. It contains exactly those languages that can be decided by Turing machines with multiple strings operating within the time bound $f(n)$.

It is easy to see, that the Turing machine in **Figure 2.4** works in *linear time*. This implies, that the string function defined in **Example 5** can be computed in time $O(n)$, where n is the input length. Thus, a *decision problem* defined as the *middle bit* of the addition modulo two would belong to $\mathbf{TIME}(O(n))$. We have just defined our first *complexity class* $\mathbf{TIME}(O(n)) = \cup_{k\in\mathbb{R}}\mathbf{TIME}(kn)$, for the input length n. Another interesting complexity class is $\mathbf{P} := \cup_{k\in\mathbb{N}}\mathbf{TIME}\left(n^k\right)$. We say that an algorithm is *time efficient* if it operates within polynomial time. Thus, the class $\mathbf{P}$ contains problems that can be efficiently solved by Turing machines. This is why $\mathbf{P}$ stands out of the plethora of the complexity classes. Complexity classes and their relations are studied in *Structural complexity theory.*

Now it may be a good time to ask a justified question. Why do we choose the multiple string Turing machines as our standard for the time consumption? It well may be unjustified by the possibility of a huge performance gap between the multiple and

single string machines. It also may be incorrect due to the possibility of significant performance gaps between multiple string machines with different number of strings. We are now to resolve these issues, and justify our choice.

Theorem 1 ([Pap94]). *Given any k-string Turing machine M operating within time $f(n)$, we can construct a single string Turing machine M' operating within time $O(f(n)^2)$ and such that, for any input x, $M(x) = M'(x)$.*

Sketch of the proof. For a machine $M = (K, \Sigma, \delta, s)$ we describe a machine $M' = (K', \Sigma', \delta', s)$. The machine M' will contain all k strings of M on its single string, and it will simulate the action of M.
We accomplish this by choosing an apropriate alphabet $\Sigma' := \Sigma \cup \underline{\Sigma} \cup \{\rhd', \lhd, \lhd'\}$, where $\underline{\Sigma} := \{\underline{\sigma} | \sigma \in \Sigma\}$ will be used to store the k heads positions of M on its k strings. Any configuration $(q, w_1, u_1, \ldots, w_k, u_k)$ of M can be simulated by the configuration of M' $(q, \rhd, w'_1 u_1 \lhd w'_2 u_2 \lhd \ldots w'_k u'_k \lhd \lhd)$. Here w'_i is w_i with the leading $\rhd$ replaced by $\rhd'$, and the last symbol σ_i by $\underline{\sigma_i}$. The last pair $\lhd\lhd$ signals the end of the string of M'.
The simulation starts shifting the head to the right, writing $\rhd'$, the input string of M, and the string $\lhd(\rhd'\lhd)^{k-1}$. The latter for the $k-1$ initially empty strings.
The simulation goes in two phases. First, the whole string is scanned for needed changes, according to the program δ of M. Then, the string of M' is traversed second time in order to apply the changes. It takes only linear overhead to simulate every step, including the initial preparation, unless a symbol must be added to one of the strings of M. In this case, the whole content of the string possibly has to be shifted in order to free the space. Since Turing machines *can not expend more space than time*, the latter procedure would not take more than $O(kf(|x|))$ steps. The one position shift right of a string x can be done in $4|x|$ steps. There are exactly $f(|x|)$ steps of M to simulate. Since k is a constant independent on the input length, M' does not waste more than $O(f(|x|)^2)$ steps.

□

We have extensively used asymptotic notation so far. However, we have not yet justified our neglecting of the constant factors! The reason is that an elegant complexity

theory emerges only if we consider *rates of growth,* rather than particular functions, in order to define complexity measures. It is also consistent with the real world, where the "Moore's Law" governs the technology advance rates. Originally, the statement was made by Gordon E. Moore, one of the co-founders of Intel® corporation, in 1965. [Moo65]:

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

The law ever held since then, doubling the number of transistors per chip every 18 months. However, by the year 2020 the silicon wafer technology would approach its limits. The transistor size will have to be a couple of atoms in size, in order to keep in pace with the "Moore's Law". That is why we are off for search of alternative technologies. Their mathematical abstractions are known as *non-classical computational models.*

But before we look on for new models, there's a question left unresolved. Is the multiple strings Turing machine, being our standard classical model, consistent with our aspirations for an elegant theory, and the practical "Moore's Law"? The linear speedup theorem addresses the question.

Theorem 2 (Linear Speedup Theorem [Pap94]). *Let* $L \in \mathbf{TIME}\left(f(n)\right)$. *Then for any* $\epsilon > 0, L \in \mathbf{TIME}\left(f'(n)\right)$, *where* $f'(n) = \epsilon f(n) + n + 2$

Sketch of the proof. Let $M = (K, \Sigma, \delta, s)$ be a k-string Turing machine that decides L and operates in time $f(n)$. We construct a k'-string Turing machine $M' = (K', \Sigma', \delta', s')$ operating within time bound $f'(n)$, and which simulates M. The number $k' = k + 1$.

A simple idea is used to compress the computation. The alphabet $\Sigma' = \Sigma \cup \Sigma^m$. We use m-tuples to encode m symbols of Σ. The value of m will be specified later.
The simulation starts with encoding the input string content of M on the second string of M'. Then, the machine M' simulates m steps of M by at most 6 steps. It moves all cursors to the left, then twice to the right, then back to the left. After that, M' has enough information to predict behavior of further m steps of M. Two more steps of M' may be needed to implement the actions of M. The output of M' will be identical to that of M, but it will use $m/6$ times less of steps for every simulated computation. Finally, setting $m := 6/\epsilon$ we prove the desired speed-up. □

We have made it fair and square with the time measure. However, time is not the only resource used by a Turing machine. Should we not also count the the strings length of Turing machines?
In order to define the *space complexity* of a Turing machine for a given input x, we could simply count all non-empty symbols written on the strings. However, this measure would not be relevant to very economical machines that don't write more than $|x|$ symbols. In the latter case all their complexities would be asymptotically identified due to the predominant additive factor of $|x|$. Consider the following example.

Example 6 (Logarithmic space Turing machine). Let M_1 and M_2 be two Turing machines that recognize the language EQUALITY $= \{x = y | x, y \in \{0,1\}^* \text{ and } x = y\}$.

1. The machine M_1 simply copies the x part to its second string and compares it with the y part. All done in one scan of the input string.

2. The machine M_2 compares the symbols of x and y one-by-one, storing in the second string only the symbol position and its value. The same part of the second string will be used each time, rewriting the previous content. This machine would utilize only $\log_2 |x = y| + 1$ positions of the second string. Although, this machine is less efficient in the time utilization, it saves space significantly, compared to M_1.

Notably, none of the two machines of the **Example 6** changed the input string.

Definition 2.2.7 ([Pap94]). Let $k > 2$ be an integer. A k-string Turing machine *with input and output* is an ordinary k-string Turing machine, with a restriction on the program δ:
Whenever $\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$, then

1. $\rho_1 = \sigma_1$;
2. $D_k \neq \leftarrow$;
3. if $\sigma_1 = \sqcup$ then $D_1 = \leftarrow$.

In other words, M has a read-only input string, a write-only output string, and the head of the read-only string is not allowed to wander off the input.

Observe that we don't weaken our standard computation model by switching to the Turing machines *with input and output.*

Proposition 2.2.1. *For any k-string Turing machine M operating within time bound $f(n)$, there exits a $k+2$-string Turing machine with input and output, recognizing the same language, and operating within a time bound $O(f(n))$.*

The proof of the proposition is trivial. We can finally define the space complexity measure.

Definition 2.2.8 ([Pap94]). Suppose that for a k-string Turing machine M and an input x it holds that $(s, \triangleright, x, \ldots, \triangleright, \epsilon) \rightarrow (H, w_1, u_1, \ldots, w_k, u_k)$, where $H \in \{h, \text{"yes"}, \text{"no"}\}$ is a halting state. Then the *space* required by M on input x is $\Sigma_{i=1}^{k} |w_i u_i|$. If, however, M is a *machine with input and output,* then the *space* required by M on input x is $\Sigma_{i=2}^{k-1} |w_i u_i|$. Suppose now that f is a function from $\mathbb{N}$ to $\mathbb{N}$. We say that a Turing machine operates *within space bound* $f(n)$ if, for any input x, M requires space at most $f(|x|)$.

Analogously to the time classes we can define space complexity classes of languages.

Definition 2.2.9 ([Pap94]). Let L be a language. We say that L is in the *space complexity class* $\mathbf{SPACE}(f(n))$ if there is a Turing machine *with input and output* that decides L and operates within space bound $f(n)$.

The Turing machine M_2 from the **Example 6** shows that

$$\text{EQUALITY} \in \mathbf{SPACE}\left(\log_2 n + 1\right)$$

As a matter of fact, a space analogue of *Linear Speedup Theorem* allows us discard constants for space complexity too. Thus, implying

$$\text{EQUALITY} \in \mathbf{SPACE}\left(\log_2 n\right) =: \mathbf{L}.$$

As another example of space complexity class we define **PSPACE**.

Example 7 (PSPACE). A class of all problems that can be solved by a Turing machine within polynomial space is called **PSPACE**.

$$\mathbf{PSPACE} := \cup_{k \in \mathbb{N}} \mathbf{SPACE}\left(n^k\right)$$

Theorem 3 ([Pap94]). *Let L be a language in* $\mathbf{SPACE}\left(f(n)\right)$*. Then for any* $\epsilon > 0$, $L \in \mathbf{SPACE}\left(2 + \epsilon f n\right)$.

Proof. The proof is an easy modification of the proof of **Theorem 2**. □

We have established the fundamentals – the standard model of the complexity theory. However, computational problem arise in many different settings. Sometimes it is easier to consider a specific computational model than stick to the Turing machines. Those new models give rise to new complexity classes. The latter, in turn, help us better understand complexity of the studied problems. The multiplicity of computational abstractions is also the source of diversity and beauty of the *structural complexity theory*. Far more complexity classes, than we present as examples, are found in Scot Aaronson's *Complexity Zoo* [Aar] that deserves a separate tribute. We present some of the alternative models of computation in coming subsections.

2.3 Branching programs

We have seen that a Turing machine can be modified in order to better fit our research purposes. However, for some complexity research areas Turing machines have to be modified so far that they can be considered a different computational model. The area of our research lays in the complexity of Boolean functions.

Definition 2.3.1. A class of functions consists of *finite functions* if the common domain and the common image of the functions are finite sets. The elements of the domain and the image can be encoded by binary strings of fixed length, the resulting functions are called *Boolean functions*.The class of Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$ is denoted $\mathbb{B}_{n,m}$. The class $\mathbb{B}_{n,1}$ is denoted simply as $\mathbb{B}_n$.

As we mentioned earlier, a Turing machine is a universal abstraction of the algorithm. Thus, it is not impossible to study families of Boolean functions using Turing machines. But recall how far Turing machine programs are from the comfort of the higher level computer languages! One can hardly advance any far trying to develop a complex database using the Turing machine model. It's just not a right abstraction for the purpose! Similarly, *branching programs* offer us better tools for studying Boolean functions complexity than Turing machines. Furthermore, branching programs allow us proving non-trivial lower bounds, that has always been difficult in theoretical computer science. Even furthermore, a branching program also can be considered as a representation of a Boolean function. There is even a way to provide *canonical representations* for Boolean functions using branching programs. Thus, the model offers quite a bit of tools to study even more than only complexity of Boolean functions!
The model of *branching programs* was first systematically studied by W. Masek in his Master's thesis [Mas76].

Definition 2.3.2 ([MT98]). A *branching program* or a *binary decision diagram* is a directed acyclic graph with exactly one root, whose

- sinks are labeled by the Boolean constants 0, 1, and whose

- internal nodes are labeled by a variable x_i and have exactly two outgoing edges, a 0-edge and a 1-edge.

A branching program *represents* a Boolean function $f \in \mathbb{B}_n$ in the following way. Each assignment to the input variables x_i defines a uniquely determined path from the root to one of the sinks of the graph. The label of the reached sink is set to be the function's value on the input.

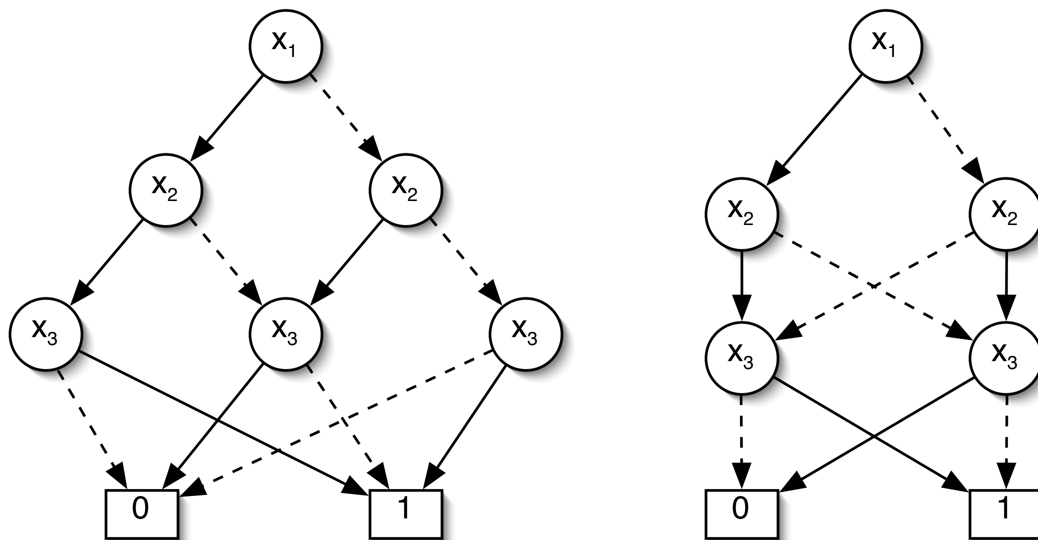

Figure 2.5: Ordered Binary Decision Diagrams of addition modulo two.

There is a certain agreement on the graphical representation of the branching programs. The nodes are usually shown as circles with the corresponding variables within. The 0-edges are drawn as dashed lines, while the 1-edges are represented by solid lines. The sinks are usually shown as squares with corresponding values in them. The **Figure 2.5** demonstrates branching programs that represent the function $g(x_1, x_2, x_3) := x_1 \oplus x_2 \oplus x_3$.

Definition 2.3.3. We define BP to be the class of all branching programs.

The size of a given branching program is a relevant complexity measure on the class of the branching programs.

Definition 2.3.4. The *size* of a branching program is number of its internal nodes.

Our first example of the branching program complexity class is given next.

Example 8. The class of functions represented by *polynomial size* branching programs is called **P-BP**.

However, the size can be decomposed into more elementary complexity measures. We achieve that by introducing the notions of the *level* and the *width* of a branching program. Imposing restrictions on the width or the length of a branching program, we may arrive at interesting restricted classes of branching programs. In fact, nontrivial *restricted computational models* are one of the most attractive features that branching programs offer.

Definition 2.3.5 ([MT98]). Let P be a branching program.

1. The kth *level* of P denotes the set of all nodes which can be reached from the root by a path of length $k-1$.

2. The *width* of P is the maximal cardinality $width\,(P)$ over all levels of P.

The most general kind of branching programs are hard to prove nontrivial statements about. It was imposing restrictions that made the interesting proofs possible.

Definition 2.3.6 ([MT98]). Let P be a branching program.

1. P is called *bounded-width* k if each level of P is of cardinality at most k: $w(P) \leq k$.

2. P is called *synchronous* if for each node v of P, all paths from the root to v are of the same length.

3. P is called *oblivious* if it is synchronous and all non-sink nodes within a level are labeled with the same variable.

4. P is *read-k-times-only* branching program if each variable occurs on every path at most k times.

The definition above can be used to introduce several subclasses of the class of all branching programs BP.

Definition 2.3.7.

1. $BP_{width-k} = \{P|P \in BP, w(P) \leq k\}$;
2. $sBP = \{P|P \in BP, P$ is synchronous$\}$;
3. $oBP = \{P|P \in BP, P$ is oblivious$\}$;
4. $BPk = \{P|P \in BP, P$ is read-k-times-only$\}$.

Let us also introduce several well-known types of branching programs, that have their own names.

Definition 2.3.8.

1. A Free BDD (FBDD) is a read-once branching program.
2. An ordered BDD (OBDD) is a FBDD, where on all paths variables are read according to the same ordering.
3. A k-OBDD consists of k layers of OBDDs respecting the same ordering of variables.
4. A k-IBDD (Indexed BDD) consists of k layers of OBDDs respecting perhaps different orderings.

Consider an OBDD P. Each internal node of the graph can be treated as an OBDD of certain subfunction of the function g represented by P.

Definition 2.3.9. For an OBDD P, we say that a function f is *represented by the node* labeled x_i if the subgraph rooted in this node is an OBDD that represents f.

In fact, for a given OBDD P, for any internal node x_i that represents a Boolean function $f \in \mathbb{B}_n$ the *Shannon's decomposition* rule holds:

$$f = x_i f_1 + \overline{x_i} f_0$$

with the cofactors

$$f_0 = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots x_n) \text{and} f_1 = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots x_n).$$

Consequently, it is clear that the OBDD model is based on the Shannon's decomposition of Boolean functions. Is it not possible to define a branching program based on a different decomposition type? The answer is "Possible!". Below we define *Reed-Muller* and *Davio* decompositions.

Theorem 4 ([MT98]). *Let $f \in \mathbb{B}_n$ be a switching function in n variables. For the functions f_0, f_1, f_2 defined by*

$$\begin{aligned} f_0(x_1, \ldots, x_n) &= f(x_1, \ldots, x_{n-1}, 0), \\ f_1(x_1, \ldots, x_n) &= f(x_1, \ldots, x_{n-1}, 1), \\ f_2(x_1, \ldots, x_n) &= f_0(x_1, \ldots, x_n) \oplus f_1(x_1, \ldots, x_n) \end{aligned}$$

the following holds true:
Reed-Muller *decomposition or* positive Davio *decomposition:*

$$f = f_0 \oplus x_n f_2;$$

Negative Davio *decomposition:*

$$f = f_1 \oplus \overline{x_n} f_2.$$

This decomposition holds, in analogous manner, for each variable $x_i, 1 \leq i \leq n$.

Proof. Due to Shannon's decomposition, with respect to the addition modulo two operation $\oplus$, we have for $x \in \mathbb{B}^n$

$$\begin{aligned} f(x) &= \overline{x_n} f_0(x) \oplus x_n f_1(x) \qquad (\overline{x_n} = 1 \oplus x_n) \\ &= f_0(x) \oplus (x_n f_0(x) \oplus x_n f_1(x)) \\ &= f_0(x) \oplus x_n f_2(x). \end{aligned}$$

Analogously,

$$f(x) = x_n f_1(x) \oplus \overline{x_n} f_0(x) \qquad (\overline{x_n} = 1 \oplus x_n)$$
$$= f_1(x) \oplus (\overline{x_n} f_1(x) \oplus \overline{x_n} f_0(x))$$
$$= f_1(x) \oplus \overline{x_n} f_2(x).$$

□

A new kind of branching programs was introduced by Kebschull, Schubert and Rosentiel in 1992 [KSR92]. They defined a *ordered functional decision diagram* based on the Reed-Muller decompositon.

Definition 2.3.10 ([MT98]). An *ordered functional decision diagram* (OFDD) is defined like an OBDD with one difference: the function f_v which is computed in a node v of the graph is now defined by the following inductive rules:

1. If v is a sink with label 1 (0), then $f_v = 1$ ($f_v = 0$);
2. If v is a node with label x_i whose 1- and 0- successor nodes represent the functions h and g, respectively, then

$$f_v = g \oplus x_i h.$$

Analogously to the construction of FBDD, a *free functional decision diagrams can be defined.*

Definition 2.3.11. A *free* functional decision diagram is a read-once functional decision diagram.

In all their varieties, decision diagrams can be well represented by graphs. In turn, the graphs remind us networks of communications. Indeed, we shall later see an interesting connection between ordered binary decision diagrams and *communication complexity* that is presented next.

2.4 Communication model

Whenever two or more *parties* (computers, humans, parts of a device) perform a task that can not be accomplished by a single participant, a *communication* gets at handy. A mathematically rigorous study of this setting was first done by A. Yao in 1979 [Yao79].
Let us list several real life examples well described by the communication model.

- An airplane and the ground control. Here we obviously have two parties.
- The Global Positioning System (GPS). This example represents multiparty communication where the end-user and the satellite grouping is involved.
- Very Large Scale Integration (VLSI, microchip) devices. If we cut the chip we obtain two parties, or simply "parts", that communicate exchanging date. The microchip complexity measure can be derived based on this approach. It is widely in use.
- A personal computer as a system of its components. Clearly, it is a multiparty communication. As a matter of of fact, it is narrow communication channels, rather than a slow CPU (Central Processing Unit) or a GPU (Graphics Processing Unit), that cause performance drops in the computers of today.

The intuition behind all these examples is that the parties involved must *exchange information* in order to achieve their goals. Left alone, none of the parties can successfully accomplish their tasks in every example above.
This intuition boils down to a significant theoretical assumption. In communication complexity, we assume that each of the parties possesses the ultimate computational power. In other words, it is only lack of the information that doesn't let the parties come up with the desired solution at once. That's why the parties have to *communicate*. Apart from communication all kinds of computation is *free of charge*.
The most basic kind of communication is where exactly two parties are involved. If these two parties, say *Alice* and *Bob*, decide to compute a Boolean function $f(x_1, \ldots, x_n) \in \mathbb{B}_n$ they would, obviously, split the input. Otherwise, there's nothing

to communicate about, since each of them, Alice and Bob, is an ultimate computing "superpower".

Definition 2.4.1 ([Hro97]). Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of input variables. Any function $\Pi : X \to 1, 2$ is called a *partition of* X. Define two sets: $\Pi_{L,X} := \{x \in X |\ \Pi(x) = 1\}$ and $\Pi_{R,X} := \{x \in X |\ \Pi(x) = 2\}$. Obviously, $\Pi_{L,X} \cup \Pi_{R,X} = X$ and $\Pi_{L,X} \cap \Pi_{R,X} = \emptyset$.

In the context of communications the set $\Pi_{L,X}$represents variables assigned to the "left" party Alice, while $\Pi_{R,X}$represents variables assigned to the "right" party Bob (See Figure 2.6).
We shall have to handle input assignments partitioned between the two computers. In order to be able to make it in elegant fasion we introduce following definition.

Definition 2.4.2 ([Hro97]). Let Π be a partition of $X = \{x_1, \ldots, x_n\}$. Let $\alpha : X \to \{0, 1\}$ be an input assignment. We denote by $\alpha_{\Pi_{L,X}}$ an assignment $\alpha_{\Pi_{L,X}} : \Pi_{L,X} \to \{0, 1\}$, such that $\alpha_{\Pi_{L,X}}$ preserves α. Analogously, $\alpha_{\Pi_{R,X}}$ is an assignment $\alpha_{\Pi_{R,X}} : \Pi_{R,X} \to \{0, 1\}$. We denote by $\Pi^{-1}(\alpha_{\Pi_{L,X}}, \alpha_{\Pi_{R,X}})$ the original assignment α.

The notion of *protocol* provides the abstraction of the *algorithm* concept in communication complexity theory. We shall consider only the protocols computing *Boolean* functions $f_n(X) = f_n(x_1, x_2, \ldots, x_n) \in \mathbb{B}_n$. This kind of protocols is the "corner stone" of the communication complexity theory. Although, analogously to the Turing machines, this model can also be generalized for *arbitrary* functions.
An informal description of the *two-party many round communication protocol* mostly coincides with our intuitive picture of a communication.

> Let Alice and Bob be two parties that jointly compute a Boolean function $f_n(X), X \in \mathbb{B}^n$. A partition $\Pi : X \to 1, 2$ of the input defines how the input is divided between the two parties. Let Alice read only the values for the variables from $\Pi_{L,X}$ and let Bob read only the values for the variables from $\Pi_{R,X}$.
>
> We the following set of rules describes a *two-party communicational protocol* Φ.

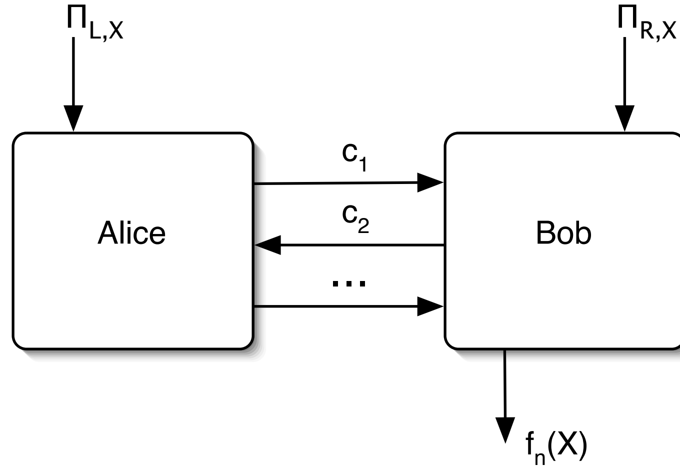

Figure 2.6: A two-party communication protocol.

Alice starts the computation reading the values for $\Pi_{L,X}$.

Alice sends Bob a *message* $c_1 \in \mathbb{B}^{l_1}$.

Bob receives both the input values for $\Pi_{R,X}$ and the message c_1 sent by Alice.

If, having c_1 and values for $\Pi_{R,X}$, Bob can calculate $f_n(X)$,

then Bob gives the output $f_n(X)$;

else Bob sends Alice a message $c_2 \in \mathbb{B}^{l_2}$

Alice If now Alice is able to determine the value of $f_n(X)$

then Alice sends Bob the answer and Bob will output it;

else Alice sends Bob another message $c_3 \in \mathbb{B}^{l_2}$. Alice and Bob keep to exchange messages until they can determine the value $f_n(X)$

Bob Finally outputs the answer: $f_n(X)$.

The description above can be formalized to a a strict mathematical definition presented next. It is strongly based on the formalism proposed by J. Hromkovč [Hro97]. Another approach, based on *binary trees* can be found in the book of E. Kushilevitz

and N. Nisan [KN97]. The latter is close to the original definition given by A. Yao [Yao79].

Definition 2.4.3. Let $f_n(X) \in \mathbb{B}_n$ be a Boolean function with the set of input variables $X = \{x_1, x_2, \ldots, x_n\}$. A *protocol* computing function f for the input X is a pair $D_n = \langle \Pi, \Phi \rangle$.

1. Π is a partition of X;

2. Φ is a *communication function.*

$$\Phi : \{0,1\}^m \times \{0,1,\$\}^\star \cup \{0,1\}^k \times \{0,1,\$\}^\star \to \{0,1\}^+ \cup \{\overline{0},\overline{1}\},$$
$$\text{where } m = \{\Pi_{L,X}\}, k = \{\Pi_{R,X}\} = n - m.$$

There are three conditions that we demand to hold for the function Φ:

(a) *Prefix-freeness.* No message is a prefix of another message, so that no special delimiter symbol is needed. That is, for each $c \in \{0,1,\$\}^\star$ and any two different $\alpha, \beta \in \{0,1\}^m$ $[\{0,1\}^k]$, $\Phi(\alpha, c)$ is not a proper prefix of $\Phi(\beta, c)$;

(b) *Single output party property.* The output value is always computed by the same party independently of the input assignment. That is, if $\Phi(\alpha, c) \in \{\overline{0},\overline{1}\}$ for an $\alpha \in \{0,1\}^m, c \in (\{0,1\}^+\$)^{2p}$ for some $p \in \mathbb{N}$ [for an $\alpha \in \{0,1\}^k$, $c \in (\{0,1\}^+\$)^{2p+1}$], then $\forall q \in \mathbb{N}$, $\forall \gamma \in \{0,1\}^k$, $\forall d \in (\{0,1\}^+\$)^{2q+1}$ [$\forall q \in \mathbb{N}$, $\forall \gamma \in \{0,1\}^m$, $\forall d \in (\{0,1\}^+\$)^{2q}$] it holds that $\Phi(\gamma, d) \notin \{\overline{0},\overline{1}\}$

(c) *End message property.* If computer A computes the output for an input assignment, then computer B knows that, and does not wait for further communication. The latter is achieved by demanding the function Φ to satisfy the following expressions. $\Phi(\alpha, c) \in \{\overline{0},\overline{1}\}$ for some $\alpha \in \{0,1\}^m$, then $\Phi(\beta, c) \notin \{0,1\}^+$ for any $\beta \in \{0,1\}^m$, and $\Phi(\alpha, c) \in \{\overline{0},\overline{1}\}$ for some $\alpha \in \{0,1\}^k$, then $\Phi(\beta, c) \notin \{0,1\}^+$ for any $\beta \in \{0,1\}^k$. For any given communication function Φ, only one of the two expressions above is needed, dependent on the choice of the output party.

We shall discuss all three conditions from the definition quite soon. But first, we introduce some complexity measures for the communications. Complexity, or *resource utilization*, is the central point of interest in *complexity theory*. That is why complexity measures we are going to define shall explain the definition of the *communication protocol*.

2.4.1 Complexity measure

We mentioned above that all computations made by Alice or Bob alone are free of charge. The only resource utilized by the communication protocol is that needed to transfer the messages. Naturally, it is the length of the communicated messages that we call *complexity* of the *two-party deterministic communication protocol* that computes $f_n(X)$.
First we formalize the notion of the "communicated messages". According to the definition of the protocol, messages are determined by the *communication function* Φ. Thus, any message c_i is in $\{0,1\}^+ \cup \{\overline{0},\overline{1}\}$

Definition 2.4.4 ([Hro97]). Let $f_n \in \mathbb{B}_n$ be a Boolean function of n variables in $X = \{x_1,\ldots,x_n\}$. Let $D_n = \langle\Phi,\Pi\rangle$ be a two-party communication protocol over X. A *computation* of D_n on an input assignment $\alpha \in \{0,1\}^n$ is a string $c(D_n,\alpha) = c_1\$c_2\ldots\$c_k\$c_{k+1}$, where

1. $k \geq 0; c_1,\ldots,c_k \in \{0,1\}^+, c_{k+1} \in \{\overline{0},\overline{1}\}$;

2. for each integer $l, 0 \leq l \leq k$, we have

   (a) if l is even, then $c_{l+1} = \Phi(\alpha_{\Pi_{L,X}}, c_1\$c_2\ldots\$c_l\$)$;

   (b) if l is odd, then $c_{l+1} = \Phi(\alpha_{\Pi_{R,X}}, c_1\$c_2\ldots\$c_l\$)$.

Indeed, this definition of *computation* is consistent with the informal description we gave earlier in this section. The party, that reads "left" part of the partition Π starts the communication. Messages are sent synchronously, the parties alternate while communicating. Finally, the output is given.

Definition 2.4.5 ([Hro97]). Let $f_n \in \mathbb{B}_n$ be a Boolean function of n variables in $X = \{x_1, \ldots, x_n\}$. Let $D_n = \langle \Phi, \Pi \rangle$ be a two-party communication protocol over X. We say that D_n *computes* f_n, if for each $\alpha \in \{0,1\}^n$, the computation of D_n on the input assignment α is finite and ends with $\overline{1}$ if, and only if $f_n(\alpha) = 1$. In this case we also say that the computation is *accepting*, otherwise we would call the computation *rejecting*.

Length of a computation c is the total length of all communicated messages excluding \$ symbols and the output value.

Definition 2.4.6. For a protocol D_n, and an appropriate input α, we define *length* of the computation $c(D_{n,\alpha}) = c_1\$c_2 \ldots \$c_k\$c_{k+1}$ as follows

$$|c(D_n, \alpha)| := \Sigma_{i=1}^{k} |c_i|.$$

Finally, we define the *communication complexity* of a two-party protocol as the maximum length of its *computation* over all inputs.

Definition 2.4.7. Let Π be a partition of $X = \{x_1, \ldots, x_n\}$. Let $D_n = \langle \Phi, \Pi \rangle$ be a protocol over X. The *communication complexity* $CC(D_n)$ of the protocol D_n is defined as follows.

$$CC(D_n) := \max_{\alpha \in \mathbb{B}^n} |c(D_n, \alpha)|.$$

The smallest complexity of a protocol that computes a Boolean function defines for the function its communication complexity.

Definition 2.4.8. Let $f_n \in \mathbb{B}_n$ be a Boolean function of n variables, let Π be a partition of $X = \{x_1, \ldots, x_n\}$. Define $CC(f_n, \Pi)$ the *communication complexity* of f_n *according to the fixed partition* Π as follows.

$$CC(f_n, \Pi) := \min\{CC_1(\langle \Phi, \Pi \rangle) \mid \text{for a } \langle \Phi, \Pi \rangle \text{ computing } f_n\}.$$

Consider a function f_n communication complexity according to a partition Π. It is worth to mention, that $CC(f_n, \Pi) \leq n$. In the protocol, that achieves this complexity

bound, the computer A simply sends its part of the input to the computer B. The latter, being an "almighty" supercomputer calculates the function value.

We can also consider defining a communication complexity measure independent from any partition. The reason is it is this measure that has important applications in parallel and distributed computations.

It doesn't make sense to define this measure as the minimum communication complexity over all partitions. There are trivial partitions that would make the measure irrelevant to a problem complexity. We shall consider a class of reasonable partitions, that has following properties.

1. No input variables enter both parts;
2. The number of variables entering one part is not "substantially different" from the number of variables entering the other part.

This thinking leads us to the definition of *balanced* and *almost balanced* partitions.

Definition 2.4.9 ([Hro97]). Let Π be a partition of $X = \{x_1, x_2, \dots, x_n\}$. We say that Π is *balanced* if $||\Pi_{L,X}| - |\Pi_{R,X}|| \leq 1$. We say that Π is *almost balanced* if $n/3 \leq |\Pi_{L,X}| \leq 2n/3$ (Obviously, this implies $n/3 \leq |\Pi_{R,X}| \leq 2n/3$). We also define two sets.

$$Bal\,(X) := \{\Pi | \Pi \text{ is a balanced partition of } X\};$$
$$Abal\,(X) := \{\Pi | \Pi \text{ is an almost balanced partition of } X\}.$$

We define communication complexity of a function as the minimum over all *balanced* or *almost balanced* partitions.

Definition 2.4.10 ([Hro97]). Let $f_n \in \mathbb{B}_n$ be a Boolean function of n variables $x_1, \dots, x_n$,. Let the set $X = \{x_1, \dots, x_n\}$. The *communication complexity of Boolean function* f_n is

$$CC\,(f_n) := \min\{CC\,(f_n, \Pi)\,|\Pi \in Bal\,(X)\}.$$

The *a-communication complexity of Boolean function* f_n is

$$ACC\,(f_n) := \min\{CC_1\,(f_n, \Pi)\,|\Pi \in Abal\,(X)\}.$$

2.4.2 Properties of communication function

Let us now discuss the conditions we set for communication protocol (See page 36). We consider all three conditions of the *communication function* separately.

Prefix-freeness

The first condition is based on a famous fact concerning so-called *prefix codes*, also known as *instantaneous codes*. We recall some basic *coding theory* first.
First, we should agree on what we call *"codes"* in general.

Definition 2.4.11. Let us call a *code* a partial function $g : A \to B^*$ that maps symbols from the alphabet A to words over the alphabet B.

There is a kind of codes that allow *unique decodability* of the initial message. Certainly, this is the most useful sort of codes.

Definition 2.4.12. For A, B – some alphabets, we call a code $g : A \to B^*$ *uniquely decodable*, if g is a surjection.

Definition 2.4.13. A uniquely decodable code with no codeword being a prefix of some other codeword is called *prefix code*.

The name "prefix code" is quite misleading, in fact, it is a "prefix-free code". However that's the settled terminology. It is easy to see that this kind of codes can be decoded "instantly". As soon as the codeword transmission is over the decoder doesn't need any end-of-message sign to map the codeword to a corresponding symbol of the initial source alphabet.
What has the coding theory to deal with communication? In communication theory, the parties exchange messages. The messages must be encoded using some uniquely decodable code. It turns out, that we can always opt for prefix codes without losing much. Here's the fact.

> The *Kraft inequality* tells when the lengths of the code words permit forming an instantaneous code. McMillan proves that the same inequality applies to all uniquely decodable codes [Ham80].

Informally, prefix codes can encode messages as good as arbitrary uniquely decodable codes. In other words, we don't loose in communication complexity. Additionally, the *prefix freeness* offers some technical advantages that can be utilized in proves where communication protocols are considered. Finally, we abstract from the detals of message encoding. The complexity measure of communication protocols (See **Definition 2.4.7, 2.4.8**) is set to reflect resource utilization of the algorithm rather than that of message encoding. The latter is studied by the *theory of codes*.

Single output party property.

Initially defined by A. Yao [Yao79], communication protocol did not have this property. It is straightforward how to modify a protocol, so that for all input strings the output value is given by the same computer. It takes just one bit plus perhaps a special message to send in addition to the communication of the initial protocol. Thus, complexity of protocols is not really affected. In the sense of complexity, **Definition 2.4.3** is equivalent to that of Yao's. However, the *single output property* allows us to formalize the *End message property* without further complicating the definition of the communication protocol. Finally, this property of communication protocols serves for convenience of uniform appearance of the protocols. In fact, often a stronger property is assumed. It demands that for all protocols, and for all inputs, the output is always given by the same computer B that reads the $\Pi_{R,X}$ part of the input X.

End message property

The property ensures that both parties halt, when the output is computed. This is a natural assumption for the model. This property is useful when we define restricted communication models later in this section. Earlier we introduce the *single output property* in order to elegantly formalize the *end message property*.

2.4.3 One-way communications

A restriction of the general two-party communication model allows only one communication round. Alice sends Bob a message c, and that's all about the communication they make.
Study of this model is worthwhile because of the following facts:

1. We later shall see its application to the *branching programs* complexity;
2. It provides lower bounds for VLSI circuits;
3. In many cases it's essentially higher than the *communication complexity.*

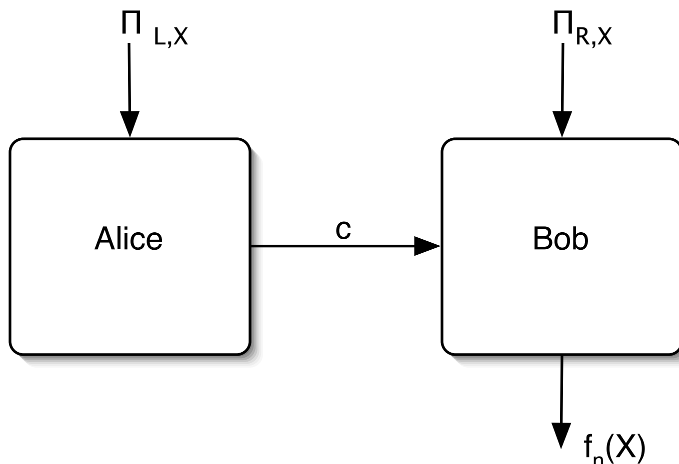

Figure 2.7: A one-way two-party communication protocol.

The informal description of the one-way communication protocol is significantly simpler than the general case described earlier.

> Let Alice and Bob be two parties that jointly compute a Boolean function $f_n(X), X \in \mathbb{B}^n$. A partition $\Pi : X \to 1, 2$ of the input defines how the input is divided between the two parties. Let Alice receive values for the variables from $\Pi_{L,X}$ and let Bob receive values for the variables from $\Pi_{R,X}$.

The following set of rules describes a *one-way two-party communicational protocol* $\langle \Phi, \Pi \rangle$

Alice starts the computation reading the values for $\Pi_{L,X}$.
Alice sends Bob a *message* $c \in \mathbb{B}^l$.

Bob receives both the input values for $\Pi_{R,X}$ and the message c sent by Alice. He must now output the function value $f_n(X)$.

Let us formally define *k-round protocol*, and the *one-way protocol* as a special case. This definition is strongly based on the definition presented by Hromkovič [Hro97].

Definition 2.4.14. Let $D_n = \langle \Phi, \Pi \rangle$ be a protocol computing a Boolean function $f_n \in \mathbb{B}_n$. For every $k \in \mathbb{N}$ a *k-round computation* of D_n is any computation of D_n of the form $c = c_1\$c_2\$\ldots\$c_k\$c_{k+1}$, where $c_i \in \{0,1\}^+$ for $i = 1,\ldots,k$, and $c_{k+1} \in \{\overline{0},\overline{1}\}^+$. That is, the computation contains exactly k messages sent.
We also say that the computation c *has* k *rounds*. The protocol D_n is called a *k-round protocol* if each computation of D_n has at most k rounds. A one-round protocol is also called a *one-way* protocol.

Notice, that a one-way protocol, computing a Boolean function, has either one-round or zero-round computations. But not both kinds. It follows from the *end message property* of communication function (See **Definition 2.4.3**).
Analogously to the general case, we define one-way communication complexity.

Definition 2.4.15 ([Hro97]). Let f_n be a Boolean function of n variables in X for some $n \in \mathbb{N} \setminus \{0\}$. The *one-way communication complexity* of f_n *according to a partition* Π is

$$CC_1(f_n, \Pi) := \min\{CC(\langle \Phi, \Pi \rangle) \,|\, \langle \Phi, \Pi \rangle \text{ is a one-way protocol computing } f_n\}.$$

The *one-way communication complexity* of f_n is

$$CC_1(f_n) := \min\{CC_1(f_n, \Pi) \,|\, \Pi \in Bal(X)\}.$$

The *one-way a-communication complexity* of f_n is

$$ACC_1(f_n) := \min\{CC_1(f_n, \Pi) \,|\, \Pi \in Abal(X)\}.$$

The next proposition is an easy to prove relation between different complexity measures we established so far.

Proposition 2.4.1. *For every Boolean function $f_n \in \mathbb{B}_n$, $n \in \mathbb{N} \setminus \{0\}$ following relations hold.*

$$\begin{aligned} ACC_1(f_n) &\leq CC_1(f_n) \\ ACC(f_n) &\leq ACC_1(f_n) \\ CC(f_n) &\leq CC_1(f_n). \end{aligned}$$

Although, simple and useful, this proposition is not the only tool we have for establishing communication complexity of a function.

2.4.4 Lower bounds

One of the most useful instruments in communication complexity is the notion of *communication matrix*. It allows proving nontrivial lower bounds on general and one-way communication complexity of Boolean functions.

Definition 2.4.16. Let $f_n(X) \in \mathbb{B}_n$ be a Boolean function. Let $X = \{x_1, \ldots, x_n\}$ be the set of input variables. For a partition Π of the set X, define a *communication matrix* of the function f to be a zero-one $2^{|\Pi_{L,X}|} \times 2^{|\Pi_{R,X}|}$ matrix CM.

$$CM_{i,j} := f_n\left(\Pi^{-1}(\alpha_i, \beta j)\right); \qquad i = 1, \ldots, |\Pi_{L,X}|; j = 1, \ldots, |\Pi_{R,X}|;$$

where α_i is ith word from lexicographically ordered set $\{0,1\}^{|\Pi_{L,X}|}$, β_j is jth word from lexicographically ordered set $\{0,1\}^{|\Pi_{R,X}|}$, α_i defines an assignment on $\Pi_{L,X}$, and β_j – an assignment on $\Pi_{R,X}$.

Two powerful lower bound theorems conclude our section on *communication complexity.*

Theorem 5 ([Hro97]). *Let f be an arbitrary Boolean function with a set X of input variables. For any partition Π of X*

$$CC(f) \geq \lceil \log_2 \mathrm{Rank}(CM(f, \Pi)) \rceil .$$

Similarly to many-round protocols, we can link complexity of a one-way protocol to the communication matrix of the function computed by the protocol. But this time the complexity can be calculated exactly!

Theorem 6. *For a Boolean function f defined over a set X of input variables, for a partition Π of the input X*

$$CC_1(f, \Pi) = \lceil \log_2 NRow(CM(f, \Pi)) \rceil ,$$

where $NRow(f)$ is the number of different rows in $CM(f, \Pi)$.

Proof. We shall argue by contradiction. Let CM be the communication matrix of f according to the partition Π. Without loss of generality, let the rows CM_1 and CM_2 are not equal. However, we assume that the *communications* corresponding to the two rows coincide. Let $\langle \Phi, \Pi \rangle$ be a protocol computing f.

$$\Phi(bin(i), \lambda) = \Phi(bin(j), \lambda) \Rightarrow$$
$$\forall \gamma \text{ an assingnment of } \Pi_{R,X} \quad \Phi(\gamma, \Phi(bin(i), \lambda)) = \Phi(\gamma, \Phi(bin(j), \lambda)) \Rightarrow$$

since $\langle \Phi, \Pi \rangle$ computes f

$$\Rightarrow f(\Pi^{-1}(bin(i), \gamma)) = f(\Pi^{-1}(bin(j), \gamma)),$$

where λ is the empty symbol.
The last expression, however, contradicts the fact that $CM_i \neq CM_j$. The theorem follows. □

We shall recall one of these theorems later on, providing yet another evidence of how useful, yet simple and elegant the communication model is.

Chapter 3

Nondeterministic and Randomized Models

> I am always doing that which I can not do, in order that I may learn how to do it.
>
> ---
>
> Pablo Picasso

Up to this moment, we have considered models that made deterministic choices on each step of their programs. The Turing machines, the branching programs, even the communication protocols. Their output could always be predicted having the input and the algorithm. Now we are up to add some juice to this boring behavior!

3.1 Nondeterministic Turing machines

Unlike it has been the case with all other models, there is no real life device behind the notion of the *nondeterministic Turing machine*. Instead, there is a plenty of real-life problems, that have one common feature. It is not always clear whether there is an *efficient*, that is polynomial time and space, solution of the problem. But given a candidate solution, we can *efficiently* check whether it is correct. As we shall see later, classical Turing machines can always offer an exponential time algorithm for

an arbitrary problem of this kind. However, it is an open question whether they can fare better. Thus, the classical Turing machines offer a poor means to characterize the class of the problems.
What kind of computational problems have we been talking about? Let's give some examples.

Example 9 (Easily checkable solutions). A very short list of problems that have easily checkable solutions, but that may be very hard to solve.

- *Proofs* in logic. It is hard to come up with a proof for a logical statement. However, we can usually efficiently check the proof for correctness.

- *Combinatorial optimization* problems. Finding a solution may involve an extensive search of the notorious needle in the notorious haystack. But it is easy to check if an instance satisfies the search criterions, or the linear programming constraints.

- *Knowledge base*, or *database* applications. For the same reason as for combinatorial optimization problems.

All these problems can be nicely put in one complexity class defined based on the notion of the *nondeterministic Turing machine*.

Definition 3.1.1. A *nondeterministic Turing machine* is a quadruple (K, Σ, Δ, s), where K, Σ, s are defined the same way as for classical Turing machines. The *program* Δ is a relation, also called a *transition relation*, $\Delta \subset (K \times \Sigma) \times [(K \cup \{h, '' yes'', '' no''\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}]$. That is, for each configuration of the nondeterministic Turing machine there can be more than one configurations it could be in the next step!

Definitions of a *configuration*, an *initial configuration* and a *computation* of the Turing machine remain the same for the nondeterministic machines (See **Def. 2.1.2**, **Def. 2.1.4**, p. 12). However, now more than one computation may correspond to a given input. In fact, for every input, there is a *computation tree*, where the paths from the root to the leafs are the appropriate computations.

Definition 3.1.2 ([Pap94]). We say that a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ *decides* a language L if for any $x \in \Sigma^*$ $x \in L$ if and only if there is a computation of N on the input x that ends in the accepting state "yes", that is $(s, \triangleright, x) \stackrel{N^*}{\to} (\text{"yes"}, w, u)$, for some strings w and u.

A nondeterministic Turing machine accepts the input if there exists an accepting computation. It rejects the input if all computations on the input end in the rejecting state. The asymmetry is due to our desire to define the classes of problems, according to the difficulty of *checking their solutions*, rather than the difficulty of computing an actual solution. Definition of the time complexity reinforces this observation.

Definition 3.1.3 ([Pap94]). Let f be a function from the nonnegative integers to the nonnegative integers. We say that a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ decides a language L *within time bound* $f(n)$ if N decides L, and on every input $x \in \Sigma^*$, every computation of N contains less than $f(|x|)$ steps, that is $(s, \triangleright, x) \stackrel{N^k}{\to} (q, w, u) \Rightarrow k \leq f(|x|)$. We use $\mathbf{NTIME}\,(f(n))$ to denote the *nondeterministic complexity class* of problems decided by nondeterministic Turing machines within time $f(n)$.

The most famous of nondeterministic complexity classes is $\mathbf{NP} := \cup_{k \in \mathbb{N}} \mathbf{NTIME}\left(n^k\right)$. The abbreviation "NP" stands for "Nondeterministic Polynomial". This is exactly the complexity class we started this section with. It contains all problems, such that, given a solution, we can efficiently check whether it is correct or not, but nothing is said about how difficult it may be to actually calculate the solution. Let's give an example.

Example 10 (Quadratic congruences). Instance: Positive integers a, b, and c.

Question: Is there a positive integer $x < c$ such that $x^2 \equiv a(\mod b)$?
Given a, b, c and x we can check whether $x^2 \equiv a(\mod b)$ in polynomial time with a Turing machine. But it is absolutely not clear how to find such $x < c$ efficiently!

We should notice, that the Turing machine is a special case of the nondeterministic Turing machine where the relation Δ happens to be a function. Next proposition is the consequence of this reasoning.

Proposition 3.1.1.

$$\mathbf{P} \subseteq \mathbf{NP}.$$

Finally, we show that *nondeterministic Turing machines* do not offer extra power in terms of computability. However, simulating a nondeterministic Turing machine by classical Turing machines can be exponentially hard.

Theorem 7. *For a function $f : \mathbb{N}^+ \rightarrow \mathbb{N}^+$, and some constant c*

$$\mathbf{NTIME}\left(f(n)\right) \subseteq \mathbf{TIME}\left(c^{f(n)}\right)$$

Proof. Let L be a language from $\mathbf{NTIME}\left(f(n)\right)$. There is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ that decides this language within time $f(n)$. For each pair (q, σ) consider a set of choices $C_{q,\sigma} := \{(q', \sigma', D) | ((q, \sigma), (q', \sigma', D)) \in \Delta\}$. Let $d := \max_{q \in K, \sigma \in \Sigma}\{|C_{q,\sigma}|\}$ be the *degree of nondeterminism* of N. Now, consider sequences $(c_1, c_2, \ldots, c_t)$ of integers $c_i \leq d$ in the order of increasing t.
We shall describe a Turing machine M, simulating N. The simulation starts with storing current sequence $(c_1, c_2, \ldots, c_t)$ on a separate string. Then M simulates t steps of N. If an accepting configuration is encountered, M accepts the input. If a halting configuration is encountered, M proceeds with the next choices sequence of the same length. If all t-element choice sequences lead to a halting configuration, M stops the simulation and rejects the input. According to the definition of the nondeterministic Turing machine, M will halt. Also it will decide the same language L. It remains to notice that M will need at most $\sum_{t=1}^{f(n)} d^t = O(d^{f(n)+1})$.

□

We shall get back to the Earth in the next section, where we consider a wired, but a realistic, computational model

3.2 Randomized Turing machines

The nondeterministic Turing machine is unrealistic, because it can perform arbitrarily many computations with no extra resource utilization. In real world we have to pay

for everything. If we accept the probability of false output as a price we pay for the multitude of possible computation, we can define a *randomized Turing machine*. For example, let us toss a coin every time we confront a nondeterministic choice. There will be a certain probability assigned to each computation. We also may limit the number of accepting computations from below. This model is totally realistic, just due to the fact that we actually described its implementation. First step towards formally describing randomized Turing machines is introducing a *standard* nondeterministic machine.

Definition 3.2.1. Let N be a polynomial $p(n)$-time bounded nondeterministic Turing machine. Without lost of generality we can assume that, for all inputs x, all computations of N are of the same length $p(|x|)$. We also assume that on each step there are exactly two non-deterministic choices. We call N a *standard nondeterministic Turing machine.*

Next we define our first randomized model of computation.

Definition 3.2.2. Let N be a *standard* nondeterministic Turing machine. Let $p(n)$ be the polynomial time bound of N. Let also for any input x, either at least half of the $2^{p(|x|)}$ computations of N on x halt in accepting state "yes", or all of them halt in the rejecting state "no". Then N is called a *polynomial Monte Carlo Turing machine.*

We say that N *accepts* a word x if there is an accepting computation of N on x, otherwise we say that N *rejects* x. A language L is *recognized* by N if $x \in L \iff N$ accepts x.
The class of all languages recognized by polynomial Monte Carlo Turing machines is called **RP** (for *randomized polynomial time*).

First note, that any arbitrary polynomial time nondeterministic Turing machine can be transformed into the *standard* form. Then, the definition above, indeed, captures the intuitive notion of the Monte Carlo algorithm:

- A fair coin flip is made on each step of the computation.
- The probability of accepting a word not in the recognized language L is zero.

- The probability of rejecting a word in L is less than one half.

Due to the equal probabilities of the coin flip procedure, outcomes on each step, for every input x, every valid computation has the same probability $1/2^{f(|x|)}$. In our definition, for any $x \in L$, at least half of the computations end up in the accepting state "yes". Thus, the accepting probability is indeed at lest one half!
It is clear that the definition of a language *recognized* by a Monte Carlo Turing machines is consistent with the definition of a language *decided* by a nondeterministic Turing machine. Thus, following proposition is true.

Proposition 3.2.1.

$$\mathbf{RP} \subseteq \mathbf{NP}.$$

Recall the definition of *compliment class of languages* (**Def. 1.2.7, p. 8**). It is natural to define a class **coRP** of languages that can be recognized by a "flipped" Monte Carlo Turing machine, where the accepting and the rejecting configurations are interchanged. Next proposition follws from the **Proposition 3.2.1** and **Lemma 1** (Page 8).

Proposition 3.2.2.

$$\mathbf{coRP} \subseteq \mathbf{coNP}.$$

The purpose of "flipping" the Monte Carlo Turing machine was to continue our casino tradition. Now we can easily describe *Las Vegas* algorithms. We start with the appropriate complexity class definition.

Definition 3.2.3.

$$\mathbf{ZPP} := \mathbf{RP} \cap \mathbf{coRP}.$$

For a language $L \in \mathbf{ZPP}$ there are two Monte Carlo Turing machines: a machine N_1 that accepts L and a machine N_2 that accepts $\overline{L}$. A *Las Vegas* algorithm LV runs both of the machines on the input x. The output of the Las Vegas algorithm is chosen according to the following rule.

- If N_1 accepts x then LV accepts x,

- if N_2 accept x, then LV rejects x,
- if both N_1 and N_2 reject the input, then it is not clear if $x \in L$ or $x \notin L$. So the algorithm simply halts in the h state. The latter case is equivalent to the "don't know" output.

One can run the Las Vegas algorithm LV until one of the definitive answers "yes" for L or for $\overline{L}$ is obtained. In this sense the algorithm is *Zero error Probabilistic*, and *Polynomial* due to N_1 and N_2 are polynomial machines. That gives the abbreviation "ZPP". The "Las Vegas" stands for the fact that the algorithm is never wrong. Just like the casino always wins.

3.3 Unrealistic probabilistic Turing machines

Previously, we introduced probability to nondeterminism in order to obtain a feasible kind of nondeterministic Turing machine. However, it can work all the way around. We next introduce a probabilistic computation that is more general than the nondeterministic computation.
As all unrealistic computational abstractions, this one also doesn't come from the real-life devices. Instead, just like the nondeterministic Turing machine, it originates in a nontrivial problem given in the example below.

Example 11 (MAJSAT). Instance: A Boolean expression with n free binary variables.

Question: Is it true that majority of the 2^n assignments satisfy the formula?

Is there a polynomial time nondeterministic Turing machine for $MAJSAT$? There is no obvious answer to this question. However, there is a generalization of the standard (polynomial time) nondeterministic Turing machine that *decides MAJSAT by majority.*

Definition 3.3.1. Let N be a standard nondeterministic Turing machine. If more than half of the computations of N accept the word x, then we say that N *accepts* x

by majority. Let L be a language. If, for any word x over the input alphabet $x \in L$ if, and only if, the machine N accepts x by majority, we say that N *decides* L *by majority.*

The class of all languages decided by majority is called **PP** (for probabilistic polynomial).

Why is this model unrealistic? One reason is that it actually tests an exponential amount of truth assignments of *MAJSAT* in polynomial time! The other reason is that this model is more powerful than the nondeterministic Turing machine.

Theorem 8.

$$\mathbf{NP} \subseteq \mathbf{PP}.$$

Proof. Let N be a nondeterministic polynomial time Turing machine that decides a language L. We shall construct a standard nondeterministic machine N' that will *decide* L *by majority.*

Let N' start the computation from a nondeterministic choice between the two parts:

1. A standardized version of N.

2. A computation with the same number of steps as the other part, but here all computations end up in the accepting state. This part must also have exactly two nondeterministic choices on every step, because we construct a standard nondeterministic Turing machine.

The total number of different computations of N' will be $2^{p(n)+1}$, where $p(n)$ is the polynomial time bound of the standardized version of N. The machine N' would accept a word x by majority if and only if there is an accepting computation of N on x, because exactly $2^{p(n)}$ computations of N' are always accepting. □

It is somewhat exciting how little we change to obtain may be the most "comprehensive yet plausible" [Pap94] notion of the realistic computation introduced next. Another surprise, it is *probabilistic* too!

3.4 Realistic yet probabilistic Turing machines

Let us slightly modify the definition from the previous section.

Definition 3.4.1. Let N be a standard nondeterministic Turing machine. For all inputs x, N has either $3/4$ of all its computations in the accepting state, or $3/4$ of all its computations in the rejecting state.

- If for some x at least $3/4$ of all of the computations of N accept x, we say that N *accepts* x by *clear majority*.
- If for some x at least $3/4$ of all of the computations of N reject x, we say that N *rejects* x by *clear majority*.

Let L be a language, such that $x \in L$ implies N accepts x by clear majority, and $x \notin L$ implies N rejects x by clear majority. We say that N *decides* L *by clear majority.* The class of all languages *decided by clear majority* is called **BPP** (for bounded error probabilistic polynomial).

The following proposition must be absolutely clear.

Proposition 3.4.1.

$$\mathbf{RP} \subseteq \mathbf{BPP} \subseteq \mathbf{PP}.$$

Although, we gave several reasons for **PP** to be unrealistic, we have never shown why **BPP**, in contrast, is feasible. We already gave several examples of computations where an error probability was involved. However, we did not discuss how we can make use of a computer that may give an erroneous output!

Let N be a standard nondeterministic Turing machine deciding a language L *by majority*, that is $L \in \mathbf{PP}$. Can we for any $x \in L$ arbitrarily reduce the probability of N rejecting it?

Consider a standard nondeterministic machine N' that simply makes $\log_2 t$ nondeterministic choices. There will be exactly t different computations of N'. Append a copy of N to the end of each of the t computations of N' to get another standard nondeterministic Turing machine N''. By the definition of the *deciding by majority,*

N'' would accept a word x by majority if, and only if, N accepted the word x by majority. What is the probability of N'' to reject a word $x \in L$?
Consider a word $x \in L$. Let $p := 1/2 + \epsilon$ be the probability of N to end up in the accepting state on the input x. Then by the *Chernoff bound* (See p. 153, **Cor. 5**), the probability that more than half of the t copies of N reject x is at most $e^{-\epsilon^2 t}$. That is, the probability that N'' rejects x is at most $e^{-\epsilon^2 t}$.
Now if we have to arbitrarily reduce the error probability for a Turing machine N that decides L by *clear majority* we need to take $t \geq \frac{1}{\epsilon^2} = 16$. The margin ϵ equals $1/4$ in this case. Moreover, if we want the error to decrease linearly with $|x|$, we can achieve that combining $t = |x|$ copies of N. However, if N did not have to decide L by *clear* majority, ϵ could be as small as $2^{-p(n)}$ for the polynomial time bound $p(n)$ of the machine N. That implies that we need $O(2^{p(n)^2})$ number t of copies of N in order to achieve at least constant small error probability. And that's why this is infeasible!
Class **PP** is so "big" that contains almost all classes that we defined so far. However, a Turing machine can not use asymptotically more space than time. This statement follows from the observation that a Turing machine writes only a constant number of symbols on each step of its computation. For nondeterministic Turing machines we stick to the same principle. We count only space used by the worst computation consistent with the program. Thus, following statement is evident.

Proposition 3.4.2.

$$\mathbf{PP} \subseteq \mathbf{PSPACE}.$$

As we can see, a polynomial space Turing machine with unbounded time is at least as powerful as a nondeterministic Turing machine that decides by majority. The conclusion follows: a model does not have to be nondeterministic to be infeasible. On the other hand, even nondeterministic models can be feasible, just like a nondeterministic Turing machine that decides by clear majority is!
The first model of the probabilistic Turing machines was introduced in the middle of the XX century. Current definition of the probabilistic Turing machine was given by Gill in 1972. He also defined all polynomial time probabilistic complexity classes presented in this chapter [Gil72, Gil77].

3.5 Randomized branching programs

As we can see, randomized computations have been around as long as branching programs. Both introduced in the late 1970s [Gil77, Mas76]. However, it was not before 1996 that first randomized branching program was introduced by Ablayev and Karpinski [AK96].
Recall how we defined a branching program (**Def. 2.3.2**, p. 27). If we add to this definition that unlabeled nodes with two outgoing edges are allowed, we obtain the *nondeterministic* branching programs.

Definition 3.5.1. A *nondeterministic branching program* or a *nondeterministic binary decision diagram* is a directed acyclic graph with exactly one root, whose

- sinks are labeled by the Boolean constants 0, 1, and whose
- internal nodes are of two kinds:
    1. labeled by a variable x_i, and have exactly two outgoing edges, a 0-edge and a 1-edge.
    2. unlabeled and have exactly two outgoing edges. The latter are called *guessing nodes*.

Each assignment to the input variables x_i corresponds to several *computation paths* that connect the root with one of the sinks. There may be several paths due to the guessing nodes.
A nondeterministic branching program represents a Boolean function g in the following way. For each input assignment to the variables x_i the function $g(x_1, \ldots, x_n) = 1$ if, and only if, there is a computation path corresponding to this assignment that leads to the 1-sink.

We give an example to illustrate this definition.

Example 12. Consider a Boolean function defined below.

$$g(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4).$$

The nondeterministic branching program that represents this function is shown in **Figure 3.1**.
Is there a deterministic branching program that is as succinct as the nondeterministic counterpart? It could be a very good way to occupy quite a bit of the vacant time attempting to answer that question.

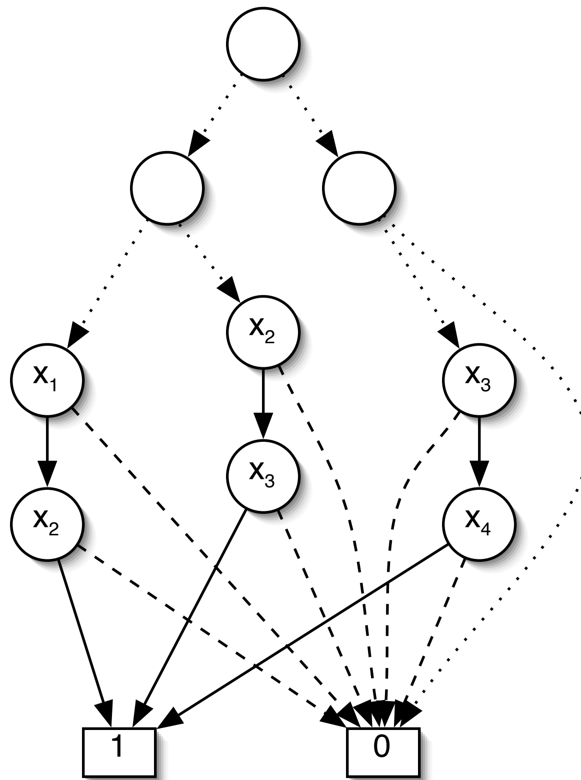

Figure 3.1: A nondeterministic branching program for $g(x)$.

Let us define *size* – the complexity measure of branching programs – for the nondeterministic case. It agrees with the definition of the *size* in the deterministic case.

Definition 3.5.2. The *size* of a nondeterministic branching program is the number of its internal nodes, that is, the sum of the number of *computational nodes* and the number of *guessing nodes*.

Example 13. The class of *functions* that can be represented by some nondeterministic branching program of *polynomial* size is called **NP-BP**

Let us define probabilistic versions of the branching program. We could do it in the same manner as we did it for the Turing machine, avoiding ever mention about probability. In that case, we would have to "pad" some computation paths with additional guessing nodes before connecting them to the sinks. However, branching programs are as much a computational model as they are a *representation type*. Introducing probabilities in the definition leads to reduced size of the representations. That is, graphs of smaller sizes. Nevertheless, the parallel with the corresponding definitions of the probabilistic Turing machine computations must be apparent.

Definition 3.5.3. Let P be a nondeterministic branching program on n variables. We assign a probability $1/2$ to each outcome of the nondeterministic choices that P does. Let $P(x)$ be the random variable that represents the sink value of the computation path P takes on x.

1. We say that P represents a Boolean function $f \in \mathbb{B}_n$ with *one-sided error* if for every input assignment

    (a) the value $f(x) = 1$ implies that $\Pr[P(x) = 1] \geq 1/2$,

    (b) and $f(x) = 0$ implies that all of the computation paths corresponding to the assignment lead to the 0-sink.

2. We say that P represents a Boolean function $f \in \mathbb{B}_n$ with *one-sided error for 0-sink* if for every input assignment

    (a) the value $f(x) = 0$ implies that $\Pr[P(x) = 0] \geq 1/2$,

    (b) and $f(x) = 1$ implies that all of the computation paths corresponding to the assignment lead to the 1-sink.

3. We say that P represents a Boolean function $f \in \mathbb{B}_n$ with *unbounded error* if for every input assignment to x the probability $\Pr[P(x) = f(x)] > 1/2$.

4. We say that P represents a Boolean function $f \in \mathbb{B}_n$ with *bounded error* if for every input assignment to x, the probability $\Pr[P(x) = f(x)] \geq 3/4$.

We can now define branching program versions of already familiar probabilistic time complexity classes **RP**, **ZPP**, **PP** and **BPP**.

Definition 3.5.4.

1. The class of functions represented by nondeterministic branching programs of *polynomial size* with *one-sided error* is called **RP-BP**.
2. The class of functions represented by nondeterministic branching programs of *polynomial size* with *one-sided error for* 0*-sink* is called **coRP-BP**.
3. The class of *zero error polynomial size probabilistic* branching programs is called **ZPP-BP** := **RP-BP** ∩ **coRP-BP**.
4. The class of functions represented by nondeterministic branching programs of *polynomial size* with *unbounded error* is called **PP-BP**.
5. The class of functions represented by nondeterministic branching programs of *polynomial size* with *bounded error* is called **BPP-BP**.

Let us note that a standard procedure of error reduction is applicable to the nondeterministic branching programs as well. We introduced it earlier when discussing the feasibility of **PP** and **BPP** (See page 56).
The relation between general branching probabilistic programs complexity classes is somewhat surprising. It gives us the lesson that *analogy* is indeed a reasoning of *probability*.

Theorem 9 ([Weg00]). P-BP=ZPP-BP=RP-BP=BPP-BP.

We can introduce restricted versions of the nondeterministic branching programs similar to the definitions for the classical case (**Def. 2.3.6, 2.3.8**, p. 29). Ordered binary decision diagrams are of particular interest for us in this study. Notice, that the branching program of the **Figure 3.1** is, in fact, a nondeterministic ordered binary decision diagram. We present the relations between polynomial size OBDD probabilistic complexity classes next.
Clearly, the following proposition is true.

Proposition 3.5.1.

$$\mathbf{NP\text{-}OBDD} \subseteq \mathbf{PP\text{-}OBDD}.$$

Next statement is a special case of the theorem proved by Sauerhoff [Sau99]. It shows that probabilistic OBDD complexity classes exhibit behavior similar to that found in Turing machines.

Theorem 10 ([Sau99]).

$$\mathbf{RP\text{-}OBDD} \subseteq \mathbf{BPP\text{-}OBDD}.$$

Proof. Let P be a nondeterministic branching program that represents a function f with *one-sided error*. We can build a new nondeterministic program P' following way. The program P' starts with a subroutine P_1' - a complete binary tree that consists entirely of guessing nodes. We shall use it to generate desired probabilities of two further tracks of the computation.
Let w be the number of leaves in P_1'. Let t be any integer, such that $\frac{3}{4}w \leq t \leq w$. We replace t of the leaves with the *root* of P, and the remaining leaves we replace with the 1-sink.
The two cases below follow directly from the definition of P.

- If $f(x) = 0$ then
$$\Pr\left[P'(x) = 0\right] = 1 - \frac{w-t}{w} \geq \frac{3}{4};$$
- If $f(x) = 1$ then
$$\Pr\left[P'(x) = 1\right] \geq \frac{w-t}{w} + \frac{3t}{4w} = \frac{4w-t}{4w} \geq \frac{3}{4}.$$

Thus, P' represents f with *bounded error*.
Notice that the size of the tree of P_1' is polynomial in the size of the input. The reason is that the number of its leaves w is polynomial, which is in turn, implied by P had a size polynomial in the length of the input. $\square$

In 1999 Karpinski and Mubarakzjanov proved a beautiful and surprizing result [KM99]. We introduce it next.

Theorem 11 ([KM99]).

$$\mathbf{P\text{-}OBDD} = \mathbf{ZPP\text{-}OBDD}.$$

As we can see, the complexity classes structure may vary for different models. It is this versatility, that helps us to understand true "hardness" of the problems. Perhaps, otherwise it is too coarse of a measure only to relate a problem to a single complexity class, and assume that would completely reflect complexity of the problem.
There are more relationships between OBDD complexity classes, that can be seen without difficulty.

Proposition 3.5.2.

$$\mathbf{BPP\text{-}OBDD} = \mathbf{coBPP\text{-}OBDD} \subseteq \mathbf{PP\text{-}OBDD} = \mathbf{coPP\text{-}OBDD};$$

The models we have introduced so far, have all had a purpose. Either we captured complexity of a class of problems, or formalized existing computational device. However, we haven't yet paid enough of attention to the physical reality. The latter obeys the laws of quantum mechanics, and so should we. Next chapter is only a small introduction to the mind boggling field. The *Quantum computations*.

Chapter 4

Quantum Computations

> God doesn't play dice with the Universe
>
> ---
>
> Albert Einstein,
> in the letter to Max Born

4.1 Invention of quantum mechanics

The Universe is a remarkably intricate creation. Live mathematics. Like a schoolchild at arithmetics lesson, the mankind strived to understand it. But like a good teacher, it always had more exercises to offer. The history goes back to Max Plank, who in 1900 introduced the constant h that was later named the *Planck's constant*. For example, the *energy* E that can be carried by a beam of light with a constant frequency ν can take only discrete set of values!

$$E = nh\nu, n \in \mathbb{N}$$
$$h \approx 6.6260693(11) \times 10^{-34} J \cdot s.$$

In 1913 a similar discrete behavior was discovered by Niels Bohr, a professor at Copenhagen, for the radiation emission and absorption spectra of atoms. Just a

few years earlier Rutherford introduced his planetary model of atom. Bohr linked the quantum hypothesis of Plank to the planetary model of Rutherford. He explained that there were *stationary orbits* where electrons could remain without emitting radiation, and that they could make *quantum leaps* instantly moving from one stationary orbit to another. The electrons either emitted or absorbed energy during quantum leaps. Then the planetary model could be explained by the only *countable set* of the energy values that an electron could take. Each of the values was called *"quantum"* that comes from latin *"quantus"* for *"how much"*.

A mathematician, Arnold Sommerfeld was a student of Felix Klein in Göttingen. In 1906 Röntgen persuaded him to take the Theoretical Physics chair in Munich. There he came in contact with the *special theory of relativety* of Albert Einstein. However, since 1911 he grew interested in the foundations of the theory began in the works of Niels Bohr and Max Plank.

Apart from his talent in mathematics, physics and teaching, Sommerfeld had a magnetic personality. Max Born told a story about a graduate zoologist who switched to theoretical physics after a lecture of Sommerfeld. Maybe this explains the numerous Nobel prize winners that happend to be students of this teacher. One of them was Wolfgang Pauli. In 1918 he left his home in Vienna in order to study physics in Munich. At that time, the eighteen years old Pauli already new the works of Einstein.

Another great student of Sommerfeld was to be Werner Heisenberg. His fate to become a scientist of some sort was determined already when he was born. His father was a professor of Byzantine literature in Munich, his grandfather specialized in Homer, and his godfather was Ernst Mach.

In 1920 Werner was going to enter the University of Munich. His father arranged him a meeting with a mathematician Ferdinand von Lidenmann. To the total distress of whole family, the famous mathematician said that the young man "was lost for mathematics". That determined the future of one of the greatest physicists of the XX century. He decided to try himself in physics. It had some mathematics, didn't it?

In his first conversation with Heisenberg, Pauli called the atomic theory taught by Sommerfeld *"atomystique"*. So novel and unexplored remained the theory of Niels

Bohr. But both of the young physicists found it intriguing enough to seek for a meeting with Bohr himself. In 1921 Heisenberg went to attend lectures that Bohr gave in Göttingen. Pauli spend the autumn of 1922 in Copenhagen. There was the cradle of the "atomystique". The term *quantum mechanics* did not yet exist. As did not yet the quantum mechanics.

Since his time in Munich, Heisenberg worked on a theory that would depend only on observable values. Thus, he hoped to resolve difficulties with interpretation of the early "quantum theory". A question disturbed him: "Is there such a thing like orbits of electrons?". In may 1924, suffering from the hey fever, Heisenberg moved to an island in the North of Germany. There he worked on theory that would depend only on *observable* values. Finally he managed to find an analog of the *Bohr-Sommerfeld conditions* on the shapes of electron orbits. Remember, there was to be no mention of orbits in the new theory!

The first draft of the article went to Hamburg, to the old pal Pauli. They stayed in an active correspondence for two more weeks. Finally, Pauli called the result *"das Morgenröte der Quantentheorie"*. After that, on the 25th of June 1925 Heisenberg submitted the article [Hei25]. That date is now widely considered as the Birthday of *quantum mechanics.* The term was already coined in by Max Born in 1924 [Bor24]. The article found its way to Paul Dirac in Cambridge. There he improved the result of Heisenberg so that the new theory did not need classical mechanics for its definition anymore [Dir25]. Another response came from Max Born who noticed that the formalism of Heisenberg is identical to the matrix calculus. The same year he published a paper [BJ25] in collaboration with young Pascual Jordan, who just had got his PhD in Göttingen. There they presented the *"matrix mechanics".*

Pauli was not very fond of the "Gelehrsamkeitsschwall der Göttingen", a sample of what the "matrix mechanics" was. In order to settle the growing intolerance of his position by Heisenberg, in 1926 he published an article of his own [Pau26]. As Max Born noticed in his Nobel Lecture in 1954:

> From this moment onwards there could no longer be any doubt about the correctness of the theory.

The new theory creation was finished. It was accepted by Bohr and always skeptical Pauli. However, Einstein was not happy with the new kind of physics his discoveries provided an inspiration for. In a discussion with Heisenberg he said:

> Principally, it is an absolute nonsense to build a theory based only on observable values. Because in reality, it is, in fact, all the way arround. Only theory decides, what exactly can be observed.

Very timely Erwin Schrödinger published his article [Sch26b], where he returned hope for the continuity in the atom world. That was the first attempt to kill the just born quantum mechanics. Schrödinger himself did not intend his discovery that way, but there were many older generation physicists, who did. Including Einstein. However, already about two months later [Sch26a] Schrödinger noticed that:

> A close internal relationship is found between Heisenberg's quantum mechanics and my wave mechanics. Formally, it should be considered as the equivalence of the bouth theories.

In 1926, to the further distress of Albert Einstein, Max Born gave the now standard *probabilistic interpretation* of quantum mechanics [Bor26]. Till the 1950s the old friends argued about this interpretation. In 1954 Born won the Nobel Prize for something that Einstein could not agree with.

The year 1926 Heisenberg was an assistant of Niels Bohr in Copenhagen. Schrödinger visited them before joining Max Plank in Berlin. The three tried to agree on a common interpretation of the atomic events that would explain the observed reality. Unfortunately their positions were different enough for them to arrive at no conclusion. Eventually Schrödinger caught severe cold. Mrs. Bohr took care after the young man, but Mr. Bohr would sit at his bad and continue the unfinished argument.

One of the issues that Bohr and Heisenberg had to deal with was the Wilson's camera experiment. There one vividly could see the trajectory of an electron. In contrast to that the new theory pictured an electron more like a cloud surrounding the atom nucleus. The cloud simply was not supposed to have a particle-like trajectory! Heisenberg reasonably observed that the bubbles of water in Wilson's camera are

much larger than an electron should be. Perhaps, it was not the trajectory but a discrete set of not completely certain positions of the electron that was observed in the experiment. He came up with a question:

> Can we describe a situation where an electron is approximately in a certain place, that is with some fixed inaccuracy, and also has approximately certain speed with some fixed inaccuracy? Could we make these inaccuracies small enough to explain the Wilson's camera experiment?

This reasoning lead him to the famous *uncertainty principle* [Hei27]. Let a physical system is described by its *position* and *momentum.* Let Δx be the *standard deviation* of the *position* measurement, and Δp be the *standard deviation* of the momentum measurement. Then it holds that

$$\Delta x \Delta p \geq \frac{\hbar}{2},$$

where $\hbar = \frac{h}{2\pi}$, and h is the Plank's constant.

Or as Heisenberg himself formulated it:

> The more precisely the position is determined, the less precisely the momentum is known in this instant, and vice versa.

This principle additionally set a bridge between the "wave mechanics" of Schrödinger and the theory of Heisenberg-Born-Jordan. Indeed, if we consider a wave, we can either determine its exact position at a moment of time, or its frequency. But not the both, since we need time at least as long as the *period* of the wave in order to measure the frequency! In the second case, exact position has no sense, that's why we can not measure it. In fact, the uncertainty principle is an absolutely natural and intuitive.

In 1928 Niels Bohr discovers his *complementarity principle.* The essence of this principle is that such phenomena as light and electrons behave sometimes wavelike and sometimes particle-like. It is impossible to observe both the wave and the particle aspects at the same time. However, full knowledge of the small-scale phenomena consists of the both parts, and it is incomplete otherwise.

The latter two discoveries laid ground to the so called *Copenhagen interpretation* of the quantum mechanics. It is the most widely accepted point of view. However, other interpretations also exist. As well as a specific point of view famous due to Feynman. He proposed: "shut up, and calculate!" – indeed, why would we need any interpretation in the first place?
Finally, in 1932 von Neumann formulated the rigorous mathematical basis for the Quantum mechanics as the theory of linear operators in Hilbert spaces [vN32]. We shall give an introduction to this simple, despite of its reputation, mathematical framework in the next section.

4.2 Introduction to quantum mechanics

Earlier we learnt that Heisenberg and Schrödinger both arrived at the same theory. But they presented it in different ways. If Heisenberg's formalism was identical to *matrix* calculus, the "wave mechanics" of Schrödinger was formulated in *differential* calculus. We shall choose Heisenberg-Born-Jordan's representation of quantum mechanics. Essentially, quantum mechanics is a set of simple rules in the language of linear algebra. However, the subject possesses its own notation, different from standard mathematical. This notation was introduced by Paul Dirac in 1930 [Dir30] in order to simplify the routine calculations that had to be done in plenty by physicists. It turned out so convenient that it was immediately adopted. The notation is presented in the **Table 4.1**, courtesy of Nielsen and Chuang [NC00].
Let us introduce the *postulates* of quantum mechanics. Unlike in most of the physical theories, these postulates do not define *laws* of a particular physical system. They rather represent a mathematical framework that *allows* development of *valid* physical theories. We began this chapter with the comparison of mathematics and the Universe. If there is indeed some kind of mathematics intertwined in the fabric of the Universe, then, up to the current knowledge, this mathematics is the *quantum mechanics*.

Postulate 1 ([NC00]). Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space) known as the *state space*

Notation	Description
z^*	Complex conjugate of the complex number z.
$\|\psi\rangle$	Column vector called *ket*.
$\langle\psi\|$	Vector dual to $\|\psi\rangle$. Also known as *bra*.
$\langle\phi\|\psi\rangle$	Inner product *"bra-ket"*.
$\|\phi\rangle \otimes \|\psi\rangle$	Tensor product of $\|\phi\rangle$ and $\|\psi\rangle$
A^*	Complex conjugate of the matrix A.
A^T	Transpose of the matrix A.
$A^\dagger$	Hermitian conjugate or adjoint of the matrix A: $A^\dagger = (A^T)^*$.
$\langle\phi\|A\|\psi\rangle$	Inner products between $\|\phi\rangle$ and $A\|\psi\rangle$.
	Equivalently, inner product between $A\|\phi\rangle$ and $\|\psi\rangle$.

Table 4.1: Elements of linear algebra in quantum mechanics

of the system. The system is completely described by its *state vector*, which is a unit vector in the state space of the system.

The simplest quantum mechanical system is a *qubit*. It is described by a *complex* vector in two dimensional Hilbert space. Suppose $|0\rangle$ and $|1\rangle$ form an orthonormal basis, then the state of the qubit system ψ is described by the vector

$$|\psi\rangle = a|0\rangle + b|1\rangle.$$

By the *first postulate*, vector $|\psi\rangle$ must satisfy the *normalization condition* $\langle\psi|\psi\rangle = 1$. The latter is equivalent to $|a|^2 + |b|^2 = 1$, because $a, b \in \mathbb{C}$. It must be clear that the state space of the qubit is a two dimensional sphere (See **Figure 4.1**). We call the coefficient a the *amplitude* of the state $|0\rangle$, analogously, b is the *amplitude* of the state $|1\rangle$. In general, $\Sigma_{i=1}^{n} a_i|\psi_i\rangle$ is a *superposition* of the states $|\psi_i\rangle$ with *amplitudes* a_i.

How does the state change in time? It turns out that the Universe is reversible on the small scale.

Postulate 2 ([NC00]). The evolution of a *closed* quantum system is described by a *unitary* transformation. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\psi'\rangle$ at time t_2 by a unitary operator U which depends only on the times t_1

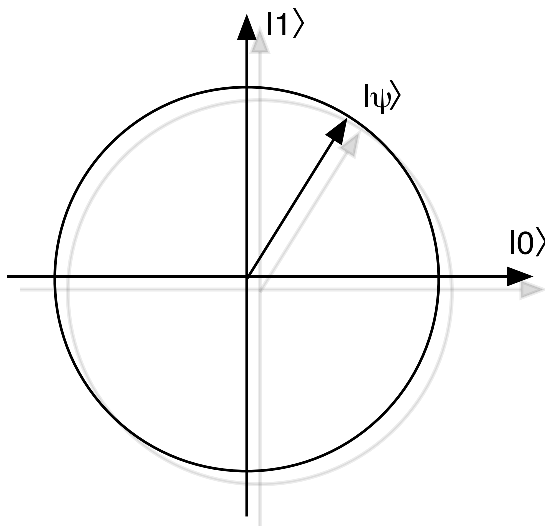

Figure 4.1: The state space of the qubit

and t_2:

$$|\psi'\rangle = U|\psi\rangle.$$

Recall that a matrix U is unitary if, and only if, $UU^\dagger = I$. That means that for any unitary transformation U there is an inverse operator $U^\dagger$. The reason that we don't deliberately travel back and forth in time is found in the thermodynamics. According to Hawking [Haw88], our individual arrow of time always points in the direction of the increasing entropy. The argumentation he gives is simple as it is. If we perform a computation (and human brain is a computer of some sort), then we start from a disordered state and finish in a state with a higher degree of order, that corresponds to the result of the computation. We need to spend some energy, that would be dissipated as heat, to achieve that. That's why the *entropy* of the Universe must increase after the computation is finished. That's why we don't time-travel at will.

Let us give an example of a unitary evolution of our two dimensional system called *qubit*.

Example 14 (The Pauli matrices). Suppose

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix};$$

then we can represent $|\psi\rangle = a|0\rangle + b|1\rangle$ as

$$|\psi\rangle = \begin{pmatrix} a \\ b \end{pmatrix};$$

define matricies

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \ Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}; \ Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix};$$

clearly X, Y and Z are unitary. For example, the action of X on $|\psi\rangle$ is a swap of the amplitudes of $|0\rangle$ and $|1\rangle$. The actions of the Pauli matrices on two-dimensional state vectors are shown below.

$$X|\psi\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a \end{pmatrix};$$

$$Y|\psi\rangle = \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} -ia \\ ib \end{pmatrix};$$

$$Z|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a \\ -b \end{pmatrix}.$$

The *second postulate* requires the system to be closed in order to be a subject of exclusively unitary evolutions. In reality, set aside the Universe as whole, most of the systems are not closed. That is, they interact with other systems to an extent. Nevertheless, it turns out that the unitary evolutions can very well approximate many of the interesting systems. Also there is a way to consider a system as a part of a larger system which is closed, and there always is such a system – the Universe.

The systems that evolve according to their unitary evolutions are of limited interest if they can never be observed. The observation is an intrusion in the closed system that reveals some information about it. Next postulate formalizes the *measurement.*

Postulate 3 ([NC00]). Quantum measurements are described by a collection $\{M_m\}$ of *measurement operators.* These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediatly before the measurement then the *probability* that result m occurs is given by

$$\Pr(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$$

and the state of the system after the measurement is

$$\frac{M|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}.$$

The measurement operators satisfy the *completeness equation,*

$$\Sigma_m M_m^\dagger M_m = I.$$

The completeness equation simply expresses the fact that all measurement outcome probabilities sum up to one. The following expression considered for arbitrary states $|\psi\rangle$ imply the *completeness equation.*

$$1 = \Sigma_m \Pr(m) = \Sigma_m \langle\psi|M_m^\dagger M_m|\psi\rangle.$$

In the next example we introduce a simple yet useful kind of measurement.

Example 15 (Measurement of a qubit in the computational basis). We define a two outcomes measurement on a single qubit. Let $|\psi\rangle = a|0\rangle + b|1\rangle$ be a qubit system in the basis $\{|0\rangle, |1\rangle\}$. We define measurement operators $M_0 = |0\rangle\langle 0|$ for the measurement outcome 0, and $M_1 = |1\rangle\langle 1|$ for the measurement outcome 1.

Observe that the measurement operators are *Hermitian*, that is, $M_i^\dagger = M_i$. Moreover, $M_i^2 = M_i$. Thus, the completeness relation is obeyed:

$$I = M_0^\dagger M_0 + M_1^\dagger M_1 = M_0 + M_1.$$

now consider the two possible outcomes of the measurement. The state $|\psi'\rangle$ is the state of the system after the measurement.

0:

$$\begin{aligned}\Pr(0) &= \langle\psi|M_0^\dagger M_0|\psi\rangle = \langle\psi|M_0|\psi\rangle = |a|^2 \\ |\psi'\rangle &= \frac{M_0|\psi\rangle}{|a|} = \frac{a}{|a|}|0\rangle \sim |0\rangle.\end{aligned}$$

1:

$$\begin{aligned}\Pr(1) &= \langle\psi|M_1^\dagger M_1|\psi\rangle = \langle\psi|M_1|\psi\rangle = |a|^2 \\ |\psi'\rangle &= \frac{M_1|\psi\rangle}{|a|} = \frac{a}{|a|}|1\rangle \sim |1\rangle.\end{aligned}$$

It is easy to see that the states $e^{i\theta}|\psi\rangle$ and $|\psi\rangle$ have the same measurement statistics. That's why we should further regard them as identical. That's why we ignore *global phase factors*.

We introduced the most general definition of measurements. Sometimes a simple special case of measurement operators is used.

Definition 4.2.1 ([NC00]). A *projective measurement* is described by an *observable* M, a Hermitian operator on the state space of the system being observed. The *observable* has the spectral decomposition

$$M = \Sigma_m m P_m,$$

where P_m is the projector onto the eigenspace of M with eigenvalue m. The possible outcomes of the measurement correspond to the eigenvalues, m, of the observable.

Upon measuring the state $|\psi\rangle$, the probability of getting result m is given by

$$\Pr(m) = \langle\psi|P_m|\psi\rangle.$$

Given that outcome m occurred, the state of the quantum system immediately after the measurement is

$$\frac{P_m|\psi\rangle}{\sqrt{\Pr(m)}}.$$

Sometimes even projective measurement are not convenient enough. If an orthonormal basis $|m\rangle$ is given, we can *measure in a basis* $|m\rangle$. That simply means a projective measurement with projectors $P_m := |m\rangle\langle m|$.
We know how to describe *evolution* of a quantum system. Then *measurement* defines how we obtain some information about the system. We mentioned, that some systems can be parts of larger systems. But, we haven't yet explained how *composite* systems are treated in quantum mechanics. The last postulate of the quantum mechanics defines the *composite* state space is related to its *components*.

Postulate 4 ([NC00]). The state space of a *composite* physical system is the *tensor product* of the state spaces of the *component* physical systems. Moreover, if we have systems numbered 1 through n, and system number i is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle$

Let us give a beautiful example of an application of the quantum mechanics.

Example 16 (Superdense coding). Suppose $|\psi\rangle = |\phi_1\rangle \otimes |\phi_1\rangle$ is a two-qubit quantum system prepared by a *third party* so that

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

The *third party* then distributes the qubits of $|\psi\rangle$ between *Alice* and *Bob* before the communication takes place. In other words, *Alice* possesses $|\phi_1\rangle$, while *Bob* owns $|\phi_2\rangle$. Thus, *Alice* and *Bob* share a pair of qubits in the *entangled* state $|\psi\rangle$. It turns out that *Alice* can communicate *two* bits of *classical* information by sending a single qubit $|\phi_1\rangle$. The setup is shown on **Figure 4.2**.

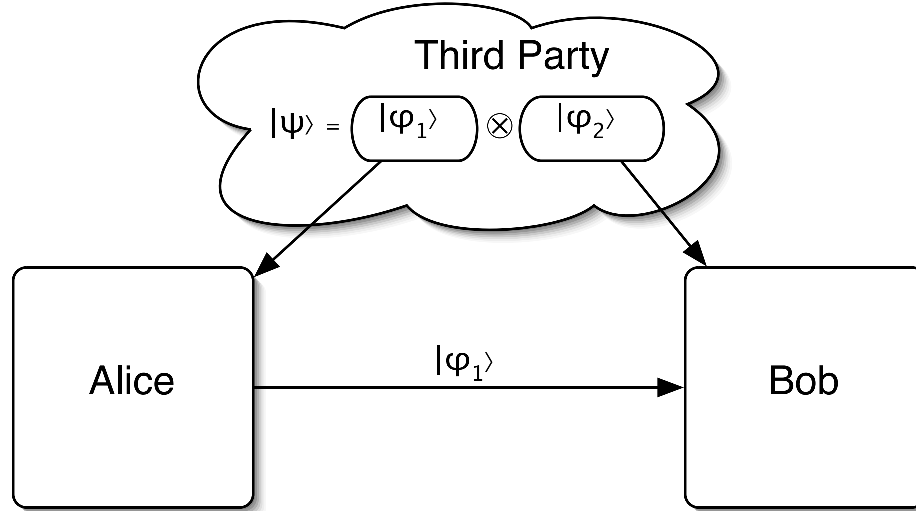

Figure 4.2: Communicating two classical bits by sending one qubit

Consider four possible two-bit combinations that *Alice* can send to *Bob*. The actions of *Alice* would be different in each of the case.

00: *Alice* does nothing to its qubit.

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\beta_{00}\rangle.$$

01: Then *Alice* applies the *NOT* operator to $|\phi_1\rangle$.

$$|\psi\rangle := (X|\phi_1\rangle) \otimes |\phi_2\rangle = \frac{|10\rangle + |01\rangle}{\sqrt{2}} = |\beta_{01}\rangle.$$

10: In this case *Alice* applies the *phase flip* Z operator to $|\phi_1\rangle$.

$$|\psi\rangle := (Z|\phi_1\rangle) \otimes |\phi_2\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} = |\beta_{10}\rangle.$$

11: Finally, if this is the message to send, *Alice* applies iY operator to $|\phi_1\rangle$.

$$|\psi\rangle := (iY|\phi_1\rangle) \otimes |\phi_2\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{11}\rangle.$$

Observe that $\{|\beta_{01}\rangle, |\beta_{10}\rangle, |\beta_{10}\rangle, |\beta_{11}\rangle\}$ is an orthonormal basis for the state space of ψ. This states basis is called *Bell basis*, and the states are called *Bell states* or *EPR pairs*. After performing described preparations, *Alice* sends her qubit ϕ_1 to *Bob*. It remains for *Bob* to measure the system ψ, he is in possession, in the *Bell basis*. As a result of the measurement *Bob* finds out which of the four possible two-bit combinations *Alice* meant to communicate.

In the example above we mentioned that the Bell states are also called *EPR pairs*. What is "EPR"? The abbreviation stands for the initials of Albert Einstein, Boris Podolsky and Nathan Rosen. In 1935 they published a paper called "Can quantum-mechanical description of physical reality be considered complete?" [EPR35]. The authors rightfully notice that two questions can be used to judge upon success of a theory:

1. Is the theory correct?

2. Is the theory complete?

To that date Einstein already spent ten years in futile attempts to prove the quantum mechanics wrong in a direct way. At this time he decided to address the second question in connection with the quantum mechanics. The group of authors considered a thought experiment, where a system was described by the state $|\beta_{11}\rangle$ from the **Example 16**. This state has an interesting feature: whenever the first qubit is measured in some basis, the value of the second qubit is also collapses to the other basis state. It is clearly so, for example, in the basis $\{|0\rangle, |1\rangle\}$. The team lead by Einstein considered a projective measurement where the only two possible outcomes corresponded to operators that *did not commute*. By the *generalized uncertainty principle,* we can measure with certainty only one of the two values. Just like it was for the *momentum* and the *position* of a particle. However, whatever different values

reveal the measurement of the first qubit, the second qubit ”knows” the outcome. Although, no interaction between the particles happens during or after the measurement. Thus, the authors argued, both values measured represent, according to the quantum mechanics, *essential elements* of reality. That, in turn, implies either

- both of the values *can* be measured with certainty, invalidating the general uncertainty principle of the quantum mechanics,
- or the quantum mechanics fails to represent with certainty some *essential elements* of reality, giving only probabilistic explanation instead.

In both cases, conclude the authors, the quantum mechanics is *incomplete!*
However, in order to construct such an experiment, Einstein and colleagues had to made certain assumptions on the nature of the physical reality. It appears that Nature does not like when we impose it some rules to obey. Nearly thirty years later it was experimentally proved that the Nature obeys rather the quantum mechanics than the common thought of Einstein, Podolsky and Rosen.
The Universe does not have to be reasonable, despite of the beliefs of the greatest explorers of its secrets. The quantum mechanics, like a phoenix, raised from the ashes every time it was thought to have passed away. Getting a better ground after each blow. Decades long scrutiny undertaken by Einstein did not unearth significant flows in the theory. The experimental results agree with it so far. It has even went to technology! All that does not proof it is correct. But we could hope, with high probability of success, that quantum mechanics is the theory sound enough to base our computational models upon. We present the main computational model considered in this work in the next section.

4.3 Quantum branching programs

Quantum branching programs were first introduced independently by Ablayev, Gainutdinova and Karpinski [AGK01] and by Nakanishi, Hamaguchi and Kashiwabara [NHK00]. The quantum branching program can be defined in a very similar way to

the *randomized branching programs*. This approach is taken by Sauerhoff and Sieling [SS04]. We only slightly modify their definition.

Definition 4.3.1. A *quantum branching program* over the input $x = \{x_1, \ldots, x_n\}$ is a directed acyclic multigraph $G = (V, E)$ with exactly one root (a node with indegree zero), such that

- the sinks are labeled by the Boolean constants 0, 1,
- the internal nodes are labeled by a variable x_i and have always two kinds of outgoing edges, 0-edges and 1-edges (labeled by 0 and 1 respectively),
- every edge (v, w, b) is labeled additionally by a *transition amplitude* $\delta(v, w, b)$, where v, w are the two nodes connected by the b-edge (b is the Boolean label of the edge). We assume there is at most one b-edge between a pair of nodes for every Boolean constant b.
- the *transition function* must satisfy the following conditions.
    1. For all $(v, e) \notin E$ $\delta(v, w, b) = 0$.
    2. The transition function must be *well formed*. That is, for all internal nodes v labeled by x_i, and w, labeled by x_j, and any assignment of the input variables x

$$\Sigma_{w \in V} \delta^*(u, w, x_i)\delta(v, w, x_j) = \begin{cases} 1, & \text{if } u = v; \\ 0, & \text{otherwise.} \end{cases}$$

An example of the quantum branching program is shown in **Figure 4.3**. The *well-formedness* condition allows us to define *computation* of the branching program in terms of *quantum mechanics*.

Definition 4.3.2. Let $G = (V, E)$ be a quantum branching program over the input $x = \{x_1, \ldots, x_n\}$. Let $s \in V$ be the *root* node. We use $F \subseteq V$ to denote the set of the *sink* nodes. For any fixed assignment of x we define the *computation* of G on x as follows.

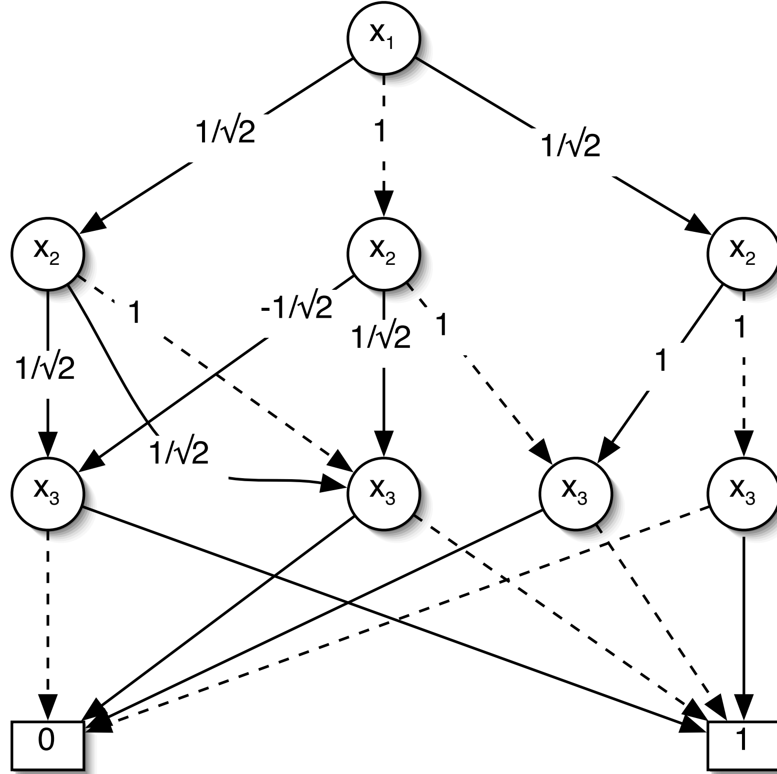

Figure 4.3: Quantum branching program

- The *state space* of $\mathcal{P}$ is $\mathcal{H}^{|V|}$. We identify the nodes of G with the basis vectors of $\mathcal{H}^{|V|}$.

- Let $L(x)$ be a linear operator $L(x) : V \setminus F \to V$, such that for all $|v\rangle \in V \setminus F$

$$L(x)|v\rangle = \Sigma_{w \in V} \delta(v, w, x_i)|w\rangle,$$

where x_i is the label of v. The *well-formedness condition* implies that $L(x)$ can be extended to a unitary operator $U(x)$ on $\mathcal{H}^{|V|}$.

- Define projection operators

$$E_{cont} = \Sigma_{v \in V \setminus F} |v\rangle\langle v|;$$
$$E_r = |r\text{-sink}\rangle\langle r\text{-sink}|.$$

- For $t \in \mathbb{N}_0$ and $r \in \{0,1\}$ we define the *probability* that G outputs r on the input x in t steps:

$$\Pr_{G,r}(x,t) = \Sigma_{k=0}^{t} \left|\left|E_r(U(x)E_{cont})^t|s\rangle\right|\right|^2$$

  and

$$\Pr_{G,r}(x) := \Pr_{G,r}(x,\infty).$$

- Then we can defined the *running time*, or the *length* of the quantum branching program on the input x.

$$T_G(x) = \min\{t|\ t \in \mathbb{N}_0 \cup \{\infty\}, \Pr_{G,0}(x,t) + \Pr_{G,1}(x,t) = 1\}.$$

  Notice, that the running time can be finite, infinite or *undefined*.

Unfortunately, there hasn't been a powerful lower bounds method invented for the unrestricted branching program model, defined above. In this thesis we consider, perhaps, the most "popular" restricted class of branching programs - OBDD (See p. 30). The definition for the quantum case is absolutely identical to the classical **Definition 2.3.8**.

The quantum OBDD were first introduced and studied by Ablayev, Gainutdinova and Karpinski [AGK01]. They called it 1QBP for *one way quantum branching program.* The definition chosen by these authors for the unrestricted quantum branching program is a *special case* of the **Definition 4.3.1**. Historically prior to it, the definition

of Ablayev, Gainutdinova and Karpinski also offers certain economy in space complexity. The *state space* is now related to a *level* of the branching program. Thus, the *width* of the branching program corresponds to the *space complexity* and the *length* – to the *time complexity*. In the definition of Sauerhoff and Sieling, both *length* and *width* defined the *space complexity* and the *time complexity* was defined separately. However, both approaches lead to equivalent OBDD models up to at most factor $(n+1)^2$ increase in size, where n is the length of the input. But we strive to come up with linear complexity bounds in this research. Therefore, we take the approach of Ablayev, Gainutdinova and Karpinski.

Definition 4.3.3 ([AGK01]). A Quantum Branching Program $\mathcal{P}$ of width d and length l (a $(d,l)-QBP$) based on a quantum system in $\mathcal{H}^d$ is defined as a triple.

$$\mathcal{P} = \langle T, |\psi_0\rangle, F\rangle,$$

where T is a sequence (of length l) of d-dimensional quantum transformations on $\mathcal{H}^d$:

$$T = (j_i, U_i(0), U_i(1))_{i=1}^{l}$$

The initial configuration of P is $|\psi_0\rangle$. $F \subseteq \{1, \ldots, d\}$ is the set of accepting states. We define a computation on $\mathcal{P}$ for an input $\sigma = \sigma_1, \ldots, \sigma_n \in \{0,1\}^n$ as follows:

1. A computation of $\mathcal{P}$ starts in the state $|\psi_0\rangle$. On the i-th step, $1 \leq i \leq l$, of the computation, $\mathcal{P}$ performs a transformation $|\psi\rangle \longrightarrow U_i\left(\sigma_{j_i}\right)|\psi\rangle$

2. After the l-th (last) step of the transformations $\mathcal{P}$ measures its configuration $|\psi_\sigma\rangle = U_l\left(\sigma_{j_l}\right) U_{l-1}\left(\sigma_{j_{l-1}}\right) \ldots U_1\left(\sigma_{j_1}\right)|\psi_0\rangle$. The measurement is represented by a diagonal zero-one projection matrix M, where $M_{ii} = 1$ if $i \in F$ and $M_{ii} = 0$ if $i \notin F$. The probability $\Pr_{accept}(\sigma)$ of $\mathcal{P}$ accepting the input σ is defined by the following equation.

$$\Pr_{accept}(\sigma) = ||M|\psi_\sigma\rangle||^2.$$

The program $\mathcal{P}$ from the definition above can be considered as a special case of *linear*

programs. That is, sequences of linear transformation in a vector space. The approach is illustrated in **Figure 4.4**.

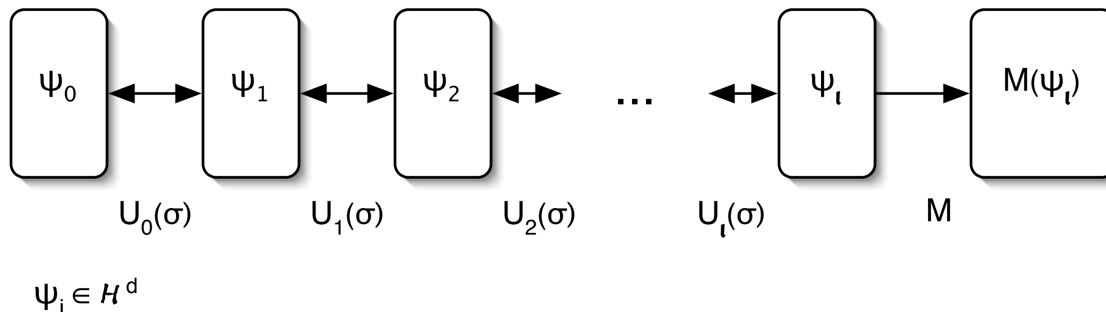

Figure 4.4: Quantum linear program

Definition 4.3.4. We say a function f is ϵ-*accepted* by a quantum branching program $\mathcal{P}$ if its error probability is at most $1/2-\epsilon$ and its correct answer probability is at least $1/2+\epsilon$. We can define acceptance *with one-sided error* analogously to the *randomized* case (See **Def. 3.5.3**, p. 59).

Observe, that the quantum branching program according to **Definition 4.3.3** is naturally *oblivious* (See **Def. 2.3.6**, p. 29). This justifies the use of the term *Read-Once Quantum Branching Program*. Indeed, oblivious read-once branching program is an OBDD, according to the **Definition 2.3.8** from the page 30. We give the following definition.

Definition 4.3.5. We call an *oblivious* Quantum Branching Program $\mathcal{P}$ an *(oblivious) Read-Once Quantum Branching Program* (Quantum Ordered Binary Decision Diagram) if each variable $x \in \{x_1, \ldots, x_n\}$ occurs in the sequence T of the quantum transformation of the program $\mathcal{P}$ at most once.

We shall use the notation 1QBP for the class of all *Read-Once Quantum Branching Program* (Quantum Ordered Binary Decision Diagram). We usually omit "oblivious" when it does not cause confusion, since the model we consider is *naturally* oblivious (See **Definition 4.3.3**).

Let us mention that we can describe oblivious *randomized branching programs* as linear programs of stochastic linear operators, analogously to the **Definition 4.3.3**. Like it was the case with quantum branching programs, we do not arrive at some new kind of computational model. At least, unless we require probabilities specified up to asymptotically more than $\log T$ bits of precision, where T is the size of the program. However, $\log T$ bits of precision is also a natural restriction for quantum amplitudes. It is natural to assume, that, during the computation, we can generate probabilities (or amplitudes) up to 1 part in 2^T where T is the size of the branching program. Better precision is not obviously achievable. The early discussion for the quantum setting can be found in the work of Bernstein and Vazirani [BV97].

It is not difficult to see that the new definition is simply a special case of the more general **Definition 3.5.3** on the page 59, written down in the language of linear algebra.

Definition 4.3.6. An *oblivious* Randomized Branching Program $\mathcal{P}$ of width d and length l (a $(d, l) - RBP$) can be defined as a triple.

$$\mathcal{P} = \langle T, s_0, F \rangle,$$

where T is a sequence (of length l) of $d \times d$ stochastic matrices:

$$T = (j_i, L_i(0), L_i(1))_{i=1}^{l}$$

The initial configuration of $\mathcal{P}$ is s_0. The configurations are described by d-dimensional stochastic vectors. $F \subseteq \{1, \ldots, d\}$ is the set of accepting states.

We define a computation on $\mathcal{P}$ for an input $\sigma = \sigma_1, \ldots, \sigma_n \in \{0, 1\}^n$ as follows:

1. A computation of $\mathcal{P}$ starts with s_0. On the i-th step, $1 \leq i \leq l$, of the computation $\mathcal{P}$ performs a transformation $s \longrightarrow L_i \left(\sigma_{j_i}\right) s$

2. After the l-th (last) step of the transformations, $\mathcal{P}$ stops in the configuration $s_\sigma = L_l \left(\sigma_{j_l}\right) L_{l-1} \left(\sigma_{j_{l-1}}\right) \ldots L_1 \left(\sigma_{j_1}\right) s_0$. The output probability is described by a diagonal zero-one projection matrix M, where $M_{ii} = 1$ if $i \in F$ and $M_{ii} = 0$

if $i \notin F$. The probability $\Pr_{accept}(\sigma)$ of $\mathcal{P}$ accepting input σ is defined by the following equation.

$$\Pr_{accept}(\sigma) = |Ms_\sigma|.$$

Despite of the apparent similarities, the two linear program definitions given above underline very important differences of the *randomized* and *quantum* models:

1. The unitary transformations in *quantum* branching programs are replaced by the stochastic matrices in the *randomized* setting.
2. The "nonlinear" l_2 norm of the acceptance probability in *quantum* case is replaced by "linear" l_1 norm in the *randomized* computation.

Note also, that the *deterministic* branching programs are a special case of the randomized linear programs defined above. There, all stochastic matrices are restricted to the *zero-one* matrices, and all *configuration* vectors are also *zero-one*.
Thus, we arrive at a very interesting mixture of similarities and differences. It will have rather surprising consequences in the next section, where we introduce several interesting complexity classes inspired by the quantum branching programs, and show curious relationships of these classes.

4.4 Quantum branching programs complexity

We introduced several complexity measures on quantum branching programs in the previous section. The *size* being the most influential of them. One of the most interesting results about quantum branching programs complexity is due to Ablayev, Moore and Polette [AMP02]. They showed that quantum branching programs of *constant width* 2 are as powerful as the *circuits of polynomial size and logarithmic depth*! Let us define the complexity classes we shall later relate to each other.

Definition 4.4.1 ([Aar]). Class $\mathbf{NC^i}$ is the class of decision problems solvable by a uniform family of Boolean circuits, with polynomial size, depth $O(log^i(n))$, and

fan-in 2. Then **NC** is the union of $\mathbf{NC^i}$ over all nonnegative i. NC stands for "Nick's Class".

The *Boolean circuit* is a circuit of usual logical *gates* (AND, OR, NOT) that connected using wires. *Fan-in* is the maximal number of input wires that a gate can have. In the definition above we mention a notion of b*uniform family.*

Definition 4.4.2. A computational model is called *uniform* if a single algorithm is used for inputs of *different length.* A *family* of algorithms is called *uniform* if, for every input length, appropriate algorithm can be generated within polynomial in the input length *resources.*

We shall mostly deal with uniform families when consider nonuniform models, such as branching programs. We define next the bounded width branching program complexity classes.

Definition 4.4.3.

1. $k-$**P-BP** is the class of functions represented by polynomial-length branching programs of width k. We denote **BWP-BP** $= \cup_k k -$ **P-BP**.
2. $k -$ **BPP-BP** is a subclass of **BPP-BP** that consists of all functions that can be represented by a polynomial size nondeterministic branching program of polynomial size and width k. We denote **BWBPP-BP** $= \cup_k k -$ **BPP-BP**.
3. **BQP-BP** is a class of all functions that ϵ-accepted by some *polynomial-size* quantum branching program, for some constant $\epsilon \in (0, 1/2)$.
4. $k -$ **BQP-BP** is a subclass of **BQP-BP** that consists of all functions that ϵ-accepted by some *polynomial-size* quantum branching program of width k, for some constant $\epsilon \in (0, 1/2)$. We denote **BWBQP-BP** $= \cup_k k -$ **BQP-BP**.
5. **EQP-BP** is a class of all functions that computed by some *polynomial-size* quantum branching program *exactly*, that is with *zero error.*
6. $k -$ **EQP-BP** is a subclass of **EQP-BP** that consists of all functions that computed by a polynomial size quantum branching program of width k exactly. We denote **BWEQP-BP** $= \cup_k k -$ **EQP-BP**.

In 1988 Barrington [Bar85] showed that $5-$ **P-BP** $=$ $\mathbf{NC^1}$. Moreover, for deterministic branching programs, width 5 is necessary, unless $\mathbf{NC^1} = \mathbf{ACC^0}$ [BT88].

Definition 4.4.4 ([Aar]). Class $\mathbf{ACC^0}$ is the class of decision problems solvable by a uniform family of Boolean circuits with additional MOD m gates for any m, with polynomial size, constant depth, unbounded fan-in.

It was not before the turn of the century that Ablayev, Moore and Pollette showed that $2-$**EQP-BP** $=$ $\mathbf{NC^1}$. This is still one of the most remarkable results concerning quantum branching programs known so far. It is also interesting to notice that all the plethora of classes that we defined above, actually, collapses!

Theorem 12 ([AMP02]).

$$2-\mathbf{EQP\text{-}BP} = 2-\mathbf{BQP\text{-}BP} = \mathbf{EQP\text{-}BP} = \mathbf{BQP\text{-}BP}$$
$$= \mathbf{BPP\text{-}BP} = \mathbf{BWP\text{-}BP} = \mathbf{NC^1}.$$

Unrestricted branching programs are generally unexplored due to the lack of powerful proof methods. The bounded width restriction brought up some non-trivial insights into the power of quantum branching programs. However it did not led to an interesting structure of complexity classes, that simply collapsed... What is the situation with OBDD?
It turns out, for OBDD the situation is quite different! Let us first introduce some interesting OBDD complexity classes that we shall deal with.

Definition 4.4.5.

1. **BQP-OBDD** is the class of all functions ϵ-accepted by some *polynomial-size* quantum OBDD, for some constant $\epsilon \in (0, 1/2)$.

2. **EQP-OBDD** is the class of all functions computed by some *polynomial-size* quantum OBDD *exactly*, that is with

3. **RQP-OBDD** is the class of all functions ϵ-accepted *with one-sided error* by some *polynomial-size* quantum OBDD, for some constant $\epsilon \in (0, 1/2)$.

4. The class of *zero error* quantum OBDD is defined as

$$\mathbf{ZQP\text{-}OBDD} = \mathbf{RQP\text{-}OBDD} \cap \mathbf{coRQP\text{-}OBDD}.$$

5. The class of all *reversible* programs from **P-OBDD** is called **Rev-OBDD**.

The last *deterministic* class defined above is a natural deterministic subclass of **BQP-OBDD**, just like **P-OBDD** is a subclass of **BPP-OBDD**. We mentioned the second inclusion in the end of the previous section.
Let us start from something similar to the *bounded-width branching programs* complexity classes hierarchy collapse. Next result is due to Sauerhoff and Sieling [SS04].

Theorem 13 ([SS04]). Rev-OBDD=EQP-OBDD=ZQP-OBDD.

However, the rest of the complexity classes do not simply collapse for quantum OBDD. In 2001 Ablayev, Gainutdinova and Karpinski [AGK01] presented an explicit function that proved an exponential gap between *stable randomized* and *stable quantum* OBDD. *Stability* is a rather strong restriction on OBDD.

Definition 4.4.6. Consider an OBDD $\mathcal{P}$, where $T = (j_i, M_i(0), M_i(1))_{i=1}^n$ is the appropriate sequence of transformations, and n is the length of the input. Then $\mathcal{P}$ is called *stable* if $M_i(0) = M_j(0)$ and $M_i(1) = M_j(1)$ for all $i, j \in \{1, \ldots, n\}$. In other words transformation do not depend on the level of $\mathcal{P}$.

Definition 4.4.7 ([AGK01]). MOD_p: On input $\sigma = \sigma_1, \ldots, \sigma_n \in \{0,1\}^n$ we have $\mathrm{MOD}_p(\sigma) = 1$ if and only if the number of ones in σ is divisible by p.

Theorem 14 ([AGK01]). *The function* MOD_p *can be presented by a* stable*, read-once, width-*$O(\log p)$ *quantum branching program with one-sided error* $\epsilon > 0$.
Any stable *probabilistic OBDD computing* MOD_p *has width at least* p.

With this result a hope was born, that *quantum computers* can be rigorously proved to be more powerful than the *randomized computers* in the OBDD setting. The latter,

such as those used to define various BPP complexity classes, are widely accepted to be a practical model of computation. Alas! In 2004 Sauerhoff and Sieling showed that *quantum* and *classical* OBDD are incomparable [SS04]! They used two following functions as witnesses.

Definition 4.4.8. The *permutation matrix test* function PERM_n is defined on $n \times n$ Boolean matrix that is arranged in a string. For an input $\sigma \in \{0,1\}^{n^2}$ $\mathrm{PERM}_n(\sigma) = 1$ if, and only if, σ corresponds to a permutation matrix. That is, a zero-one matrix where each row and column contains exactly one entry 1.

It is well known that unrestricted *nondeterministic* read-once branching programs for PERM_n have exponential size (See [KMW88, Juk89]). However, it has also been known that this function is not hard for *randomized* OBDDs [Sau97, Weg00].

Theorem 15 ([Sau97]). *The function* PERM_n *can be computed with a one-sided* $\epsilon(n)$*-error by a randomized read-once ordered branching program of size*

$$O\left(\epsilon(n)^{-2} n^5 \log^3 n\right).$$

Additionally Sauerhoff and Sieling define a beautifully simple *neighboring ones* function.

Definition 4.4.9 ([SS04]). The *neighboring ones* function NO_n is defined on the Boolean variables $x_1, \ldots, x_n$. It takes the value 1 if, and only if, there are two neighbored variables with value 1. That is, if there is an integer $i \in \{1, \ldots, n-1\}$, such that $x_i = x_{i+1} = 1$.

It is obvious how to construct a deterministic OBDD of size $O(n)$ that would compute NO_n. Now the separation of *deterministic* and *quantum* OBDD is shown by the following results.

Theorem 16 ([SS04]). *There are* quantum *OBDDs for* $\neg\mathrm{PERM}_n$ *with one sided error* $1/n$ *and size* $O(n^6 \log n)$*. Thus, it follows*

$$\textbf{BQP-OBDD} \not\subseteq \textbf{P-OBDD}.$$

Theorem 17 ([SS04]). *The size of each* quantum *OBDD G with bounded error for* NO_n *is at least* $2^{\Omega(n)}$. *Thus, it follows*

$$\mathbf{P\text{-}OBDD} \not\subseteq \mathbf{BQP\text{-}OBDD}.$$

Some of the known class relations are shown in the **Figure 4.5**. Note that, although

$$\mathbf{BPP\text{-}OBDD} \not\subseteq \mathbf{BQP\text{-}OBDD},$$

it remains an open question whether

$$\mathbf{BQP\text{-}OBDD} \subseteq \mathbf{BPP\text{-}OBDD}.$$

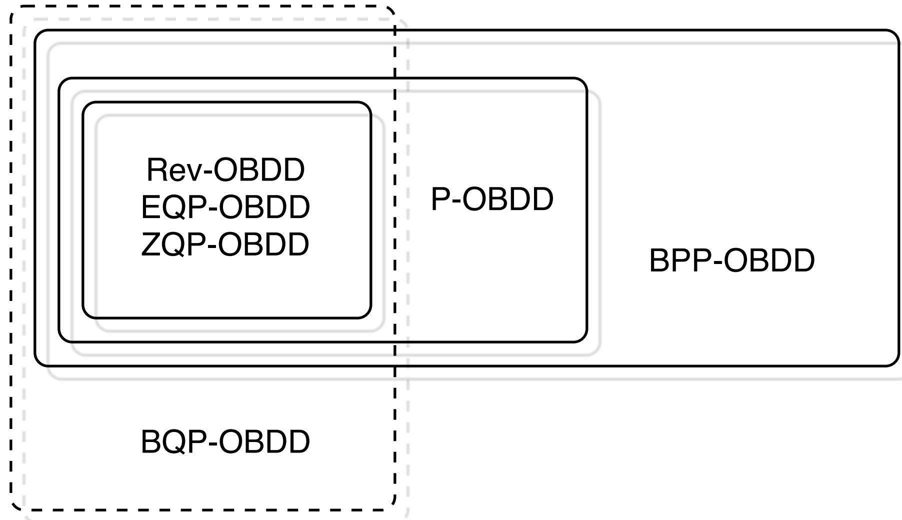

Figure 4.5: OBDD complexity classes hierarchy

So far we have considered quantum OBDD model, which is the same as read-once *oblivious* quantum branching programs. However, general **Definition 4.3.2** allows also construction of read-once quantum *non-oblivious* branching programs. Does this opportunity promise any advantage in terms of computational power? It proves it does, as it was noticed by Sauerhoff and Sieling [SS04]. We follow their presentation.

Definition 4.4.10. Let $n = 2^k$, for the input consisting of $y = \{y_0, \dots, y_{k-1}\}$ and $x = \{x_0, \dots, x_{n-1}\}$ we define the *indirect storage access* function $\mathrm{ISA}_n(x, y)$ in the following way. The string y is interpreted as a binary number $s(y)$. If $s(y) \leq b := \lfloor n/k \rfloor$ then the k-bit sequence $x_{ks}, \dots, x_{ks+k-1}$ is also interpreted as a binary number $x^s \leq n-1$ and the output is $x_{x^s(y)}$. Otherwise, if $s(y) > b$ the output is 0.

The definition of the *indirect storage access* function is practically a definition of a *decision tree* of size $O(2^k \cdot b) = O(n^2/\log_2 n)$. This decision tree can also be considered as a read-once *quantum branching program*, since it represents a *reversible* process. However, any quantum ZQP-OBDD that computes ISA_n has to be of size $2^{\Omega(n/\log_2 n)}$ [SS04].
Recently Sauerfhoff showed the separation also for *non-oblivious* branching programs [Sau05]. There, Sauerhoff used the well-known function *set disjointness* and a function he constructed for the purpose.

Definition 4.4.11. Let $x = (y_1, \dots, y_n),\ y = (y_1, \dots, y_n) \in \{0,1\}^n$. We define *set disjointness* function

$$\mathrm{DISJ}_n(x, y) = \neg(x_1 y_1 \vee \cdots \vee x_n y_n).$$

Definition 4.4.12 ([Sau05]). For a positive integer n and $x = (x_1, \dots, x_n) \in \{0,1\}^n$, let $p(n)$ be the smallest prime larger than n and let $s_n(x) = (\Sigma_{i=1}^n i \cdot x_i) \mod p(n)$. Define the *weighted sum* function by $\mathrm{WS}_n(x) = x_{s_n(x)}$ if $s_n(x) \in \{1, \dots, n\}$ and 0 otherwise.
For a further input $y = (y_1, \dots, y_n) \in \{0,1\}^n$ define the *mixed weighted sum* function by $\mathrm{MWS}_n(x, y) = x_i \oplus y_i$ if $i = s_n(x) = s_n(y) \in \{1, \dots, n\}$ and 0 otherwise.

The separation results are presented in the next two statements.

Theorem 18 ([Sau05]). *Each* randomized *read-once branching program computing* MWS_n *with two-sided error bounded by an arbitrary constant smaller than* $1/2$ *equires size* $2^{\Omega(n)}$ *while* MWS_n *can be computed by an error free* quantum *read-once branching program of size* $O(n^3)$.

Theorem 19 ([Sau05]). *Each quantum read-once branching program computing* DISJ_n *with two-sided error bounded by a constant smaller than* $1/2 - 2\sqrt{3}/7$ (≈ 0.005) *has size* $2^{\Omega(n)}$

The set disjointness is trivially computed by deterministic OBDD with an appropriate ordering.
The several incomparability results presented in this section unfortunately destroy the hopes to prove quantum computations superiority in the OBDD setting. The best we can hope for is to come up with good algorithms for particular important problems, or to show the best possible complexity for those problem. We take this approach in the next chapters.

4.5 Occam's Razor and quantum computers

In this chapter we introduced the quantum computers, the model of quantum branching programs, that would be of central interest in the next chapters. But before we proceed, it's good to say a pair of words to justify our interest in *quantum computational models.* We arrange them in the list below.

- First of all the quantum version of the Church-Turing thesis is a *provable conjecture* that depends on the validity of the given physical theory – of *quantum mechanics.* As we mentioned in the first chapter, this is a better situation than building up a theory based on the pure intuition only very indirectly related to the physical reality.

- Now physicists learn to think computationally. It is not only that any *computational process* is a kind of a *physical process.* It also works all the way around. Namely, an arbitrary physical process can be considered as a process of computation. The *Universe* is the largest Turing machine available, and it is doing a computation. Only we don't know what it is computing exactly.

- Quantum computing gave raise to the *quantum cryptography.* Those cipher machines produce unbreakable cryptographic protocols. The situation of *perfect*

> *security* was first considered by Claud Shannon [Sha49]. But prior to quantum cryptography protocols were insured to be of perfect security only if they were *symmetric block-ciphers* with the *key* size equal to the size of the cipher text. They also had to be *pad* ciphers, in other words they had to use different random keys for encoding each new message. All those demands were impractical as well as the assumption that the eavesdropper could use unrestricted computational power, for which the very notion of perfect security was necessary. Now there's no need of unrealistic assumptions of the malevolent party. The "perfect security" of quantum cryptographic protocols doesn't need those assumptions for justifying ridiculous resource demands. For a single reason – there are no those demands!

That's why we consider it worthwhile to spend our time and energy studying quantum computations. The engineering difficulties that yet prevent construction of full-scale quantum computers encourage us in our research. Here we consider computational models that utilize very restricted resources. They are easy to implement. However, they are powerful enough to solve some very important and hard computational problems. This topic is uncovered in the next chapters.

Chapter 5

Connections to the Hidden Subgroup Problem

> All animals are equal but some animals are more equal than others.
>
> ---
>
> George Orwell,
> *"Animal Farm"*

5.1 Introductions

One of the problems we consider in this chapter is *Equality*. It is one of the most fundamental problems in the computer science. Therefore, it is not surprising it has been extensively studied. One of the recent results somewhat related to our own is due to Høyer and de Wolf [HW02]. They prove that *quantum communication complexity* of the *Equality* is $n+1$ for *exact*, *one-sided error* and *non-deterministic* protocols, where n is the input length. The authors also gave a short review of the subject as well as they stated some open problems. The explicit connection between quantum *communication complexity* and quantum *branching programs* was established by Ablayev [Abl05]. The second result, combined with the first can provide alternative prove for the lower bounds presented in this chapter. However, it does not provide us with the

explicit quantum OBDD for *Equality*. The latter is the crucial construction element of our upper bound algorithms.
We define several boolean functions that we investigate in the first part of this work.

Definition 5.1.1. Let us define the *Equality* function $\mathrm{EQ}_n(x, y)$.

$$\mathrm{EQ}_n(x, y) \equiv [x = y]$$

Definition 5.1.2. For a binary sequence $\sigma \in \{0,1\}^n$, s – the period parameter, we define the *Periodicity* function $\mathrm{Period}_{s,n}(\sigma)$.

$$\mathrm{Period}_{s,n}(\sigma) = \begin{cases} 1 & \text{if } \sigma_i = \sigma_{i+s \mod n}, i = \overline{0, n-1}; \\ 0 & \text{otherwise.} \end{cases}$$

Before we tackle the real Simon problem, let's define a function that is similar but somewhat simpler. This function is intended to exhibit similarities and differences of finding period and solving Simon problem.

Definition 5.1.3. For $n = 2^l, l \in \mathbb{N}$ $s \in \{0, \dots n-1\}$ and for $\sigma \in \{0,1\}^n$, a binary sequence, we define the *Semi-Simon* function $\text{Semi-Simon}_{s,n}(\sigma)$.

$$\text{Semi-Simon}_{s,n}(\sigma) = \begin{cases} 1 & \sigma_i = \sigma_{i \oplus s}, i = \overline{0, n-1}; \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\oplus$ is a bitwise addition modulo 2.

We present a weak upper bound for *Equality* first. Its aim is to present one more proof technique as well as smoothly give an intuition for *Quantum Branching Programs.*

Theorem (Weaker Equality Upper Bound). *The function $EQ_n(x, y)$ can be computed with constant error $\delta < \frac{1}{2}$ by a 1QBP of width $O(n \log n)$ for n large enough.*

Eventually, we obtain linear upper bounds for all three Boolean functions. First, for the *Equality.*

Theorem (Stronger Equality Upper Bound). *The function $EQ_n(x, y)$ can be computed with one-sided error $o(1) \leq \frac{1}{7}, (n \to \infty)$ by a 1QBP of width $O(n)$, where $n = |xy|$ is the length of the input.*

Similar result holds for the *Periodicity*. It establishes linear, over the parameter size, upper bound. The program size is obviously constant over the input sequence size.

Theorem. *For all $s \in (0, n]$ and $\forall \sigma \in \{0, 1\}^n$ the function $Period_{s,n}(\sigma)$ can be computed with one-sided error $o(1) < 0.4, (s \to \infty)$ by a 1QBP of width $O(s)$.*

The theorem for the *Semi-Simon* problem is a corollary of the result for the *Equality* function.

Theorem. *For all $s \in (0, n]$ and $\forall \sigma \in \{0, 1\}^n$ the function Semi-Simon$_{s,n}(\sigma)$ can be computed with one-sided error $o(1) < 1/7, (s \to \infty)$ by a 1QBP of width $O(n)$.*

We conclude this chapter by proving linear lower bound on of the considered problems. Thus, we obtain asymptotic characterization of their quantum OBDD complexity.

5.2 Missing an important function

This section is dedicated to the function f_n defined by F. Ablayev, M. Karpinski [AK96].

Definition 5.2.1. Consider the finite alphabet Σ. For $\sigma_1, \sigma_2 \in \Sigma, x \in \Sigma^*$ define $\text{Proj}_{\sigma_1,\sigma_2}(x)$ to be a subsequence x' of the sequence x that consists only of the symbols σ_1 and σ_2.
Define funnction $f_n : \sigma^{2n} \to \{0, 1\}$ as follows. Function $f_n(x) = 1$ if, and only if two conditions are satisfied.

1. $\text{Proj}_{0,1}(x)$ and $\text{Proj}_{\hat{0},\hat{1}}(x)$ have the same length;

2. For all indicies i the i-th symbol in $\text{Proj}_{0,1}(x)$ is σ_i if, and only if the i-th symbol in $\text{Proj}_{\hat{0},\hat{1}}(x)$ is $\hat{\sigma}_i$.

Informally f_n essentially is an equality function for two binary words. First word written using $\{0, 1\}$, and second word – using $\{\hat{0}, \hat{1}\}$. Having the input encoded in binary (say $0 \to 00$, $1 \to 01$, $\hat{0} \to 10$, $\hat{1} \to 11$), function f_n can be easily defined as a boolean function over $\mathbb{B}^{4n}$.

$$f'_n : \{0, 1\}^{4n} \to \{0, 1\}. \tag{5.1}$$

Once this function made a contribution to *Structural Complexity*. It was f_n and $PERM$ defined in [Sau97] that were used to demonstrate classes AC_0 and $BPP-OBDD$ are incomparable [AK97].

The function f_n is a generalization of the function $\text{EQ}_n(x, y)$ that we consider in the next section. In fact, only slightly modified proof for *Equality* would lead to the theorem below.

Recall that a 1QBP Q is called *stable* if transformations applied on each level do not depend on the position of the corresponding argument but depend only on its value (See **Def. 4.4.6**, p. 87).

Theorem 20. *The function f_n can be computed with one-sided error $o(1) \leq \frac{1}{7}, (n \to \infty)$ by a stable 1QBP of width $O(n)$.*

Notice that the Read Once Branching Program, computing f_n in this case is *stable*! A lower bound is also valid for f_n. Similarly to the lower bound for *Equality*, the linear lower bound on the width of quantum OBDD for f_n follows from the exponential deterministic read-once branching program lower bound. This lower bound was first shown in the same paper where the the function was initially introduced [AK96].

Theorem 21 ([AK96]). *Any deterministic read-once branching program that computes the function f_n has the size of no less than 2^n.*

We have already mentioned that f_n has some history. Apart from the lower bound shown above, a polynomial upper bound on *randomized* branching program complexity was shown for f_n.

Theorem 22 ([AK96]). *Function f_n can be computed with one-sided error $\epsilon(n)$ by a*

randomized read-once ordered branching program P so that following holds. $size(P) \in O\left(\frac{n^6}{\epsilon^3(n)} \log^2 \frac{n}{\epsilon(n)}\right)$.

Next statement was originally given as a corollary in [AK97] for function f'_n. Naturally, it remains true for f_n.

Theorem 23. *Function f_n can not be computed by a nondeterministic ordered read-k-times branching program in polynomial size for $k = o(n/\log n)$.*

In other words, quantum setting offers significant polynomial advantage over corresponding known probabilistic algorithms. The complexity gap between the deterministic and the quantum OBDD is super-polynomial for f_n.
Other than what was mentioned, we don't prove statements for f_n. Despite of the function being a beautiful generalization encapsulating three other problems we consider, it lacks elegancy of presentation when several different problems are considered. Thus, in sake of clarity from now on we deal only with Boolean functions (images of $\{0,1\}^n \to \{0,1\}$, for some integer n). Nevertheless, we should keep in mind that f_n is an important function we also have non-trivial results about.

5.3 The upper bound for the equality function

First we present the weaker upper bound.

Theorem 24 (The Weaker Upper Bound). *The function $EQ_n(x,y)$ can be computed with constant error $\delta < \frac{1}{2}$ by a 1QBP of width $O(n \log n)$ for n large enough.*

Proof. To prove the statement, we shall build a 1QBP that computes $\mathrm{EQ}_n\ (x,y)$with requested properties.

Definition 5.3.1.

$$\mathrm{EQ}_{n,p}(x,y) \equiv [x \equiv y \mod p]$$

Lemma 2. *$EQ_{n,p}(x,y)$ can be computed by a 1QBP of width 2 with one-sided error*

$$\Theta\left(1 - \frac{1}{p}\right). \tag{5.2}$$

To prove this lemma, we conceive a one qubit (acting in 2-dimensional Hilbert space) 1QBP $\mathcal{A}_p = \langle T, |\psi_0\rangle, F\rangle$. Let $|0\rangle$ and $|1\rangle$ be two orthonormal fixed states to form a basis of the $\mathcal{H}^2$. Now we construct $\mathcal{A}_p$:

1. The *QOBDD* receives input $\sigma = xy$;
2. $|\psi_0\rangle = |0\rangle$;
3. $F = \{|0\rangle\}$;
4. (a) On the x-part of the input σ:
   $U_i(\sigma_i) = \begin{pmatrix} \cos\frac{\pi 2^{i+1}\sigma_i}{p} & \sin\frac{\pi 2^{i+1}\sigma_i}{p} \\ -\sin\frac{\pi 2^{i+1}\sigma_i}{p} & \cos\frac{\pi 2^{i+1}\sigma_i}{p} \end{pmatrix}$;

   (b) On the y-part of the input σ:
   $U_i(\sigma_i) = \begin{pmatrix} \cos\frac{\pi 2^{i+1}\sigma_i}{p} & -\sin\frac{\pi 2^{i+1}\sigma_i}{p} \\ \sin\frac{\pi 2^{i+1}\sigma_i}{p} & \cos\frac{\pi 2^{i+1}\sigma_i}{p} \end{pmatrix}$.

This program performs a rotation of $|\psi_0\rangle$ by the angle $x\frac{2\pi}{p}$ while reading x and then a rotation in opposite direction of the resulting vector by the angle $y\frac{2\pi}{p}$. Clearly, the program will end up in the state from F if and only if $x \equiv y \mod p$. Otherwise $(x-y)\frac{2\pi}{p} \geq \frac{2\pi}{p}$. Thus, **the statement of the lemma follows.**

Lemma 3. *$EQ_n(x, y)$ can be computed by a 1QBP of width $\Theta(n)$ with one-sided error*

$$O\left(\left(1 - \frac{1}{n \log n}\right)^{cn}\right). \tag{5.3}$$

for some constant c.

The proof of the lemma is based on combining 1QBPs computing $\mathrm{EQ}_{n,p_k}(x, y)$, $k = 1, \ldots, d$ in order to build up a 1QBP $\mathcal{B}$ that computes $\mathrm{EQ}_n(x, y)$, where $d = cn$, c is a constant, p_k is the kth prime number. Indeed, it is clear that for any $x \neq y$ $\mathrm{EQ}_{n,p_k}(x, y) = 1$ for at most n different p_k, since $x, y \leq 2^n$, and they can't contain more than n different prime factors. Thus, we combine d 1QBPs $\mathcal{A}_{p_k}$ as they were defined in **Lemma 2**. Let the set $F_{\mathcal{B}}$ of accepting states consists of a single vector.

$$F_{\mathcal{B}} = \{|0\rangle^d\}. \tag{5.4}$$

Thus, for every input $xy = \sigma \in \{0,1\}^n$ and every fixed prime number p_k, the error probability for this program $\mathcal{B}$ is bounded as it is shown below.

$$\Pr_{err}(\sigma, \mathcal{B}) \leq P_{err}(\sigma, \mathcal{A}_{p_k})^{d-n} \preccurlyeq \left(1 - \frac{1}{p}\right)^{(c-1)n} \tag{5.5}$$

From the Chebishev's theorem [HW79] it easily follows that if p is a n-th prime number then $p \in O(n \log n)$.

$$\Pr_{err}(\sigma, \mathcal{B}) \preccurlyeq \left(1 - \frac{1}{n \log n}\right)^{(c-1)n} \tag{5.6}$$

Since selection of the constant c was not fixed, **the statement of the lemma follows.**

Now we are to finish the proof of the theorem

We construct yet another 1QBP $\mathcal{C}$, building it up from $\frac{\log n}{c}$ copies of program $\mathcal{B}$. The final set is again defined to contain a single vector.

$$F_{\mathcal{C}} = \{|0\rangle^{\frac{d \log n}{c}}\}. \tag{5.7}$$

Since we still have a program that computes the function with one-sided error, it follows from **Lemma 3** that the error probability is bounded for every input σ.

$$\Pr_{err}(\sigma, \mathcal{C}) \leq P_{err}(\sigma, \mathcal{B})^{\frac{\log n}{c}} \preccurlyeq \left(1 - \frac{1}{n \log n}\right)^{n \log n} \underset{n \to \infty}{\longrightarrow} \frac{1}{e} < \frac{1}{2}. \tag{5.8}$$

The total witdth of the resulting 1QBP $\mathcal{C}$ is $2cn\frac{\log n}{c} = 2n \log n$. Thus, the theorem follows.

□

Next we present the main result about *Equality* function.

Theorem 25 (The Stronger Upper Bound). *The function $EQ_n(x, y)$ can be computed with one-sided error $o(1) \leq \frac{1}{7}, (n \to \infty)$ by a 1QBP of width $O(n)$, where*

$n = |xy|$ is the length of the input.

Proof. We shall build an $(O(n))-$1QBP $\mathcal{C}$.
Let us first introduce one qubit (acting in 2-dimensional Hilbert space) 1QBP*s* $\mathcal{A}_k = \langle T_k, |\psi_0\rangle, \Psi_k, F\rangle$, where $k \in \{1, \ldots, p-1\}$. Let $|0\rangle$ and $|1\rangle$ be two orthonormal fixed states to form a basis of the $\mathcal{H}^2$. Now we construct $\mathcal{A}_k$ for a $p \in (\sigma, 2\sigma) \cap PRIMES$. According to *Bertrand's postulate* there's always such p [Nag51].

1. The $QOBDD$ receives input $\sigma = xy$;
2. $|\psi_0\rangle = |0\rangle$;
3. $F = \{|0\rangle\}$;
4. $\Psi_k = \{|0\rangle, |1\rangle\}$
5. $T_k = (i, U_i(0), U_i(1))_{i=1}^n$
   (a) On the x-part of the input σ:
   $$U_i(\sigma_i) = \begin{pmatrix} \cos\frac{2\pi k\sigma_i 2^i}{p} & \sin\frac{2\pi k\sigma_i 2^i}{p} \\ -\sin\frac{2\pi k\sigma_i 2^i}{p} & \cos\frac{2\pi k\sigma_i 2^i}{p} \end{pmatrix};$$
   (b) On the y-part of the input σ:
   $$U_i(\sigma_i) = \begin{pmatrix} \cos\frac{2\pi k\sigma_i 2^i}{p} & -\sin\frac{2\pi k\sigma_i 2^i}{p} \\ \sin\frac{2\pi k\sigma_i 2^i}{p} & \cos\frac{2\pi k\sigma_i 2^i}{p} \end{pmatrix}.$$

Clearly if $x = y$ then $\mathcal{A}_k$ accepts σ with probability 1,

Definition 5.3.2. Call $\mathcal{A}_k$ *"good" for input $\sigma = xy, x \neq y$ if $\mathcal{A}_k$ rejects σ with probability at least $1/2$.*

Lemma 4. *For any $\sigma = xy, x \neq y$, at least $(p-1)/2$ of all $\mathcal{A}_k$ for different k are "good".*

Our branching program is defined so that, reading the input $\sigma = xy$, it first rotates the state vector (initially $|\psi_0\rangle$) by the angle $\theta_k(x) = \frac{2k\pi}{p}x$ in the direction towards $|1\rangle$ axis, and then backwards while reading y by the angle $\theta_k(y) = -\frac{2k\pi}{p}y$
After reading the input $\sigma = xy$ $(x \neq y)$ the branching program ends up in the state ψ^k.

$$|\psi^k\rangle = (\cos\theta_k)\,|0\rangle + (\sin\theta_k)\,|1\rangle, \tag{5.9}$$

$$\theta_k = \theta_k(x) + \theta_k(y) = \frac{2k\pi}{p}x - \frac{2k\pi}{p}y = \frac{2k\pi(x-y)}{p} \tag{5.10}$$

Since $k \in \{1, \ldots, p-1\}$, it is co-prime with p, thus, θ_k would constitute a finite cyclic additive group of residues modulo p. Clearly, the elements of the set of the angles $I = \{\theta_k | k \in \{1, \ldots, p-1\}\}$ are uniformly distributed on the circumference of a unit circle.
Now θ_k is "good" if and only if $\theta_k \in \left[\frac{\pi}{4}, \frac{3\pi}{4}\right] \cup \left[\frac{5\pi}{4}, \frac{7\pi}{4}\right]$, for only then have we $\cos^2\theta_k < \frac{1}{2}$ that means a θ_k to be "good". We conclude the proof.

$$\left| I \cap \left(\left[\frac{\pi}{4}, \frac{3\pi}{4}\right] \cup \left[\frac{5\pi}{4}, \frac{7\pi}{4}\right]\right)\right| \geq \left\lfloor \frac{p}{2} \right\rfloor \geq \frac{p-1}{2}. \tag{5.11}$$

Thus, the lemma follows.

Definition 5.3.3. We call a set of quantum programs $S = \{\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_t}\}$ "good" for $\sigma = xy, x \neq y$, if at least 1/4 of all its elements are "good" for this σ.

Lemma 5. *There is a set S of width-2 quantum branching programs with $|S| = t = \lceil 24 \log p \rceil$ which is "good" for all inputs $\sigma = xy$, that have $x \neq y$.*

We shall construct a "good" set S that will prove the lemma.
The construction is trivial: for a fixed input $\sigma \leq p-1$ a branching program from $\{\mathcal{A}_1, \ldots, \mathcal{A}_{p-1}\}$ is selected uniformly and randomly and added to initially empty set S.
Consider a sum $X = \Sigma_{i=1}^{n} X_i$, where $X_i = [\mathcal{A}_i \textit{ is not "good"}]$. By the definition of the set S and previous lemma, probability that every given program $\mathcal{A}_i \in S$ is not

"good" equals $q = \frac{1}{2}$. Now set parameter $\theta = \frac{1}{2}$ and apply Chernoff's bound.

$$\Pr\left[X \geq (1+\theta)qt\right] \leq e^{-\frac{\theta^2}{3}qt}, \tag{5.12}$$

that after substitution gives

$$\Pr\left[X \geq \frac{3}{4}t\right] \leq e^{-\log p} = \frac{1}{p}. \tag{5.13}$$

That is the probability of constructing a set S with less than $\frac{1}{4}$ of "good" (for any given $\sigma < p$) entries is not greater than $\frac{1}{p}$. Thus, the probability $\Pr\left[S \textit{ is not "good"}\right]$ that the set S is not "good" for *at least one* $\sigma < p$ is at most the following positive fraction.

$$\Pr\left[S \textit{ is not "good"}\right] \leq \frac{p-1}{p}. \tag{5.14}$$

Therefore, there exists a set which is "good" for all inputs $\sigma < p$. Recalling that $|S| = t = \lceil 24 \log p \rceil$ **the lemma follows.**

To finish the proof of the theorem we describe the 1QBP $\mathcal{C}$ accepting $\sigma = xy$ with probability 1 for $x = y$ and rejecting the input with probability $1 - o(1), (|\sigma| \to \infty)$, at least $\frac{6}{7}$, for $x \neq y$.

$\mathcal{C} = \langle T, |\psi_0\rangle, \Psi, F\rangle$:

1. The *QOBDD* receives input $\sigma = xy$;

2. $|\psi_0\rangle = |0\rangle$;

3. $F = \{|0\rangle\}$;

4. For each program $\mathcal{A}_i \in S$ we introduce corresponding state $|i\rangle$, which is $|1\rangle$ is replaced in each respective program for, when we combine them to result into $\mathcal{C}$. Thus, $\Psi = \{|0\rangle, \ldots, |t\rangle\}$, where t is from the **Lemma 5**.

5. T - the transition function is defined as a weighted with equal amplitudes superposition of the corresponding transformation for the programs $\mathcal{A}_i \in S$.

Notice that for inputs $\sigma = xy, x = y$ $\mathcal{C}$ never errs, the reason is that every $\mathcal{A}_i$ ends up in the accepting state $|0\rangle$. On the other hand, if $x \neq y$, the following lemma holds.

Lemma 6. *For program $\mathcal{C}$ constructed above, if input $\sigma = xy$ is such that $x \neq y$, the input is rejected with probability $P_{rej}(\sigma)$ at least $1 - o(1), (|\sigma| \to \infty)$. Furthermore, for such σ $P_{rej}(\sigma)$ is never less than $\frac{6}{7}$.*

Let us define ψ_n be the state of the program $\mathcal{C}$ after having read the input $\sigma = xy$, where $x \neq y$.

$$\psi_n = \sum_{i=0}^{t} a_i |i\rangle \tag{5.15}$$

According to the **Lemma 5**, at least $\frac{1}{4}$ of the programs $\mathcal{A}_i$ would reject it with probability at least $\frac{1}{2}$. Without loss of generality let all "good" programs constituting the "good" sequence of $\mathcal{C}$ correspond to basis vectors of $\mathcal{H}_C$, the Hilbert space assigned to $\mathcal{C}$, with indicies in $G = \left\{1, \ldots, \left\lfloor \frac{t}{4} \right\rfloor\right\}$. By the definition of $\mathcal{C}$, for all $0 < i \leq t$ on each subspace $\mathcal{H}_{\mathcal{A}_i} = Span(|0\rangle, |i\rangle)$ of the space $\mathcal{H}_C$, the program $\mathcal{C}$ behaves just like $\mathcal{A}_i$. Now let's call a subspace of $\mathcal{H}_{\mathcal{A}_i}$ "good" if it corresponds to a "good" program $\mathcal{A}_i$. By the definition of "goodness", projection $\psi_n^i, i \in G$ of the vector ψ_n on any of the "good" subspaces $\mathcal{H}_{\mathcal{A}_i}$ would satisfy the equation.

$$\psi_n^i = \frac{a_i}{\sqrt{a_i^2 + a_0^2}} |i\rangle + \frac{a_0}{\sqrt{a_i^2 + a_0^2}} |0\rangle, \tag{5.16}$$

where a_i are coefficients from the original representation of the vector ψ_n, and

$$\left| \frac{a_i}{\sqrt{a_i^2 + a_0^2}} \right| \geq \frac{1}{\sqrt{2}}. \tag{5.17}$$

Subsequently

$$a_i^2 \geq a_0^2; \tag{5.18}$$

Let $P_{acc}(\sigma) = a_0^2$, by the definition of program $\mathcal{C}$, be the probability of accepting the input σ.

$$1 = P_{acc}(\sigma) + P_{rej}(\sigma) = a_0^2 + \sum_{i=1}^{t} a_i^2 = a_0^2 + \sum_{i=1}^{\lceil t/4 \rceil} a_i^2 + \sum_{i=\lceil t/4 \rceil+1}^{t} a_i^2. \tag{5.19}$$

Now combining the two lines above we obtain following inequalities.

$$1 \geq a_0^2 + \frac{t}{4} a_0^2, \tag{5.20}$$

$$a_0^2 \leq \frac{4}{t+4}; \tag{5.21}$$

$$P_{rej}(\sigma) = 1 - a_0^2 \geq 1 - \frac{4}{t+4} = 1 - \frac{4}{\lceil 24 \log p \rceil + 4} \asymp \tag{5.22}$$

(where last equality follows from the **Lemma 5**)

$$\asymp 1 - \frac{4}{\lceil 24 \log 2^n \rceil + 4} = 1 - \frac{4}{24n + 4} \geq 6/7, \tag{5.23}$$

where last inequality obviously follows from $n > 0$. Thus, **the statement of the lemma follows.**

The error can be reduced arbitrarily running $d = d(\epsilon)$ copies of the program $\mathcal{C}$ taken and run uniformly at random. Clearly, the width of the program $\mathcal{C}$ is $O(\log p) = O(\log 2^n) = O(n)$, thus, completing the proof.

□

5.4 The upper bound for the Periodicity function

The next theorem is inspired by the proof of the **Theorem 25**.

Theorem 26. *For all $s \in (0, n]$ and $\forall \sigma \in \{0,1\}^n$ the function $Period_{s,n}(\sigma)$ can be computed with one-sided error $o(1) < 0.4, (s \to \infty)$ by a 1QBP of width $O(s)$.*

Proof. The main idea of the proof is to divide the input sequence into words of length s, and utilize the elementary 2-state branching programs for equality to compare all

resulted subsequences with some chosen subsequence of the set.
$\text{Period}_{s,n}(\sigma) = 1 \iff \forall i, 1 \leq i \leq n \ (\sigma_i = \sigma_{i+s \mod n})$. Thus, $\text{Period}_{s,n}(\sigma) = 1 \iff$ the elements of the input sequence form a cyclic group of size s over addition of their indicies modulo n. Subsequently the input having been split into words of length s would consist only of equal words if, and only if $\text{Period}_{s,n}(\sigma) = 1$.
The only obstacle on the way is that the input length might be not divisible by s. Following lemma adresses the question.

Lemma 7. *Given a binary sequence* $X = \{x\}_{i=1}^{ks+r}, k, r \in \mathbb{N} \cup \{0\}, s \in \mathbb{N}, r < s$, *consider a sequences of* s*-element subsequences of* X *with corresponding tails:*
$W = \left(x_1 \ldots x_s, x_{s+1} \ldots x_{2s}, \cdots, x_{(k-1)s+1} \ldots x_{ks}, x_{(k-1)s+r+1} \ldots x_{ks+r}\right);$
$t_{left} = x_1 \ldots x_s,$
$t_{right} = x_{(k-1)s+r+1} \ldots x_{ks+r}.$
Then following statement holds.

$$\forall w, w' \in W, \forall i, 1 \leq i \leq n(x_i = x_{i+s \mod n} \iff w = w'),$$

where $n = ks + r$.

We prove the lemma.
$\Rightarrow$ This direction is straightforward.
$\Leftarrow$ Suppose that $\forall w, w' \in W$ and $t_{left} = t_{right}$. Clearly, for all indicies i such that $1 \leq i \leq (k-1)s + r$ it holds that $x_i = x_{i+s}$. Now, if there is a number $i_0, (k-1)s + r + 1 \leq i_0 \leq ks + r$ such that $x_{i_0} \neq x_{i_0+s \mod n}$, it would contradict to what follows from that $t_{left} = t_{right}$.
Thus, **we have proved the lemma.**
It remains to present the 1QBP computing $\text{Period}_{s,n}(\sigma)$ with desired properties. It is done by cascading 2-state elementary programs from **Theorem 25** that compute equality, and then applying the same, as in previously mentioned theorem, technique to amplify correct answer probability.
Just like we did in the proof of the **Theorem 25**, we start by defining an elementary branching program that we shall use in following construction. Although this time we shall use three slightly different types of 2-state branching programs. Clearly,

σ can be considered as a sequence of words of length s, like in **Lemma 7**, $\sigma \to w_1 w_2 \dots w_{\lceil n/s \rceil}$, $w_1 = t_{left}$ and $w_{\lceil n/s \rceil} = t_{right}$. Then first type program ($j = 1$) would compute $\mathrm{EQ}_s(w_i, w_{i+1})$, second type ($j = 2$) program compute $\mathrm{EQ}_s(w_{i+1}, w_{i+2})$, $i = \overline{1, \lceil n/s \rceil - 2}$, and the third ($j = 3$) would check for equality only of the first and the last words: $\mathrm{EQ}_s(t_{left}, t_{right})$.
Now we construct $\mathcal{D}_{kj}$, $j \in \{1, 2, 3\}$ for a $p \in (2^s, 2^{s+1}) \cap PRIMES$. *Bertrand's postulate* assures us again there's always such p [Nag51].

1. The 1QBP receives input σ;
2. $|\psi_0\rangle = |0\rangle$;
3. $F = \{|0\rangle\}$;
4. $\Psi_{kj} = \{|0\rangle, |1\rangle\}$
5. $T_{kj} = (i, U_i(0), U_i(1))_{i=1}^n$, $n = |\sigma|$. We define the transition function explicitly only for $j = 1$, two other cases are easily derived from this one.
   (a) On the left part of the corresponding $2s$-symbol segment of σ:
   $$U_{i1}(\sigma_i) = \begin{pmatrix} \cos \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} & \sin \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} \\ -\sin \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} & \cos \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} \end{pmatrix};$$
   (b) On the right part of the corresponding $2s$-symbol segment of σ:
   $$U_{i1}(\sigma_i) = \begin{pmatrix} \cos \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} & -\sin \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} \\ \sin \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} & \cos \frac{2\pi k p_{\lceil i/s \rceil} \sigma_i 2^i}{p} \end{pmatrix}.$$
   Here $p_{\lceil i/s \rceil} \neq p, i = \overline{1, n}$ are prime numbers.

According to **Lemma 7**, if σ is so that $\mathrm{Period}_{s,n}(\sigma) = 1$ then $\mathcal{D}_k$ accepts σ with probability 1. Moreover, the program attempts to check the chain of equalities $w_1 = w_2 = \dots = w_{\lceil n/s \rceil}$ in order to reject all σ that have $\mathrm{Period}_{s,n}(\sigma) = 0$.
Let us write down the state, first of the three elementary programs would be in after having read the input. It is similarly easy to do for the rest of the three. We can

obtain the explicit expression for θ_k in the given above notation.

$$|\psi_1^k\rangle = (\cos\theta_k)\,|0\rangle + (\sin\theta_k)\,|1\rangle, \tag{5.24}$$

$$\theta_k = \frac{2\pi k}{p}\sum_{l=1}^{\lceil n/s\rceil - 1}(w_l - w_{l+1})p_l \tag{5.25}$$

$$k, w_l < p, l = \overline{1, \lceil n/s\rceil}; p \nmid p_l. \tag{5.26}$$

Thus, a result analogous to the **Lemme 4** holds for this case as well. Subsequently, since construction of the **Theorem 25** does not depend on the particular matter of the angles as far as **Lemma 4** holds, we can apply the same technique here.

We combine each of the three elementary programs analogously to the construction of the **Theorem 25**. Thus, for each of them we obtain a program of width $O(s)$ that computes its sub-chain of equalities with the probability of correct answer at least 6/7. Finally, we take the direct sum of the three to obtain the program $\mathcal{D}$. Clearly, error probability for that program would not exceed $127/343 < 0.4$. Asymptotically, the error probability would be bounded as we claimed (see the proof of the **Theorem 25** for details).

$$P_{err} \leq 1 - \left(1 - \frac{4}{24s+4}\right)^3 \asymp 3/s - 3/s^2 + 1/s \in o(s), (s \to \infty). \tag{5.27}$$

We present the formal description of the 1QBP $\mathcal{D}$ that computes $\text{Period}_{s,n}(\sigma)$.

$\mathcal{D} = \langle T, |\psi_0\rangle, \Psi, F\rangle$:

1. The $QOBDD$ receives input σ;

2. $|\psi_0\rangle = |0\rangle$;

3. $F = \{|0\rangle\}$;

4. $\Psi = \{|0\rangle, \ldots, |s'\rangle\}$, where $s' \in O(s)$.

5. $T = (i, U_i(0), U_i(1))_{i=1}^n$ The transition function definition is clear from the discussion presented above.

The theorem follows. □

5.5 The upper bound for the Semi-Simon function

This result is essentially a corollary of the **Theorem 25** in contrast to the previous theorem that was based on the same proof technique, but was not directly derived from the earlier statement.

Theorem 27. *For all $s \in (0, n]$, and for all $\sigma \in \{0,1\}^n$ the function Semi-Simon$_{s,n}(\sigma)$ can be computed with one-sided error $o(1) < 1/7, (s \to \infty)$ by a 1QBP of width $O(n)$.*

Proof. Computing of the equality function will be again in the core for the proof.

Definition 5.5.1. Denote $S_n = \{i | 1 \leq i \leq n, i \oplus s \neq i\}$. A set defined for all n-element binary sequences. Clearly, S does not depend on any particular function σ.

Lemma 8. *Set S_n can be partitioned into two sets of the same cardinality $S_n = S_n^1 + S_n^2$ so that for $l = 1, 2$ $\forall i, j \in S_n^l (j \neq i \oplus s)$.*

First, we show that following statement holds. For any $i \in S_n$, if $i \oplus s = j$ then there is no $k \in S_n, k \neq j$ so that $k = i \oplus s$.
Clearly, $k \neq i$, since it would yield $i = i \oplus s$ that contradicts $i \in S_n$. Moreover if $k = i \oplus s$, and $j = i \oplus s$, then $j \oplus s = i = k \oplus s$, that is $j = k$.
It is also clear, that if $i \neq i \oplus s$, $j = i \oplus s$ is contained in S_n. Since $j \in \{0, \ldots, n\}$, and $\{0, \ldots, n\} = S_n + S_n^c$. So if i does not equal its bitwise sum modulo two with s, i.e. not in S_n^c, it must be in S_n.
Now we take any $i \in S_n$ and add it to initially empty S_n^1, $j = i \oplus s$ we add to also initially empty S_n^2. Then we remove i and j from S_n. Since the latter set is finite and contains even number of elements that can be coupled into pairs $i, i \oplus s$ we will eventually exceed all elements of S_n. Note that the two new sets contain exactly the

same number of elements, by construction. Thus, desired partitioning is achieved. We showed **the lemma holds.**

Let us introduce two binary sequences, the two arguments to use in the equality computation that will finish our proof.

Definition 5.5.2.

$$S_{left}(\sigma) = \{\sigma_i\}_{i \in S_n^1},$$

$$S_{right}(\sigma) = \{\sigma_j\}_{j \in S_n^2},$$

where $S_{right}(\sigma)$ is ordered so that $S_{right}(\sigma)_i = \sigma_j$ and $j = i \oplus s$.

Now it is straightforward that

$$\text{Semi-Simon}_{s,n}(\sigma) = 1 \iff \text{EQ}_n(S_{left}(\sigma), S_{right}(\sigma)) = 1. \tag{5.28}$$

We already know how to compute equality efficiently, only here we shall have to rearrange rotation angles according to the permutation of the elements in $S_{right}(f)$. Clearly $|S_{left}(\sigma)| = |S_{right}(\sigma)| \in O(n)$, thus, **Theorem 25** gives us exactly what we claimed to achieve. We proved the linear upper bound for Semi-Simon problem.

□

5.6 The lower bounds for Equality, Periodicity and Semi-Simon

We prove lower bounds in two steps. First, we find the lower bound of the problems for deterministic OBDD. Then we use a general lower bound theorem for quantum OBDD (1QBP):

Theorem 28 ([AGK+05]). *Let $\epsilon \in (0, 1/2)$. Let $f_n(x_1, \dots, x_n)$ be a Boolean function which is ϵ-accepted (accepted with a margin ϵ) by a 1QBP Q. Then it holds that*

$$width(Q) = \Omega(\log_2 width(P)), \tag{5.29}$$

where P is a deterministic OBDD of minimal width computing $f_n(x_1, \dots, x_n)$.

Let's start with the lower bound for the *Equality* function.

Theorem 29. *Let $\epsilon \in (0, 1/2)$. If the function $EQ_n(x, y)$ is ϵ-accepted by a 1QBP Q then $width\,(Q) \in \Omega\,(n)$, where $n = |xy|$ is the length of the input.*

Proof. As we described in the beginning of the section, first we consider deterministic OBDD complexity for the function.

Lemma 9. *If the function $EQ_n(x, y)$ is computed by a deterministic OBDD P then $width\,(P) \in \Omega\,(2^n)$, where $n = |xy|$ is the length of the input.*

We shall prove the lemma by contradiction. Let xy be the input, $|xy| = n = 2m$. Suppose that there's a program P of $width\,(P) < 2^m$. Denote by $Vertex(x)$ the vertex that the path defined by the assignment to x leads to. There are 2^m possible assignments to x. On the other hand, by the hypothesis, there are at most $2^m - 1$ states on each level of the OBDD. That is, there exist two different binary sequences σ_1 and σ_2 – assignments to x – such that $Vertex(\sigma_1) = Vertex(\sigma_2)$, by the *"pigeon hole" principle*. Now whatever input for y would follow in our read-once leveled (oblivious) branching program, the two comparisons

$$\sigma_1 \stackrel{?}{=} y$$

and

$$\sigma_2 \stackrel{?}{=} y,$$

for any fixed y, could not be distinguished by the program. Thus, having an input $y = \sigma_1$, the program would either accept both of the combinations $\sigma_1\sigma_1, \sigma_1\sigma_2$ or reject them, thus, contradicting the fact it was computing function $\mathrm{EQ}_n(x, y)$ (see **Definition 5.1.1**). The lemma follows.

Final step of the proof is to refer to the **Theorem 28**. Which assures every 1QBP Q computing $\mathrm{EQ}_n(x, y)$ would satisfy relation 5.29:

$$width\,(Q) \in \Omega\,(\log_2 2^n) = \Omega(n), \tag{5.30}$$

for some constant c. That concludes the proof.

□

In order to prove a lower bound for the *Periodicity* function, we reduce *Equality* to *Periodicity*.

Theorem 30. *Let $\epsilon \in (0, 1/2)$. If for all $\sigma \in \{0,1\}^n$ the function $Period_{s,n}(\sigma)$ is ϵ-computed by a 1QBP Q then $width(Q) \in \Omega(s)$, where s is the period parameter.*

Proof. We simply reduce $\mathrm{EQ}_n(x, y)$ to $\mathrm{Period}_{s,n}(xy)$. Indeed, let $|xy| = 2s = n$ and $\sigma = xy$ is a concatenation of words x and y. It is straightforward that the following holds.

$$\mathrm{EQ}_n(x, y) = \mathrm{Period}_{s,n}(\sigma). \tag{5.31}$$

Thus, the lower bound for the *Periodicity* follows from the lower bound for the *Equality*. This proves the theorem. □

Similarly, we prove the lower bound for the *Semi-Simon* problem.

Theorem 31. *Let $\epsilon \in (0, 1/2)$. If for all $\sigma \in \{0,1\}^n$ the function $Semi\text{-}Simon_{s,n}(\sigma)$ is ϵ-computed by a 1QBP Q then $width(Q) \in \Omega(n)$.*

Proof. In order to reduce $\mathrm{EQ}_n(x, y)$ to $\text{Semi-Simon}_{s,n}(\sigma)$ we notice that the latter is essentially an equality computation. But its argument bits are mixed up according to the permutation defined by s. Let's once more write down the definition for *Semi-Simon* function.

$$\text{Semi-Simon}_{s,n}(\sigma) = 1 \iff \forall i \in [1, n] \, (\sigma_i = \sigma_{i \oplus s}). \tag{5.32}$$

For an arbitrary input σ and a positive s, computing function $\text{Semi-Simon}_{s,n}(\sigma)$ is equivalent to evaluating following equality.

$$\sigma_1 ... \sigma_n \stackrel{?}{=} \sigma_{1 \oplus s} ... \sigma_{n \oplus s}. \tag{5.33}$$

Now let us define s as shown below.

$$s = \overbrace{10 \ldots 0}^{n} \tag{5.34}$$

For s defined above, **Expression 5.33** turns into the desired *Equality* evaluation.

$$\begin{aligned}\sigma_1 ... \sigma_{n/2} \sigma_{n/2+1} \sigma_n &\stackrel{?}{=} \sigma_{n/2+1} ... \sigma_n \sigma_1 \dots \sigma_{n/2} \sim \\ \sim \sigma_1 ... \sigma_{n/2} &\stackrel{?}{=} \sigma_{n/2+1} ... \sigma_n \sim \\ &\sim \mathrm{EQ}_n(x, y), \text{for } \sigma = xy.\end{aligned} \tag{5.35}$$

Finally, we notice that for $\sigma = xy$ n would be even, and our reduction goes as follows.

$$\mathrm{EQ}_n(x, y) = \text{Semi-Simon}_{s,n}(xy), \text{ where } s = n/2. \tag{5.36}$$

That concludes the proof. □

In the next chapter we generalize our techniques to prove a linear upper bound and provide lower bounds for the *hidden subgroup test* function.

Chapter 6

The Hidden Subgroup Problem

> I hate quotations, tell me what you know.
>
> — Ralph Waldo Emerson

6.1 Introduction

The functions considered in the previous chapter were invented in order to get closer to the *hidden subgroup problem*. Finally, we consider the *Hidden Subgroup Problem* itself. This problem is the one that *factoring integers* and *discrete logarithm* can be reduced to. There is no efficient solution of these problems for non-quantum computers known so far. That is, although these problems belong to **BQP**, the class of efficient quantum algorithms, it is still an open question whether they are in **BPP**, the class of efficient algorithms (non-quantum). The RSA [RSA78] open-key cryptographic system relies on the assumption that those problems are not in **BPP**. The system is used in banking, secure Internet transactions etc. This is where the main drive of the interest to the hidden subgroup problem comes from.
The first quantum polynomial time algorithm for the *abelian stabilizer*, which is a special kind of the *hidden subgroup problems*, was found by Kitaev in 1995, there the author also generalized the results of Shor for descrete logarithm and factoring

integers [Sho97]. The results of Shor and Kitaev were carefully studied by Jozsa [Joz97]. He analyzed how all that problems can be reduced to the Abelian *hidden subgroup problem*. Since then many more research works were dedicated to the *hidden subgroup problem*. Interesting positive and negative results for the non-abelian *hidden subgroup problem* were obtained by Grigni, Schulman, Monica and Umesh Vazirani [GSVV01]. We also mention remarkable results obtained by Friedl, Magniez, Santha and Sen [FMSS03]. They considered *property testers*, where the computational device is allowed to read only a small fraction of the input. For several related to the *hidden subgroup problem* properties, like *Periodicity*, efficient *quantum* testers were found. A comprehensive review of the *hidden subgroup problem* related research with open problem is recently presented by Lomont [Lom04].
Already in one of the early works where the *hidden subgroup problem* was considered, Høyer [Høy97] noticed that the *graph isomorphism* (See p. 154) was easily reduced to the *non-abelian "unknown group problem"*. In the results we prove below, we consider a *non-abelian* version of the *hidden subgroup problem*. Although, the algorithm of this chapter is of linear width in the group size, the latter is exponential in the size of the considered graphs in the *graph isomorphism* problem. Thus, it is too early to celebrate an efficient solution of a **NPI** (See p. 154) problem by a quantum computer. Nevertheless, our algorithm has no match so far, and it is a good candidate for at least a polynomial speed-up over classical counterparts.
The lower bounds, that conclude this chapter, show that our upper bound is "almost" tight. The communication complexity lower and upper bounds we prove show that our quantum OBDD lower bound can not be improved using the same *communication complexity* technique. Thus, any improvement is a matter of a separate research, when possible.
In order to investigate the *hidden subgroup problem* complexity in Quantum Branching Programs setting, we define its *decision version*.

Remark 4 *(Important remark!).* We say simply *coset* everywhere below. However, we actually mean either *left* cosets or *right* cosets. The choice is not crucial, but once we make it, it must be read either *left* coset or *right* coset everywhere!

Definition 6.1.1. Let G be a finite group of order $n = |G|$. Let K be a subgroup of G. Let X be a finite set. For a binary sequence $x \in \{0,1\}^{|G| \log_2 |X|}$ let $\sigma \in X^{|G|}$ be a sequence of length n over X encoded by x in binary.

$$\mathrm{HSP}_{G,K,X}(\sigma) = \begin{cases} 1, & \text{if } \forall a \in G\ \forall i,j \in aK\ (\sigma_i = \sigma_j) \\ & \text{and } \forall a,b \in G\ \forall i \in aK\ \forall j \in bK \\ & (aK \neq bK \Rightarrow \sigma_i \neq \sigma_j); \\ 0, & \text{otherwise.} \end{cases}$$

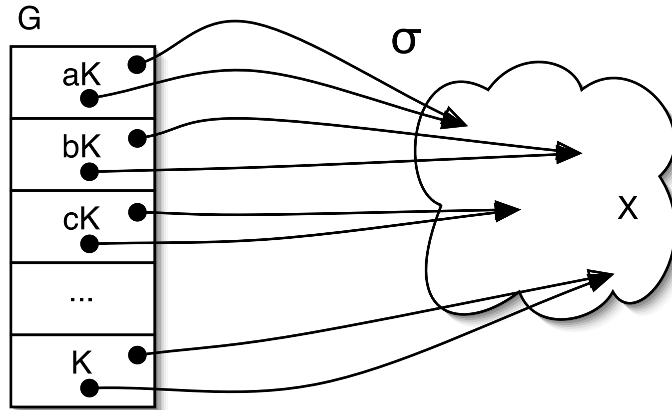

Figure 6.1: Hidden subgroup problem

The *hidden subgroup test function* asks to decide whether $f : G \to X$ "hides" the subgroup K in the group G, where X is a finite set. Our program receives G, K, and X as *parameters*, and function f as an *input string* containing values of f it takes on X. The values are arranged in the lexicographical order of their binary encodings. See **Definition 6.1.1**. The "proper" *Simon test* function defined below is a special case of the *hidden subgroup test* function.

Definition 6.1.2. For $n = 2^l c\ (l, c \in \mathbb{N}), s \in \{0, \dots n-1\}$ and for $\sigma \in \{0, \dots 2^c - 1\}^n$, a sequence, we define the *Simon* function $\mathrm{Simon}_{s,n}(\sigma)$.

$$\mathrm{Simon}_{s,n}(\sigma) = \begin{cases} 1 & \sigma_i = \sigma_j \iff (i-j) \in \{0, s\}, i, j = \overline{0, n-1}; \\ 0 & \text{otherwise.} \end{cases}$$

Where $\oplus$ is a bitwise addition modulo 2.

It is easy to reduce *Simon* function to the *Hidden Subgroup* function.

Lemma 10. *Let $\epsilon \in (0, 1/2)$. If for all $\sigma \in \{0,1\}^n$ the function $HSP_{G,K,X}(\sigma)$ is ϵ-computed by a 1QBP Q then $width(Q) = \Omega((G : K) \log |X|)$.*

Proof. Let $K = \{0, s\}$ and $X = \mathbb{B}$. Let group $G = (\mathbb{Z}_{2^l}, \oplus)$, the group operation $\oplus$ is from the definition of the *Simon* function (See **Definition 5.1.3)**. Let's notice several truth statements:

$$
\begin{aligned}
&aK = \{a, a \oplus s\},\\
&\qquad aK \neq bK \iff (a-b) \notin K, \forall i \in a, a \oplus s\ \forall j \in b, b \oplus s\ (\sigma_i \neq \sigma_j) \iff\\
&\qquad\qquad \sigma_a \neq \sigma_b \text{ and } \sigma_a \neq \sigma_{b\oplus s} \text{ and } \sigma_{a\oplus s} \neq \sigma_b \text{ and } \sigma_{a\oplus s} \neq \sigma_{b\oplus s}.
\end{aligned} \tag{6.1}
$$

Then by the definition of $\mathrm{HSP}_{G,K,X}(\sigma)$:

$$
\mathrm{HSP}_{G,K,X}(\sigma) = \begin{cases} 1, & \text{if } \forall a \in \{0, \ldots n-1\}(\sigma_a = \sigma_{a\oplus s}) \\ & \text{and } \forall a, b \in \{0, \ldots n-1\} \\ & (a - b \notin \{0, s\} \Rightarrow \sigma_a \neq \sigma_b); \\ 0, & \text{otherwise.} \end{cases} \tag{6.2}
$$

$$
n = |G| = 2^l. \tag{6.3}
$$

It remains to compare the expression above with the definition of the *Simon test* function. □

First in this chapter we present the upper bound for the most general problem of its family – the Hidden Subgroup Problem.

Theorem. *The function $HSP_{G,K,X}(\sigma)$ can be computed with two-sided error $o(1) \leq 0.4, (|\sigma| \to \infty)$ by a 1QBP of width $O(|G/K| \log |X|)$.*

Finally, we prove several lower bound theorems that represent the problems complexity related to different parameters of the problem. The worst-case parameter lower

bound asymptotically matches the parameter independent upper bound presented above.

6.2 The upper bound for the hidden subgroup test function

Theorem 32. *The function* $HSP_{G,K,X}(\sigma)$ *can be computed with two-sided error* $o(1) \leq 0.4, (|\sigma| \to \infty)$ *by a 1QBP of width* $O((G:K)\log|X|)$.

Proof. In order to prove the upper bound we shall construct a linear width 1QBP that computes the desired function.

Lemma 11. *In order to compute function* $HSP_{G,K,X}(\sigma)$ *it is enough to perform the following computations.*

1. *For each coset compute equalities of all values from the input string that have indicies from this coset;*

2. *For each coset take a representative, and compute equality for all of the representatives, then take a negation of the answer.*

Let's prove the lemma. Indeed, if $\mathrm{HSP}_{G,K,X}(\sigma) = 1$ then by the definition following holds:

1. For any coset of K, all its elements are equal;

2. The function values on any two elements taken from different cosets are not equal.

Thus, one direction of the lemma follows.
Now suppose the two conditions of the lemma hold. Does it follow that $\mathrm{HSP}_{G,K,X}(\sigma) = 1$?
First conditions of the lemma and the *hidden subgroup test function* definition are identical. It remains to check if it is enough to test the inequalities among arbitrary representatives taken one from each coset, if we want to output $\mathrm{HSP}_{G,K,X}(\sigma)$.

If the function indeed takes different values on different cosets, the algorithm would not produce a wrong output. On the other hand, when the function values coincide for some arguments taken from different cosets, there still can be elements that actually have different images over f. However, it doesn't turn to be a problem! Let aK and bK be the two different cosets that contain elements $d \in aK$ and $c \in bK$ respectively, such that:

$$\sigma_d = \sigma_c.$$

Let's fix the element $c \in bK$. There can be only two possible cases.

1. For all $d \in aK(\sigma_d = \sigma_c)$;

2. There exits $d' \in aK(\sigma_d \neq \sigma_c)$.

Clearly, in the first case our algorithm can select any element of aK to check for inequalities, as it is proposed in the second condition of the lemma.
In the second case aK apparently contains elements that have different images over the function f encoded by σ. That means, the first condition of the lemma would fail to be satisfied.
The argument for bK are totally symmetric.
Thus, indeed, the two conditions of lemma are satisfied if, and only if $\text{HSP}_{G,K,X}(\sigma) = 1$. **We have proved the lemma.**
As a matter of fact, **Lemma 11** gives us a model of the algorithm that we are about to build. It will consist substantially of two parts. First, for every coset, compute equalities of images of its elements over function f, given as the input. Second, for every coset, take an arbitrary element, calculate the equality of all cosets representatives, and take a negation of the result. Finally, the result of the algorithm is the conjunction of the results of the first and the second stages.
We should fix some notation that will also determine the structure of our algorithm.

- Denote $\mathcal{A}$ the subroutine, encompassing $\mathcal{A}_i$ for all $i \in G/K$, that computes the first statement of the **Lemma 11**.

- Denote $\mathcal{A}_i$ the subroutine that calculates equalities for the coset $i \in G/K$.

- Denote $\mathcal{A}_{i,s}$, $s = \overline{1,3}$ the subroutines used in computation of the equalities for the coset $i \in G/K$.
- Denote $\mathcal{A}_{i,s,k}$, $s = \overline{1,3}$ the subroutines the program $\mathcal{A}_{i,s}$ consists of. The parameter k will be specified later, as we did in the proof of **Theorem 25**
- Denote $\mathcal{B}$ the subroutine that calculates negation of the equality of the representatives of different cosets. We can fix particular representatives arbitrarily. Say, for a coset aK take always an element a as a representative, if aK does not equal to a coset that has already been considered.

Let's first show how to compute the function of σ defined below for a given coset aK and for all $i, j \in aK$.

$$\bigwedge_{i,j \in aK, i \neq j} \sigma_i = \sigma_j. \tag{6.4}$$

We already know how to compute each of the the equalities within the width $O(\log |X|)$ (See **Theorem 25**)! In order to compute this sequence of equalities we shall construct three slightly different types of branching programs. Then we combine them like we did in the proof of **Theorem 25** before.
Let's introduce integer indicies for the sequence $\Sigma = \{\sigma_i\}_{i \in aK}$ ordered lexographicaly (consistently with the input) $\Sigma = \{\sigma'_l\}_1^m$, where $m = |aK|$. Then we can define the three sorts of programs we shall use to compute the chain of equalities 6.4 by the type of computation we assign to each of them.

1. $\sigma'_l = \sigma'_{l+1}$ only for odd values of $l < k$;
2. $\sigma'_l = \sigma'_{l+1}$ only for even values of $l < k$;
3. $\sigma'_1 = \sigma'_k$.

Now we show how to construct $\mathcal{A}_{i,s,k}$, $s \in \{1, 2, 3\}$ for a $p \in (|X|, 2|X|) \cap PRIMES$. *Bertrand's postulate* [Nag51] assures us such there's always such p.

1. The 1QBP receives input $x = bin\,(\sigma)$;
2. $|\psi_0\rangle = |0\rangle$;

3. $F = \{|0\rangle\}$;

4. $\Psi_{i,s,k} = \{|0\rangle, |1\rangle\}$

5. $T_{i,s,k} = (i, U_i(0), U_i(1))_{i=1}^{n}$, $n = |\sigma|$. We define the transition function explicitly only for $s = 1$, two other cases are easily derived from this one. Also $U_i(0) = U_i(1) = I$ for all $\sigma_i, i \notin aK$. Now consider only the transformations applied upon consecutive reading the sequence Σ.

   (a) On the left part of the corresponding $2\log|X|$-symbol segment of the input:

$$U_{i,1,k}(\sigma_i) = \begin{pmatrix} \cos\frac{2\pi k p_i \sigma_i' 2^i}{p} & \sin\frac{2\pi k p_i \sigma_i' 2^i}{p} \\ -\sin\frac{2\pi k p_i \sigma_i' 2^i}{p} & \cos\frac{2\pi k p_{i/} \sigma_i' 2^i}{p} \end{pmatrix};$$

   (b) On the right part of the corresponding $2\log|X|$-symbol segment of the input:

$$U_{i,1,k}(\sigma_i) = \begin{pmatrix} \cos\frac{2\pi k p_i \sigma_i' 2^i}{p} & -\sin\frac{2\pi k p_i \sigma_i' 2^i}{p} \\ \sin\frac{2\pi k p_i \sigma_i' 2^i}{p} & \cos\frac{2\pi k p_{i/} \sigma_i' 2^i}{p} \end{pmatrix};$$

   Here $p_i \neq p, i = \overline{1, n}$ are prime numbers.

If all $\sigma_i, i \in aK$ are indeed equal then $\mathcal{A}_{k,i}$ accepts σ with probability 1. Moreover, the program attempts to check the chain of equalities $\sigma_1 = \sigma_2 = \ldots = \sigma_m$ in order to reject all σ that don't satisfy the equalities.
Let us write down a state, the first of the three elementary programs would be in after having read the input. It is similarly easy to do for the other two programs. In the given above notation we can obtain equation for θ_k.

$$|\psi_1^k\rangle = (\cos\theta_k)\,|0\rangle + (\sin\theta_k)\,|1\rangle, \tag{6.5}$$

$$\theta_k = \frac{2\pi k}{p} \sum_{l=1}^{m} (\sigma_l' - \sigma_{l+1}') p_l \tag{6.6}$$

$$k, \sigma_l' < p, l = \overline{1, m}; p \nmid p_l. \tag{6.7}$$

Thus, a result analogous to the **Lemma 4** holds for this case as well. Subsequently, since construction of the **Theorem 25** does not depend on the particular matter of the angles as far as **Lemma 4** holds, we can apply the same technique here.

We combine each of the three elementary programs analogously to the construction of the **Theorem 25**. Thus, for each of them we obtain a program of width $O(\log |X|)$ that computes its sub-chain of equalities with the probability of correct answer at least $6/7$. Finally, we take the direct sum of the three programs to obtain the program $\mathcal{A}_i$ for all $i \in G/K$. This program will not exceed one-sided error probability $127/343$. Then we take direct sum of all programs $\mathcal{A}_i$ to obtain the program $\mathcal{A}$. Clearly, the error probability for that program would not exceed $(127/343)^{|G/K|} < 0.4^{|G/K|}$.

Analogously to what we did computing the chain 6.4 for programs $\mathcal{A}_i$, we can compute the chain for program $\mathcal{B}$:

$$\sigma_{i_1} = \sigma_{i_2} = \ldots = \sigma_{i_{|G/K|}}. \tag{6.8}$$

We can do this within one-sided error probability 0.4, as we have stated earlier. In order to obtain the result claimed for program B, we need to take the complementation of the result that computation of the chain 6.8 would give us.

We combine the two subroutines $\mathcal{A}$ and $\mathcal{B}$ in order to obtain our final program. We use the same direct sum method of combining elementary branching programs into more complex ones. Thus, ending up with a program that, according to the **Lemma 11**, *hidden subgroup test function* within two-sided error 0.4.

Asymptotically, the error probability would be bounded as we claimed (see the proof of the **Theorem 25** for details).

That finishes the proof of the theorem.

□

6.3 The lower bound for the hidden subgroup test function

Let us refresh here the definition of the *decision version* of the hidden subgroup problem.

Definition 6.3.1. Let G be a finite group of order $n = |G|$. Let K be a finite subgroup of G. Let X be a finite set, such that $|X| \geq (G : K)$. For a binary sequence $x \in \{0,1\}^{|X||G|}$ let $\sigma \in X^{|G|}$ be a sequence of length n over X encoded by x in binary.

$$\mathrm{HSP}_{G,K,X}(\sigma) = \begin{cases} 1, & \text{if } \forall a \in G\ \forall i,j \in aK\ (\sigma_i = \sigma_j) \\ & \text{and } \forall a,b \in G\ \forall i \in aK\ \forall j \in bK \\ & (aK \neq bK \Rightarrow \sigma_i \neq \sigma_j); \\ 0, & \text{otherwise.} \end{cases}$$

We develop a simple language to be used to formulate our technique of proving the lower bound.

First, we notice that an assignment to the string σ from the definition above is also an assignment to the input variables for any program computing $\mathrm{HSP}_{G,K,X}(\sigma)$. That is, a string $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_{|G|}),\ \sigma_i \in X, i = 1, \ldots, |G|$ defines the input variables set $(\sigma_1, \sigma_2, \ldots, \sigma_{|G|})$ that we shall denote with the same letter σ.

The indices $i = 1, \ldots, |G|$ are in one-to-one correspondence with the group elements of G. We shall further refer to the indices as to the elements of G. Naturally, if we mention a group structure on the set of indices, we mean the group structure of G, and not the structures of the semi-ring $\mathbb{N}$.

In the hidden subgroup problem we have sets of the algebraic structure of the group G. On the other hand, the communication model (See pages 34, 36) has its own sets. Namely, for an input σ we consider a partition Π (See **Definition 2.4.1**, page 34), that defines the two sets: $\Pi_{L,\sigma}$ and $\Pi_{R,\sigma}$. In order to keep the two systems of sets separate, yet to be able to make statements containing the input partition and the algebraic properties of G, we present the following definitions.

Definition 6.3.2. For a group G, its subgroup K, and a set $A \subset G$, we define

$$\mathcal{C}_{G,K}(A) := \{C \mid \exists a \in A (C \text{ is a coset of } K \wedge a \in C)\}$$

A set of cosets that have elements in both of the two disjoint sets is called the set of *common cosets*.

Definition 6.3.3. For a group G, a subgroup K, and two disjoint sets $A, B \subset G$, define the set of *common cosets*.

$$\mathcal{CC}_{G,K}(A; B) := \{C \mid \exists a \in A, b \in B (C \text{ is a coset of } K \wedge a, b \in C)\}.$$

We can obviously define the set of the *common cosets of a partition* Π of the input σ.

$$\mathcal{CC}_{G,K}(\Pi) := \mathcal{CC}_{G,K}(\Pi_{L,\sigma}; \Pi_{R,\sigma}).$$

We call $\#\mathcal{CC}_{G,K}(A; B) = |\mathcal{CC}_{G,K}(A; B)|$ the *common cosets number* of the sets A and B. Analogously, we say $\#\mathcal{CC}_{G,K}(\Pi)$ is the common coset number of the partition Π.

As well as there are common cosets for a pair of given subsets of the group G, there can be cosets, that are not common for the two subsets.

Definition 6.3.4. For a group G, a subgroup K and two disjoint sets $A, B \subset G$ define the set of *independent cosets* of B with respect to A.

$$\mathcal{IC}_{G,K,A}(B) := \mathcal{C}_{G,K}(B) \setminus \mathcal{CC}_{G,K}(A; B).$$

For a partition Π of the input σ, *the set of the independent cosets of the partition* Π is defined as follows.

$$\mathcal{IC}_{G,K}(\Pi) := \mathcal{IC}_{G,K,\Pi_{L,X}}(\Pi_{R,X}) \cup \mathcal{IC}_{G,K,\Pi_{R,X}}(\Pi_{L,X}).$$

We call $\#\mathcal{IC}_{G,K,A}(B) = |\mathcal{CC}_{G,K}(\Pi)|$ the *independent cosets number* of the set B with respect to the set A. Analogously, the *independent cosets number* of the partition Π

is $\#\mathcal{IC}_{G,K}(\Pi) = |\mathcal{IC}_{G,K}(\Pi)|$.

We shall consider a special kind of partitions that we call *cuts*.

Definition 6.3.5. A partition Π of the input variables σ is a *cut* if there is an integer k such that for all integer $i < k$ $\sigma_i \in \Pi_{L,X}$ and for all $j \geq k$ $\sigma_j \in \Pi_{R,X}$. We shall call this integer k the *cuting point*.

Finally, we present an exclusively technical definition. Its purpose is to simplify reading of the proof.

Definition 6.3.6. We say that a coset C *takes a value* x for a string σ if all variables with their indexes in C are assigned the value x.

This is a valid notion for all input strings σ, to which the function $\mathrm{HSP}_{G,K,X}(\sigma)$ assigns the value one. For any given coset, all its member variables are assigned the same input value in any of that strings.
The proof of the lower bound would mainly consists of the three steps:

1. We begin by proving a theorem that establishes connection between *one-way communication complexity* and *one-way quantum branching program complexity* of a problem;

2. Next, we prove bounds on the *one-way communication complexity* of the function $\mathrm{HSP}_{G,K,X}(\sigma)$;

3. Finally, we obtain our desired lower bound as a corollary of the two previous theorems.

Theorem 33. *Let $\epsilon \in (0, 1/2)$ be a constant. Let Q_f be a one-way quantum branching program that $(1/2+\epsilon)$-computes (computes with the margin ϵ) function $f_n \in \mathbb{B}_n$. Then for any partition Π of the input, following holds.*

$$width(Q_f) = \Omega(CC_1(f, \Pi)).$$

Proof. We prove this theorem in two steps. First we show how to relate communication complexity to the width of branching programs. Then we apply the general lower bound theorem to translate this relation to the quantum branching programs.

Lemma 12. *For any deterministic OBDD P representing a Boolean function $f \in \mathbb{B}_n$, let Π be a cut of the input variables, and let k be the cutting point of Π. Then P defines a* two party one-way communication protocol Φ*.*

$$width\,(P) \geq 2^{CC_1(f)-1},$$

where $CC_1\,(f)$ is the deterministic one-way two-party communication complexity of the function f.

The proof is done in one step. Let $X \in \mathbb{B}^n$ be the input of the program P. For the program P that represents the function f, we define a one-way two-party communication protocol $\langle \Phi, \Pi \rangle$ computing the very same function (See **Definition 2.4.14**). Let Alice read the variables in $\Pi_{L,X}$. Let Bob read the variables in $\Pi_{R,X}$. The program P is represented by a leveled (See **Definition 2.3.5**) directed graph. Now let us number all the vertices on the kth level of P.

Alice simulates computation of the program P on the first k input variables. Any computation apart from sending messages to the parties is "free of charge" in communication complexity. Let the message c that Alice is supposed to send Bob be the number of the vertex of the kth level, where the path in P defined by the first k input variables ends.

Bob can obviously continue the simulation of P having received the message from Alice. Finally, Bob will output the desired value $f(X)$. The idea is illustrated on the **Figure 6.2**.

This correctly defines the protocol $\langle \Phi, \Pi \rangle$.

There can not be more than $width\,(P)$ vertices on the kth level of P. Thus, it is enough to send the $\lfloor \log_2 width\,(P) \rfloor + 1$ bits long message s, as it is described in the protocol. **The lemma is evident.**

For the conditions of this theorem, **Theorem 28** implies following relation.

$$width\,(Q_f) = \Omega\,(\log_2 width\,(P))\,, \tag{6.9}$$

where P is a deterministic OBDD of minimal width computing f_n.

$width\,(Q_f) = \Omega\,(\log_2 width\,(P))$ **Lemma 12** implies

$$\Rightarrow width\,(Q_f) = \Omega\,(CC_1\,(f))\,. \tag{6.10}$$

This proves the statement. □

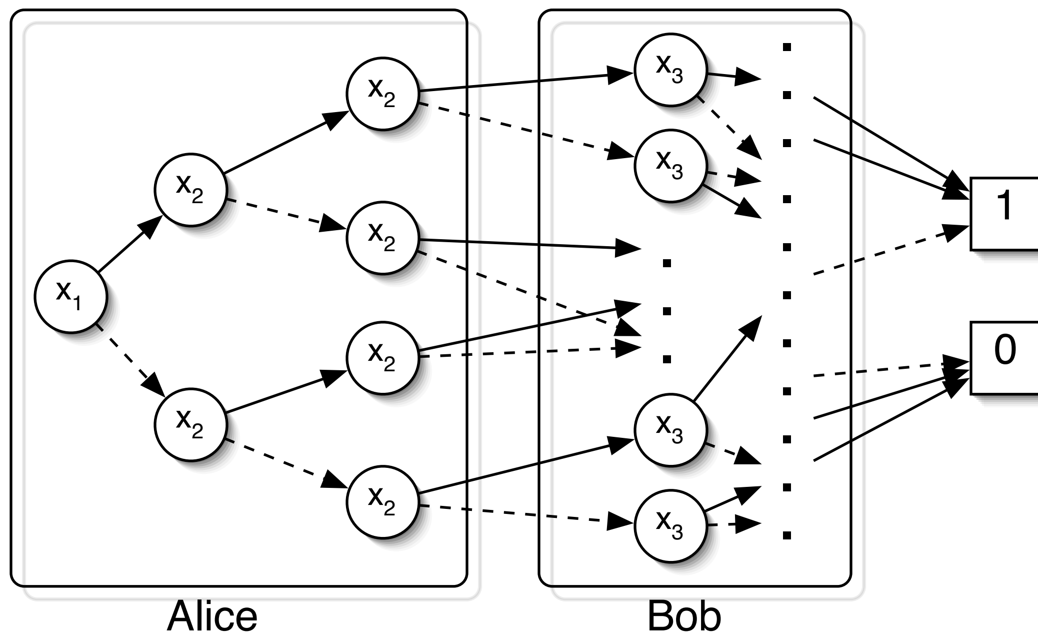

Figure 6.2: Communication protocol simulating OBDD

Now we state the communication complexity lower bound for the hidden subgroup function.

Theorem 34. *Let K be a non-trivial subgroup of a finite group G. Let X be any finite set, such that $|X| \geq (G : K)$. For any partition Π, one-way communication*

complexity according to Π *of the hidden subgroup test function is bounded as follows.*

$$CC_1\left(HSP_{G,K,X}\left(\sigma\right),\Pi\right) = \Omega\left(\log_2\binom{|X|}{\#\mathcal{CC}_{G,K}\left(\Pi\right)}\#\mathcal{CC}_{G,K}\left(\Pi\right)! + \left[\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}\left(\Pi_{R,\sigma}\right) > 0\right]\log_2\binom{|X| - \#\mathcal{CC}_{G,K}\left(\Pi\right)}{\#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}\left(\Pi_{L,\sigma}\right)}\right).$$

Proof. Our combinatorial proof relies on the concept of *communication matrix* (See **Definition 2.4.16** introduced on the page 44). It is more elegant to use short notation $CM := CM\left(\text{HSP}_{G,K,X}\left(\sigma\right),\Pi\right)$, where it can cause no confusion, what function, and according to what partition, is considered. We shall use common notation CM_i to denote the ith row of the matrix CM.

Other shorthands used throughout the proof are presented in the list below.

$$l := |X|; \tag{6.11}$$

$$d := \#\mathcal{C}_{G,K}\left(\Pi_{L,\sigma}\right); \tag{6.12}$$

$$n := \#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}\left(\Pi_{L,\sigma}\right), \qquad \text{notice that } n = (G:K) - d; \tag{6.13}$$

$$m := \#\mathcal{CC}_{G,K}\left(\Pi\right), \qquad \text{notice that } m \leq d \leq l. \tag{6.14}$$

The function $\text{HSP}_{G,K,X}\left(\sigma\right) = 1$ if, and only if, the values of the input variables with indices from the same coset equal but never equal if the variables have indices from different cosets.

Obviously, for all strings δ that fail to satisfy the two conditions, corresponding rows of CM must consist only of zero entries. Let E_Π be the set of all "bad" row indices for a given partition Π.

$$E_\Pi = \left\{\delta \middle|\ \forall\gamma \in X^{|\Pi_{R,\sigma}|}\ \left(\delta \in X^{|\Pi_{L,\sigma}|} \wedge \text{HSP}_{G,K,X}\left(\delta;\gamma\right) = 0\right)\right\}. \tag{6.15}$$

Lemma 13. *Let* CM *be a communication matrix of* $HSP_{G,K,X}\left(\sigma\right)$ *according to a partition* Π *of input* σ. *Let* $M(\delta)$ *be a submatrix of* CM *that consists only of rows* $CM_i, i \notin E_\Pi$, *such that* i *assigns values to all cosets from* $\mathcal{CC}_{G,K}\left(\Pi\right)$ *according to the*

string $\delta \in X^m$. For strings $\delta_1, \delta_2 \in X^m$ we claim the following.

$$\delta_1 \neq \delta_2 \Rightarrow M(\delta_1) \cap M(\delta_2) = \emptyset. \tag{6.16}$$

Let $\delta_1, \delta_2 \in X^m$ be two different strings defining assignments to the variables in $\mathcal{CC}_{G,K}(\Pi)$. Now let $i_1, i_2 \in X^{|\Pi_{L,\sigma}|}$ be row indices such that $CM_{i_1} \in M(\delta_1)$, and $CM_{i_2} \in M(\delta_2)$, clearly $i_1, i_2 \notin E_\Pi$. It is also clear that there is a column index $j \in X^{|\Pi_{R,\sigma}|}$ that assigns the elements of $\mathcal{CC}_{G,K}(\Pi)$ values defined by δ_1 and the rest of the values so that $\mathrm{HSP}_{G,K,X}(\Pi^{-1}(i_1, j)) = 1$. According to the definition of hidden subgroup function, $\delta_1 \neq \delta_2 \Rightarrow \mathrm{HSP}_{G,K,X}(\Pi^{-1}(i_2, j)) = 0$. Thus, $CM_{i_1} \neq CM_{i_2}$. **We proved the lemma.**

In the next lemma we count the number of different rows in a submatrix $M(\delta)$ for some $\delta \in X^m$.

Define a set of available assignments to the cosets in $\mathcal{CC}_{G,K}(\Pi)$.

$$\mathcal{W}_n^{l-m}(\Pi) = \big\{\{x_0, \dots, x_{n-1}\} | \forall i, j, 0 \leq i, j \leq n-1 \tag{6.17}$$

$$(x_i \in X \setminus \mathcal{CC}_{G,K}(\Pi) \land i \neq j \Rightarrow x_i \neq x_j)\big\}; \tag{6.18}$$

$$|\mathcal{W}_n^{l-m}(\Pi)| = \binom{l-n}{m}. \tag{6.19}$$

Lemma 14. *Let σ be the input in $X^{|G|}$. Let $\delta \in X^m$ be a string that defines an assignment for the variables in $\Pi_{L,\sigma}$. Let $M(\delta)$ be a submatrix of CM as defined in the lemma above. If $\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) > 0$ (note that it is not the same as $n > 0$) then for any two row indices $i_1, i_2 \in X^{|\Pi_{L,\sigma}|}$ such that they assign values from different sets from $\mathcal{W}_n^{l-m}(\Pi)$ to the cosets in $\mathcal{CC}_{G,K}(\Pi)$,*

$$CM_{i_1} \neq CM_{i_2}.$$

Let U, V be two different sets in $\mathcal{W}_n^{l-m}(\Pi)$. Let i_1, i_2 be two row indices that correspond to the rows in $M(\delta)$. Assume, i_1 assigns the cosets in $\mathcal{IC}_{G,K}(\Pi)$ values from U, and i_2 assign the cosets in $\mathcal{IC}_{G,K}(\Pi)$ values from U.

Since $U \neq V$ there is a value $x \in X \setminus \mathcal{CC}_{G,K}(\Pi)$, such that $x \in U$ and $x \notin V$. By assumption, $\mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) > 0$. Consider a column index j that assigns value x to one of its independent cosets, but $\mathrm{HSP}_{G,K,X}(\Pi^{-1}(i_2, j)) = 1$. Such an index j exisits, by the definition of $\mathrm{HSP}_{G,K,X}(\sigma)$. Also, by the definition of the hidden subgroup test function, it is clear that $\mathrm{HSP}_{G,K,X}(\Pi^{-1}(i_1, j)) = 0$. **This proves the lemma.**

Let's bring together the results of the two previous lemmas and estimate the number of unequal rows.

There are exactly $\binom{l}{m} m!$ different ways to choose assignments for the cosets in the set of *common cosets* $\mathcal{CC}_{G,K}(\Pi)$. According to the **Lemma 13**, rows, that have indices with different assignments of the values of the common cosets, never equal.

If $\mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) > 0$, then there are exactly $\binom{l-m}{n}$ ways to assign values to the independent cosets. According to the **Lemma 14**, rows with indices that assign to independent cosets values from different sets from $\mathcal{W}_n^{l-m}(\Pi)$ never equal.

That means there are at least

$$\binom{l}{m} m! \binom{l-m}{n} \tag{6.20}$$

unequal rows in the communication matrix CM. By the **Theorem 6**

$$\begin{aligned} CC_1(\mathrm{HSP}_{G,K,X}(\sigma), \Pi) = \Omega \Bigg(\log_2 \binom{l}{m} m! + \\ \left[\# \mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) > 0\right] \log_2 \binom{l-m}{n} \Bigg). \end{aligned} \tag{6.21}$$

Now substitute the values for l, m and n to obtain the statement of the theorem. □

In fact, our lower bound is actually tight. In other words, it coincides with the best algorithm that we can construct. That means we could not prove a statement about the lower bound of the communication complexity, according to the considered kind of partitions, any stronger than we already did.

Theorem 35. *Let K be a non-trivial subgroup of a finite group G. Let X be any finite set, such that $|X| \geq (G : K)$. For any partition Π, one-way communication*

complexity according to Π *of the hidden subgroup test function is bounded as follows.*

$$CC_1\left(HSP_{G,K,X}\left(\sigma\right),\Pi\right)=\Theta\left(\log_2\binom{|X|}{\#\mathcal{CC}_{G,K}\left(\Pi\right)}\#\mathcal{CC}_{G,K}\left(\Pi\right)!\right.$$
$$\left.+\left[\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}\left(\Pi_{R,\sigma}\right)>0\right]\log_2\binom{|X|-\#\mathcal{CC}_{G,K}\left(\Pi\right)}{\#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}\left(\Pi_{L,\sigma}\right)}\right).$$

Proof. The lower bound is already proved in the **Theorem 34**. In order to prove the upper bound we give an informal description of the protocol computing $\mathrm{HSP}_{G,K,X}\left(\sigma\right)$ according to the partition Π.

Let us use the same short-hand notation as in the **Theorem 34**.

The protocol is straightforward. There is just one round of communication. The computer A sends message c_1c_2, if $\mathcal{IC}_{G,K,\Pi_{L,\sigma}}\left(\Pi_{R,\sigma}\right)$ is positive, and sends only c_1 otherwise. The message parts $c_1\in\{0,1\}^a, c_2\in\{0,1\}^b$, where

$$a:=\log_2\binom{l}{m}m!, \tag{6.22}$$

$$b:=\log_2\binom{l-m}{n}. \tag{6.23}$$

First part c_1 of the message specifies the submatrix corresponding to the assignment of the common cosets values. Second part c_2 corresponds to the assignment of the independent cosets of $\Pi_{L,\sigma}$. The latter part allows the computer B choose values for its independent cosets of $\Pi_{R,\sigma}$ so that they do not coinside with the values of $\Pi_{L,\sigma}$. By the definition of the hidden subgroup function, information sent by A to B is enough to compute the function value for any assignment of $\Pi_{R,\sigma}$.

Note, that we can encode the message sent using a prefix code without loss of the efficiency (See page 41). □

We have considered communication complexity only according to a fixed partition so far. But our results hold for an arbitrary partition of the input. That is why, one-way communication complexity of the hidden subgroup test function is a direct consequence of the proven results.

Corollary 1. *Let K be a non-trivial subgroup of a finite group G. Let X be any finite set, sch that $|X| \geq (G:K)$. Let σ be the input. One-way communication complexity of the hidden subgroup test function is bounded as follows.*

$$CC_1\left(HSP_{G,K,X}(\sigma)\right) = \Theta\left(\min_{\Pi \in Bal(\sigma)} \left\{ \log_2 \binom{|X|}{\#\mathcal{CC}_{G,K}(\Pi)} \#\mathcal{CC}_{G,K}(\Pi)! \right.\right.$$
$$\left.\left. + \left[\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) > 0\right] \log_2 \binom{|X| - \#\mathcal{CC}_{G,K}(\Pi)}{\#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}(\Pi_{L,\sigma})} \right\}\right).$$

Proof. By the definition of one way communication complexity (See **Definition 2.4.14**, page 43), and as a consequence of **Theorem 35** the statement follows. □

The bounds, we have proved so far, hold for arbitrary partitions. However, in order to prove the "best" quantum lower bound, we need to find the "worst" partition for the hidden subgroup problem. Let's recall our tactics. Essentially, we prove a lower bound for a *classical* branching program, that we then use to obtain the quantum bounds. According to the classical branching program width definition, it is in our interest to find a cut that corresponds to the maximum width of the leveled branching program of our lower bound. What kind of cuts could be good candidates for this job? With this question in mind, we define a new kind of "balanced" partitions, designed specifically for a given instance of the hidden subgroup problem.

Definition 6.3.7. Let G be a finite group. Let K be a non-trivial proper subgroup of G. A partition Π of the input σ is called (G,K)-*coset balanced*, if

$$\#\mathcal{C}_{G,K}(\Pi_{L,\sigma}) = \lfloor (G:K)/2 \rfloor.$$

If Π is a cut, then we call it (G,K)-*coset balanced cut.*

It is not difficult to see that for any finite group G, and for any its non-trivial proper subgroup K, there exists a (G,K)-balanced cut. For this kind of cuts we state the next result.

Theorem 36. *Let K be a non-trivial proper subgroup of a finite group G. Let X be any finite set such that $|X| \geq (G:K)$. For any* (G,K)-coset balanced cut Π *of*

σ, one-way communication complexity of the hidden subgroup function is bounded as follows.

$$CC_1\left(HSP_{G,K,X}\left(\sigma\right),\Pi\right)=\Omega\left(\#\mathcal{CC}_{G,K}\left(\Pi\right)\log_2|X|+\#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}\left(\Pi_{L,\sigma}\right)\right).$$

Proof. We shall use the short-hand notation from the proof of **Theorem 34**.

The partition Π is (G,K)-balanced by assumption. It implies that

$$\begin{aligned}\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}\left(\Pi_{R,\sigma}\right)=&\\ &(G:K)-\#\mathcal{CC}_{G,K}\left(\Pi\right)-\#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}\left(\Pi_{L,\sigma}\right)=\\ &\qquad(G:K)-\#\mathcal{C}_{G,K}\left(\Pi_{L,\sigma}\right)\geq(G:K)/2>0.\end{aligned}\tag{6.24}$$

According to **Theorem 34**

$$CC_1\left(\mathrm{HSP}_{G,K,X}\left(\sigma\right),\Pi\right)=\Omega\left(\log_2\binom{l}{m}m!\binom{l-m}{n}\right).\tag{6.25}$$

We break the expression into parts and estimate the parts separately.

$$N_1:=\binom{l}{m}m!=\frac{l!}{(l-m)!}=l\cdot(l-1)\cdot\ldots\cdot(l-m+1)\geq\frac{l^m}{2^m},\tag{6.26}$$

because $m\leq\#\mathcal{C}_{G,K}\left(\Pi_{L,\sigma}\right)=\lfloor(G:K)/2\rfloor$.

$$N_2:=\binom{l-m}{n}=\frac{(l-m)!}{(l-m-n)!n!}.\tag{6.27}$$

Now we consider two cases:

1. Suppose $n=o(l)$. Since $m+n\leq(G:K)/2\leq l/2$, we may apply the Stirling's approximation:

$$N_2 \succeq \frac{(l-m)^{l-m}}{(l-m)^{l-m-n}e^n n!} = \frac{(l-m)^n}{e^n n!} \geq \frac{(l-m)^n}{e^n n^n} \geq \frac{l^n}{e^n n^n 2^n}, \tag{6.28}$$

because $m \leq l/2$. Thus, it follows

$$\log_2 N_1 N_2 \succeq m \log_2 l + n \log_2 l - m - n(\log_2 e + 1) - n \log_2 n \asymp \tag{6.29}$$

recall that $n = o(l)$

$$\asymp m \log_2 l + n \log_2 l \geq m \log_2 l + n. \tag{6.30}$$

2. Suppose that $l = O(n)$.

$$N_2 = \frac{(l-m-n+1) \cdot \ldots \cdot (l-m)}{n!} \succ \tag{6.31}$$

using Stirling's approximation for $n!$

$$\succ \frac{(l-m-n+1) \cdot \ldots \cdot (l-m)e^n}{n^n} \geq \frac{l^n e^n}{2^n n^n}. \tag{6.32}$$

note that $m + n \leq l/2$. Thus, it follows

$$\log_2 N_1 N_2 \succeq m \log_2 l + n \log_2 l/n + n(\log_2 e - 1) \geq \tag{6.33}$$

$$\geq m \log_2 l + n \log_2 e \asymp m \log_2 l + n. \tag{6.34}$$

The statement of the theorem follows after the substitution.

□

Finally, we obtain the quantum read-once branching program lower bound as a simple corollary.

Corollary 2. *Let K be a non-trivial subgroup of a finite group G. Let X be any finite set, such that $|X| \geq (G : K)$. Let $\epsilon \in (0, 1/2)$. If for all $\sigma \in \{0,1\}^n$ the function $HSP_{G,K,X}(\sigma)$ is ϵ-computed by a 1QBP Q then for any* coset-balanced *partition* Π *of the input σ*

$$width(Q) = \Omega\left(\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) \log_2 |X| + \#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}(\Pi_{L,\sigma})\right).$$

Proof. The statement follows as a corollary of **Theorem 33** and **Theorem 36**. □

A less precise, lower bound can also be shown. This time we make the statement in terms of the group order, or equally, the length of the input string $bin(\sigma)$ (See **Definition 6.3.1, page 122**). This bound shows that our upper bound (See **Theorem 32**, page 117) is quite tight. In fact, since the upper bound does not depend on the structure of G/K, the lower bound may simply reflect the complexity deviations for different choices of the parameters.

Corollary 3. *Let G be a finite group. Let K be be a non-trivial proper subgroup of G. Let X be a finite set, such that $|X| \geq (G : K)$. Let $t := (G : 1) \log_2 |X|$ be the length of the binary input $bin(\sigma)$. Let $\epsilon \in (0, 1/2)$. If for all $\sigma \in \{0,1\}^n$ the function $HSP_{G,K,X}(\sigma)$ is ϵ-computed by a 1QBP Q then for any* coset-balanced *partition* Π *of the input σ*

$$width(Q) =$$
$$CC_1(HSP_{G,K,X}(\sigma)) = \begin{cases} \Omega(|X|) = \Omega(t/log(t)) & if\ \mathcal{CC}_{G,K}(\Pi) = o((G:1)) \\ \Omega((G:K)\log_2 |X|) = \Omega(t) & otherwise \end{cases} \tag{6.35}$$

Proof. Consider the input $bin(\sigma)$ of the length g. See the $\mathrm{HSP}_{G,K,X}(\sigma)$ definition for details (Page 122). We want to establish an asymptotical lower bound on the quantum and communication complexity of $\mathrm{HSP}_{G,K,X}(\sigma)$ for $g \to \infty$.
The "hidden subgroup" K is a parameter in the function $\mathrm{HSP}_{G,K,X}(\sigma)$. As such, it has a cardinality $k := (K : 1)$, that we assume constant.
We shall again use the short-hand notation from the proof of **Theorem 34**.

From **Theorem 36** it follows that

$$CC_1\left(\mathrm{HSP}_{G,K,X}(\sigma), \Pi\right) = \tag{6.36}$$

$$\Omega\left(\#\mathcal{IC}_{G,K,\Pi_{L,\sigma}}(\Pi_{R,\sigma}) \log_2 |X| + \#\mathcal{IC}_{G,K,\Pi_{R,\sigma}}(\Pi_{L,\sigma})\right). \tag{6.37}$$

Consider two cases:

1. Suppose $\mathcal{CC}_{G,K}(\Pi) = m = o((G:1))$. Then by the definition of the coset-balanced partition (See **Definition 6.3.7**),

$$n + m = \Theta((G:1)) \Rightarrow n = \Theta((G:1)) \Rightarrow \tag{6.38}$$

$$CC_1\left(\mathrm{HSP}_{G,K,X}(\sigma), \Pi\right) = \Omega\left(o((G:1)) \log_2 l + (G:1)\right) = \Omega((G:1)). \tag{6.39}$$

   On the other hand

$$(G:1) = \Theta(|X|), \tag{6.40}$$

   thus,

$$t = \Theta((G:1)\log_2(G:1)) \Rightarrow \Omega((G:1)) = \Omega(t/\log_2 t). \tag{6.41}$$

   Finally,

$$CC_1\left(\mathrm{HSP}_{G,K,X}(\sigma), \Pi\right) = \Omega(t/\log_2 t). \tag{6.42}$$

2. Suppose $(G:1) = O(m)$. We directly obtain the desired result:

$$CC_1\left(\mathrm{HSP}_{G,K,X}(\sigma), \Pi\right) = \Omega((G:1)\log_2 l) = \Omega(t). \tag{6.43}$$

The generalization of the result for the width of the quantum branching program follows from **Corollary 2**. □

6.4 Sure states remark

There is an ongoing controversy about *quantum mechanics* and lately about *quantum computing*. We already mentioned Albert Einstein being maybe the most famous opponent of quantum mechanics. Emergence of quantum computing did not bring peace into the world. Contrary, it lead to further escalation of the conflict. The critique of *quantum computing* recently received a rigorous treatment by Scot Aaronson [Aar04a, Aar04b]. He addressed the question of unfeasibility of quantum computers by introducing *complexity measure* over *quantum states*. Let us shortly introduce here his approach.

Aaronson suggests to consider a separator set S that consists of all quantum states that have been demonstrated experimentally, "Sure states", but contains no states sufficient for non-trivial factoring, "Shor states".

Such a separator set would allow us discuss feasibility of a quantum algorithm based on rigorous ground. However, what should one choose as a "Sure states" set? There are demonstrated states with very small amplitudes, as well as states that involve entanglement across of hundreds of thousands of particles. Aaronson proposed a very good candidate for the separator set. He called it the set of *"Tree states"*.

Definition 6.4.1 ([Aar04b]). A *quantum state tree* over $\mathcal{H}_2^{\otimes n}$ is a rooted tree where each leaf vertex is labeled with $\alpha|0\rangle + \beta|1\rangle$ for some $\alpha, \beta \in \mathbb{C}$, and each non-leaf vertex (called a *gate*) is labeled with either $+$ or $\otimes$. Each vertex v is also labeled with a set $S(v) \subseteq \{1, \ldots, n\}$, such that

1. If v is a leaf then $|S(v)| = 1$,
2. If v is the root then $S(v) = \{1, \ldots, n\}$,
3. If v is a $+$ gate and w is a child of v then $S(v) = S(w)$,
4. If v is a $\otimes$ gate, and $w_1, \ldots, w_k$ are the children of v, then $S(w_1), \ldots, S(w_k)$ are pairwise disjoint and form a partition of $S(v)$.

Finally, if v is a $+$ gate then the outgoing edges of v are labeled with complex numbers. For each v, the subtree rooted at v represents a quantum state of the qubits in $S(v)$

in the obvious way. We require this state to be normalized for each v.
The tree is called *orthogonal* if it satisfy the further condition that if v is a $+$ gate, then any two children w_1, w_2 of v represent states $|\psi_1\rangle, |\psi_2\rangle$ with inner product $\langle\psi_1|\psi_2\rangle = 0$. If the condition $\langle\psi_1|\psi_2\rangle = 0$ can be replaced by a stronger condition that for all basis states $|x\rangle$, either $\langle\psi_1|x\rangle = 0$ or $\langle\psi_2|x\rangle = 0$, then we sat the tree is *manifestly orthogonal*.

The quantum state tree provides a natural complexity measure over the quantum states.

Definition 6.4.2 ([Aar04b]). We define the *size* (T) size of the tree T to be the number of leaf vertices. Then given a state $|\psi\rangle \in \mathcal{H}_2^{\otimes n}$, the *tree size* $\mathrm{TS}(|\psi\rangle)$ is the minimum size of a tree that represents $|\psi\rangle$. The *orthogonal tree size* $\mathrm{OTS}(|\psi\rangle)$ and *manifestly orthogonal tree size* $\mathrm{MOTS}(|\psi\rangle)$ are defined similarly.

With the state tree concept in mind, following quantum state *complexity classes* can be defined.

Definition 6.4.3 ([Aar04b]). Define the *quantum state complexity classes*:

Classical is the set of the *classical* basis sets of the form $|x\rangle$ for some $x \in \{0,1\}^n$;

$\otimes_{\mathbf{1}}$ is the set of *separable states* of the form $(\alpha_1|0\rangle + \beta_1|1\rangle) \otimes \cdots \otimes (\alpha_n|0\rangle + \beta_n|1\rangle)$;

$\mathbf{\Sigma_1}$ is the state of all states that are *superpositions* of at most $p(n)$ *classical* states, where p is some plynomial;

$\otimes_{\mathbf{i}}$ contains the states that can be written as a tensor product of $\mathbf{\Sigma_{i-1}}$ states, with qubits permuted arbitrarily, where $i > 1$;

$\mathbf{\Sigma_i}$ contains the states that can be written as linear combinations of a polynomial number of $\otimes_{\mathbf{i-1}}$ states, where $i > 1$;

TSH is the *tensor sates hierarchy* class is defined as $\mathbf{TSH} := \cup_k \mathbf{\Sigma_k} = \cup_k \otimes_{\mathbf{k}}$.

Tree is the class of all states expressible by a polynomial-size tree of additions and tensor products nested arbitrarily.

OTree is the class of all states $|\psi_n\rangle \in \mathcal{H}_2^{\otimes n}$ such that $\mathrm{OTS}(\psi_n) = p(n)$, for some polynomial $p(n)$.

MOTree is the class of all states $|\psi_n\rangle \in \mathcal{H}_2^{\otimes n}$ such that $\mathrm{MOTS}(\psi_n) = p(n)$, for some polynomial $p(n)$.

We would like to analyze the states involved in our upper bound proofs. The basic subroutine used by all of the upper bound algorithms is that computing *Equality*. We consider the upper bound algorithm for Equality. Conclusions concerning the rest of the algorithms will follow. The more elaborate algorithms include tensor products of not more than linear (over the input size) number of the states used to compute *Equality*.

A state $|\psi\rangle$ in the *state space* $\mathcal{H}_C$ of the Equality algorithm (See **Thm. 25, p. 99**) is described as

$$|\psi\rangle = \Sigma_{i=0}^{t} \alpha_i |i\rangle, \tag{6.44}$$

where

$$|i\rangle \in \mathcal{H}_2^{\oplus \log_2 t}. \tag{6.45}$$

The expression above is a linear combination of *separable states*, that is, states from $\otimes_{\mathbf{1}}$. As such, $|\psi\rangle$ is in $\mathbf{\Sigma_2}$. Although, already $\mathbf{\Sigma_2} \not\subseteq$ **MOTree** [Aar04b]. Note, that in more complex algorithms presented here, the states may even belong to $\otimes_{\mathbf{3}}$. It is also clear, that whole *tensor states hierarchy* is contained in **Tree**.

Nevertheless, the states involved in our algorithms are *Sure states*. This is remarkable, because we, in fact, solve a decision version of the problem that is more general than those considered by P. Shor. However, we don't use Shor states!

Chapter 7

Reducibility Theory

> Somewhere, something incredible is waiting to be known.
>
> Blaise Pascal

7.1 Introduction

We say that a problem A can be *reduced* to a problem B if solving B, we can also solve A. An example of reduction is the so-called *polynomial time many to one reduction*, also known as *Karp reduction*.

Definition 7.1.1. [BDG88] Given two sets A_1 and A_2, we say that A_1 is *polynomial time many-one reducible* to A_2 if and only if there exists a function $f : \Sigma^* \to \Sigma^*$, computable in polynomial time, and such that $x \in A_1$ if and only if $f(x) \in \Sigma^*$. In this case we write $A_1 \leq_m A_2$.

This reduction can be used to define already mentioned *class of* **NP***-complete problemes.*

Definition 7.1.2. Class **NPC** consists of all problems $A_2 \in \mathbf{NP}$, such that, for all $A_1 \in \mathbf{NP}$ $A_1 \geq_m A_2$.

There are many more reductions defined for different purposes. In the next section we present one that suits the best to our case. That is, it can be used to define the class of problems that can be efficiently solved by application of the methods presented in the two previous chapters.

7.2 Rectangular reduction

An analog of the Karp's reduction in the communication complexity is called *rectangular reduction*.

Definition 7.2.1 ([Weg00]). Let $f(x, y) \in \mathbb{B}_{n+m}$ and $g(x, y) \in \mathbb{B}_{k+l}$. A pair (ϕ_A, ϕ_B) of functions $\phi_A : \{0, 1\}^n \to \{0, 1\}^k$ and $\phi_B : \{0, 1\}^m \to \{0, 1\}^l$ is called a *rectangular reduction* from f to g if $f(a, b) = g(\phi_A(a), \phi_B(b))$ for all $(a, b) \in \{0, 1\}^n \times \{0, 1\}^m$. We write $f \leq_\square g$ if f is reduced to g.

The reduction is called rectangular because if $f \leq_\square g$, the monochromatic rectangles of *communication matrix* of f are mapped to those of g. See **Definition 2.4.16** on the page 44 for *communication matrix*.
According to the definition of communication complexity (**Definition 2.4.7**, p. 38), only the communicated messages are counted. The rest computations that the communicating parties may do on their own are "free of charge" (See page 33 for the discussion). Thus, each of the party can compute corresponding function $\phi_A(a)$ and $\phi_B(b)$. Having that done, they can proceed communication according to the protocol for g! We just proved a proposition.

Proposition 7.2.1. *If for a partition Π_X of the variables set $X = \{x_1, \ldots, x_n\}$ and for a partition Π_Y of the variables set $Y = \{y_1, \ldots, y_m\}$ $f(X) \leq_\square g(Y)$ then*

$$CC(f, \Pi_X) \leq CC(g, \Pi_Y);$$
$$CC_1(f, \Pi_X) \leq CC_1(g, \Pi_Y).$$

Next statement follows from the **Theorem 33** (See page 124) and the proposition above.

Corollary 4. *Let $\epsilon \in (0, 1/2)$ be a constant. Let Q_f be a one-way quantum branching program that $(1/2+\epsilon)$-computes (computes with the margin ϵ) function $f_n \in \mathbb{B}_n$. Let g be a function in $\mathbb{B}_n$.*
If for a partition Π_X of the variables set $X = \{x_1, \ldots, x_n\}$ and for a partition Π_Y of the variables set $Y = \{y_1, \ldots, y_m\}$ $f(X) \leq_{\Box} g(Y)$ then

$$width\,(Q_f) = \Omega(CC_1\,(g, \Pi_Y)).$$

Thus, we show that the *rectangular reductions* provide a good tool to generalize results of this thesis to more computational problems. An interesting alternative reduction concept is presented in the next section.

7.3 Polynomial projections

This type of reduction was intensively studied by Skyum and Valiant [SV85]. It was carefully investigated by Billig and Wegener [BW96] in the formal circuit verification context. The most basic *polynomial projection* is defined below.

Definition 7.3.1 ([BW96]). The sequence of functions $f = (f_n)$ is a *(polynomial) projection* of $g = (g_n)$, $f \leq_{proj} g$, if

$$f_n(x_1, \ldots x_n) = g_{p(n)}(y_1, \ldots, y_{p(n)})$$

for some polynomially bounded function p and $y_j \in \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}, 0, 1\}$. For functions with many outputs, each output of f_n has to equal one output of $g_{p(n)}$. The number of j such that $y_j \in \{x_i, \overline{x_i}\}$ is called *multiplicity* of x_i. The projection is *monotone*, $f \leq_{mp} g$, if $y_j \in \{x_1, \ldots, x_n, 0, 1\}$.

It turns out that polynomial projections is a reduction concept too general for deterministic branching programs. Next theorem shows that $f \leq_{proj} g$ doesn't imply existence of a polynomial-size program for f even if there's one for g!

Theorem 37 ([BW96]). *1. There exist functions f and g, where $f \leq_{proj} g$ for projections of multiplicity two, g has OBDDs of linear size but f has only FBDDs and k-OBDDs (for constant k) of exponential size.*

2. There exist functions f^* and g^*, where $f^* \leq_{proj} g^*$, g^* has OBDDs of linear size but f^* has for constant k only k-IBDD of exponential size.

This theorem shows that general projections can not yield any sensible complexity theory for OBDDs, k-OBDDs, k-IBDDs or FBDDs. Already for multiplicity two a projection may have exponentially larger OBDD (FBDD, k-OBDD) than the original function! Therefore, we define *read-once* projections.

Definition 7.3.2 ([BW96]). A projection is called *read-once*, $f \leq_{rop} g$, if the multiplicity of each variable is bounded by one. Additionally, $\leq_{mrop}$ denotes *monotone read-once* projections.

Read-once projection turns out to be an appropriate reduction concept for deterministic branching programs.

Theorem 38 ([BW96]). *1. Read-once projection $\leq_{rop}$ is reflexive and transitive.*

2. If $f \leq_{rop} g$ and g has polynomial OBDD (FBDD, k-OBDD, k-IBDD) size, then f has polynomial OBDD (FBDD, k-OBDD, k-IBDD) size.

3. There exist functions f and g such that $f \leq_{mrop} g$, g has polynomial OFDD size and f has exponential FFDD size.

This reduction has very nice properties, indeed, suitable to describe the OBDD complexity theory. However, consider the functions studied in this thesis.

1. Equality;
2. Periodicity;
3. Semi-Simon;
4. Hidden subgroup problem test;

5. Simon test;

Observe that for any function $f \in \mathbb{B}_n$ from the list above it holds that

$$f \leq_{2mp} (\mathrm{EQ}_n(x_1, y_1))^{\sigma_1} \wedge (\mathrm{EQ}_n(x_2, y_2))^{\sigma_2} \wedge \cdots \wedge (\mathrm{EQ}_n(x_k, y_k))^{\sigma_k}$$

where $\leq_{2mp}$ is a polynomial monotone projection of the multiplicity 2, $\sigma_i \in \{0, 1\}$, $x_i, y_i \in \{0, 1\}^s$, $k, s\mathbb{N}, ks \geq n$, and

$$(\mathrm{EQ}_n(x_i, y_i))^0 = \mathrm{EQ}_n(x_i, y_i), \text{ and } (\mathrm{EQ}_n(x_i, y_i))^1 = \neg\mathrm{EQ}_n(x_i, y_i).$$

The next statement follows from the construction we undertake in the proof of the **Theorem 32** (See page 117).

Proposition 7.3.1. *If a function $f \in \mathbb{B}_n$ is a* monotone polynomial projection of multiplicity 2 *of*

$$(EQ_n(x_1, y_1))^{\sigma_1} \wedge (EQ_n(x_2, y_2))^{\sigma_2} \wedge \cdots \wedge (EQ_n(x_k, y_k))^{\sigma_k},$$

where $\sigma_i \in \{0, 1\}$, $x_i, y_i \in \{0, 1\}^s$, $k, s\mathbb{N}, ks \geq n$. Then f can be computed with two-sided error $o(1)$ by a 1QBP of linear over n width.

The proof of the **Theorem 37** [BW96] shows that for multiplicity 2 there are functions f^* and $g*$, $f^* \leq_{proj} g*$ so that deterministic OBDD complexity of $f*$ is exponentially larger then that of $g*$. This is not surprising, since we know that deterministic and quantum branching programs are incomparable (See page 89).
On the other hand, it is not obvious whether all the functions listed above can be *read-once* polynomial projections of the same function. This is one of the open questions that can be considered in further research.
There are many more question that this research project left open. In this chapter we gave but just few tools to generalize results presented in this thesis. This chapter may also suggest some new methods, or insights that could lead to a new research.

Chapter 8

Conclusion

In this research we investigated complexity of the hidden subgroup problem in the context of quantum branching programs. We started with a very basic though fundamental *Equality* function. We proved non-trivial linear quantum oblivious read-once branching programs upper and lower bounds for this function. We then generalized the technique to prove similar bounds for *Periodicity*, and the simplified version of the Simon problem, which we called *Semi-Simon*.

The central objective of the research was achieved when we were able to prove linear upper bounds for the *hidden subgroup test*. Remarkably, our algorithm does not need the group to be *Abelian*. Thus, it also generalizes the famous *graph isomorphism* problem. It is also, to our knowledge, the first quantum algorithm that tackles the *hidden subgroup problem* by means different from *Fourier sampling* or *Eigenvalue estimation*.

When proving the quantum OBDD lower bounds for the function, as a byproduct, we obtained additionally the *one-way communication complexity* lower bounds. We also proved that the communication bounds are tight!

The upper bounds algorithms of this thesis use only so-called *Tree* states that have been demonstrated experimentally. The programs don't require creation of more elaborate quantum states required for the Shor's algorithm that raise controversy over feasibility of such algorithms.

Finally, our research provides various opportunities to generalize obtained results.

We can briefly list some further directions of investigation:

- Consider the test of integer multiplication. This problem is as hard as factoring integers for deterministic branching programs. An efficient randomized algorithm is known due to Ablayev and Karpinski [AK98].
- Define a *necessary* condition for a function to have a quantum OBDD of linear size. We provide several reduction concepts that can be used as sufficient conditions.
- A connection between the Fourier transform and fingerprinting can be investigated, as both those techniques can be used to tackle hidden subgroup problem related algorithms. The evidences are provided by the Shor's algorithm and this research.
- Lower bounds for the randomized OBDD of the problems considered in this thesis can be proved. For the *Equality* function, an upper bound $n^2 \log_2 n$ on the randomized OBDD width can be found in the book of Wegener [Weg00].
- The reduction concepts we presented can be applied to prove upper and lower bounds for new problems that are not considered in this work.

The problems we studied had not been considered in the branching programs setting prior to this research. We apply different techniques to create our own. The approach we take to prove lower bounds of the *hidden subgroup test*, establishes direct connection between the width of *quantum* OBDD and *deterministic* one-way communication complexity. This can be further used to simplify proofs of the quantum lower bounds. Ultimately, this research was about making sense of why quantum algorithms are so powerful when it comes to solving *Hidden Subgroup* related problems (see e.g. [ME99] or [NC00] for details). Famous instances of those problems are *factoring* and *descrete logarithm*. They were effectively solved by P.Shor [Sho97]. Our starting point was to translate Shor's technique into language of *Branching Programs*. The functions we define for *Periodicity* and *Simon* were taken as other well-known *Hidden Subgroup* problem instances. Finally, solution boiled down to *Equality* computation. In turn,

the latter function can be considered as a special case of the artificial function f_n defined in [AK96]. There was already some research done for both f_n and *Equality* [AF98, AK97, AK96, AGK01]. Modifying existed *Branching Program* and *One-Way Finite Automata* techniques we obtained what may be the translation of Shor's technique. There is still a number of open questions. That means a lot of research for physicists, mathematicians and computer scientists to be done. Hopefully, this work will be of some help for them.

Appendix A

The list of notation

The most frequently used notation presented in the list below. Every notation instance that has been defined in the text is provided a references. Otherwise, informal in-place definition is presented.

Table A.1: List of most frequently used notation

$\wedge$	*Conjunction*, logical *AND*;
$\vee$	*Disjunction*, logical *OR*;
$\neg$	*Negation*, logical *NOT*;
$\oplus$	*Exclusive OR*, or *bitwise addition modulo* 2;
$\otimes$	*Tensor product;*
$f = O(g)$	Means there is a constant c such that $\lvert f(x)\rvert \leq c\lvert g(x)\rvert$ for all x;
$f = \Omega(g)$	is the same as g=O(f);
$f = \Theta(g)$	$f = O(g)$ and $g = O(f)$;
$f = o(g)$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$;
$f \asymp g$	$f = \Theta(g)$;
$f \prec g$	$f = O(g)$;
$f \succ g$	$f = \Omega(g)$;
$\mathbb{B}_n$	The set of Boolean functions (p. 27, **Definition 2.3.1**) over n variables;

$\mathcal{H}^n$	The same as $\mathcal{H}_2^{\otimes \log_2 n}$ the n-dimensional Hilbert space;
$\#A$	The cardinality of the set A, the same as $\|A\|$;
G/K	The factor group of G by K;
$(G : K)$	The *left index* of the subgroup K on the group G, equals the number of *left* cosets of K in the group G;
$(G : 1)$	The *order* of the group G;
$\Pr(X)$	This is how we denote probability of X;
$\mathrm{Proj}_F(\|\psi\rangle)$	Projection of the vector $\|\psi\rangle$ onto space F.
$[n]$	{1,. . . , n};
$\lfloor x \rfloor$	the greatest integer less or equal to x;
$\lceil x \rceil$	the least integer larger or equal to x;
λ	empty word, p. 7, **Definition 1.2.1**
B^0	$\{\lambda\}$ p. 7, **Definition 1.2.6**;
B^+	p. 7, **Definition 1.2.6**;
B^*	$B^+ \cup \{\lambda\}$ p. 7, **Definition 1.2.6**;
$\|\psi\rangle$	Ket-vector (column vector), p. 69;
$\langle\psi\|$	Bra-vector (row vector, dual to $\|\psi\rangle$), p. 69;
$bin\,(\sigma)$	Binary representation of σ;
$width\,(P)$	The width of the branching program P, p. 29, **Definition 2.3.5**;
$\Pi_{L,X}$	Left part of the partition Π over X, p. 34, **Definition 2.4.1**;
$\Pi_{R,X}$	Right part of the partition Π over X, p. 34, **Definition 2.4.1**;
$Bal\,(X)$	Set of all balanced partitions of X, p. 39, **Definition 2.4.9**;
$Abal\,(X)$	Set of all almost balanced partitions of X, p. 39, **Definition 2.4.9**;
$\overline{0}$,$\overline{1}$	Just some extra symbols we occasionally use, see for example p. 36;

$CM(f, \Pi)$ Full notation for communication matrix of the function f defined according to the partition Π, p. 44, **Definition 2.4.16**;

CM Short notation for communication matrix adopted where it does not cause confusions, see proof of the **Theorem 34**, p. 126;

$CC_1(f)$ Deterministic one-way communication complexity of the function f according to a fixed partition Π, p. 43, **Definition 2.4.15**;

$CC(f)$ Deterministic many-round communication complexity of the function f according to a fixed partition Π, p. 39, **Definition 2.4.10**;

1QBP *oblivious* Read-once Quantum Branching Program, p. 81, **Definition 4.3.3**;

$\mathrm{EQ}_{nx,y}(x, y)$ Equality function notation, p. 94, **Definition 5.1.1**;

$\mathrm{Period}_{s,n}(\sigma)$ Periodicity function notation, p. 94, **Definition 5.1.2**;

$\text{Semi-Simon}_{s,n}(\sigma)$ Semi-Simon function notation, p. 94, **Definition 5.1.3**;

$\mathrm{Simon}_{s,n}(\sigma)$ Simon function notation, p. 115, **Definition 6.1.2**;

$\mathrm{HSP}_{G,K,X}(\sigma)$ Hidden Subgroup function notation, p. 114, **Definition 6.1.1**;

$\mathcal{C}_{G,K}(A)$ Number of different cosets members in the set A, p. 122, **Definition 6.3.2**;

$\mathcal{CC}_{G,K}(A, B)$ Common cosets number of the sets A and B, p. 123, **Definition 6.3.3**;

$\mathcal{IC}_{G,K,A}(B)$ Independent cosets number of the set B with respect to the set A, p. 123, **Definition 6.3.4**;

$NRow(M)$ Number of unequal rows in the matrix M, see p. p. 45, **Theorem 6**;

$\leq_{\square}$ Rectangular reduction, p. 140, **Definition 7.2.1**;

$\leq_{proj}$ Polynomial projection, p. 141, **Definition 7.3.1**;

$\leq_{rop}$ Polynomial read-once projection, p. 142, **Definition 7.3.2**;

$\leq_{mrop}$ Polynomial monotone read-once projection, p. 142, **Definition 7.3.2**;

Appendix B

Additional material

B.1 On Chernoff bound

We dedicate this chapter to the Chernoff bound. A theorem from the theory of probability that is widely used in computer science.

Proposition B.1.1 ([Pap94]). *Suppose that* $x_1, x_2, \ldots, x_n$ *are independent random variables taking the values* 1 *and* 0 *with probabilities* p *and* $1-p$*, respectively, and consider their sum* $X = \Sigma_{i=1}^{n} x_i$*. Then for all* $0 \leq \Theta \leq 1$*, probability* $\Pr[X \geq (1+\Theta)pn] \leq e^{-\frac{\Theta^2}{3}pn}$.

It is easy to obtain a simple corollary of this statement.

Corollary 5. *Let* $x_1, x_2, \ldots, x_n$ *are independent random variables taking the values* 1 *and* 0 *with probabilities* $1/2 + \epsilon$ *and* $1/2 - \epsilon$ *respectively. Then the following holds.*

$$\Pr[\Sigma_{i=1}^{n} x_i \leq n/2] \leq e^{\epsilon n}$$

Proof. We use $\overline{x}$ denote the negation of x.

$$\Sigma_{i=1}^{n} x_i \leq n/2 \iff \Sigma_{i=1}^{n} \overline{x}_i \geq n/2.$$

Thus

$$\Pr[\Sigma_{i=1}^{n} x_i \leq n/2] = \Pr[\Sigma_{i=1}^{n} \overline{x}_i \geq n/2] \leq$$

by the Chernoff bound, where $p(1+\Theta) = 1/2$ and $p = 1/2 - \epsilon$

$$\leq e^{-\frac{(\epsilon)^2(1/2-\epsilon)n}{(1/2-\epsilon)^2}} =$$

the latter due to $(1/2-\epsilon)(1+\Theta) = 1/2 \Rightarrow \Theta = \frac{\epsilon}{1/2-\epsilon}$. Finally

$$= e^{-\frac{\epsilon^2 n}{1/2-\epsilon}} \leq e^{-\epsilon^2 n}.$$

That completes the proof. □

B.2 Complexity classes

In **Figure B.1** a graphical representation of the relations between some basic complexity classes is shown. The representation of the complexity classes hierarchy is due to José L. Balcázar, Josep Díaz and Joaquim Gabarró [BDG88].

B.3 NP-Intermediate problems

If the two classes **P** and **NP** are indeed unequal, then there exist problems that neither **NP**−*complete* nor they are in **P** [GJ79]. We call al those problems **NP**-*Intermediate*, and denote the class of such problems as **NPI**.
Since $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ was an open question to the moment of writing of this thesis, it is not clear whether **NPI** even exists. However, problems that could be good candidates to represent this class are long since known. One of them is called *graph isomorphism*.

Example 17 (GRAPH ISOMORPHISM, [GJ79]).

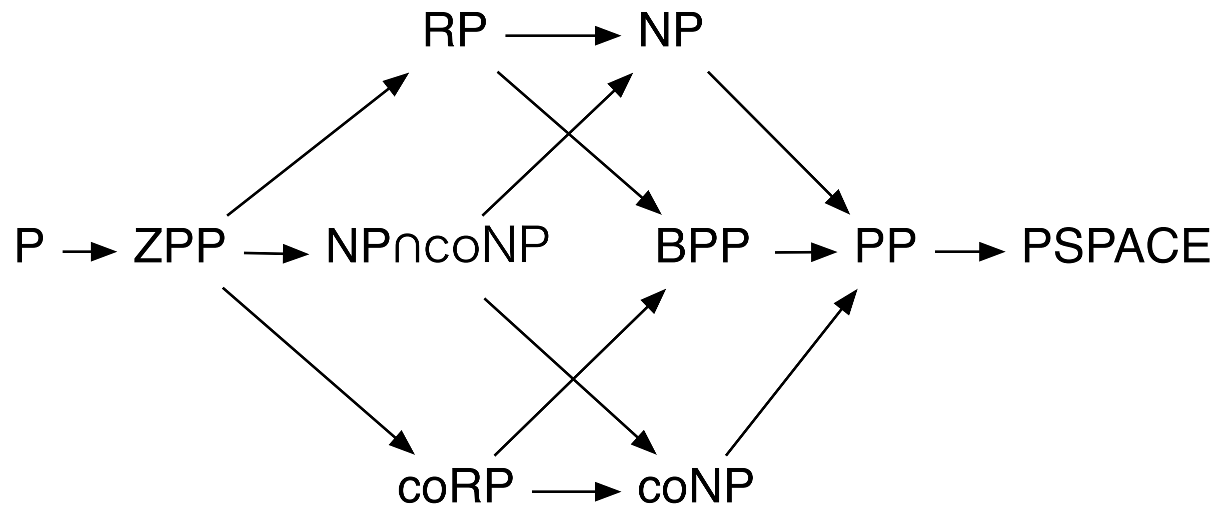

Figure B.1: Plethora of complexity classes.

Instance: Graphs $G = (V, E)$, $G' = (V', E')$.

Question: Are G and G' isomorphic, that is, is there a one-to-one function $f : V \to V'$ such that $\{u, v\} \in E$ if and only if $\{f(u), f(v)\} \in E'$?

Another long standing candidate for **NPI**, the problem of deciding whether a given number is composite was dismissed in 2002. It was discovered that $PRIMES \in \mathbf{P}$ [AKS02].

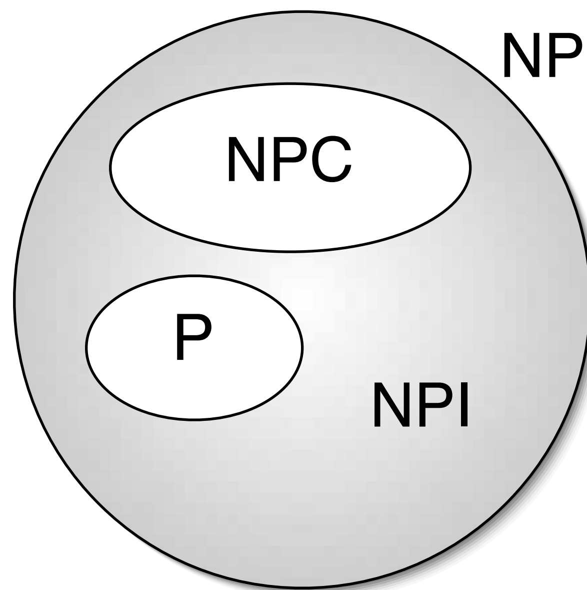

Figure B.2: Class NP-Intermediate

Bibliography

[Aar] Scott Aaronson, *The complexity zoo*, http://www.complexityzoo.com/.

[Aar04a] ______, *Limits on efficient computation in the physical world*, Ph. D. Thesis, University of California, Berkeley, 2004.

[Aar04b] ______, *Multilinear formulas and skepticism of quantum computing*, arXive e-print quant-ph/0311039 (2004).

[Abl05] Farid Ablayev, *On one-way quantum communication complexity and its applications for quantum OBDDs*, Manuscript, personal communication (2005).

[AF98] A. Ambainis and R. Freivalds, *1-way quantum finite automata: strengths, weaknesses and generalization*, Proceeding of the 39th IEEE Conference on Foundation of Computer Science, 1998, See also arXiv:quant-ph/9802062 v3, pp. 332–342.

[AGK01] F. Ablayev, A. Gainutdinova, and M. Karpinski, *On computational power of quantum branching programs*, Lecture Notes in Computer Science, no. 2138, Springer-Verlag, 2001, See also arXiv:quant-ph/0302022 v1, pp. 59–70.

[AGK+05] F. Ablayev, A. Gainutdinova, M. Karpinski, C. Moore, and C. Pollette, *On the computational power of probabilistic and quantum branching programs*, To appear in Information and Computation (2005).

[AK96] F. Ablayev and M. Karpinski, *On the power of randomized branching programs*, Proceedings of the ICALP'96, Lecture Notes in Computer Science, no. 1099, Springer-Verlag, 1996, pp. 348–356.

[AK97] ———, *On the power of randomized ordered branching programs*, Tech. Report 85181-CS, University of Bonn, 1997, see also Electronic Colloquium on Computational Complexity, TR98-004, (1998), available at http://www.eccc.uni-trier.de/eccc/.

[AK98] ———, *A lower bound for integer multiplication on randomized read-once branching programs*, ECCC (1998), no. 11.

[AKS02] M. Agrawal, N. Kayal, and N. Saxena, *PRIMES is in p*, http://www.cse.iitk.ac.in/news/primality.html, 2002.

[AMP02] F. Ablayev, C. Moore, and C. Pollett, *Quantum and stochastic branching programs of bounded width branching progrmas of finite width*, Proc. 29th Intl.Colloquium on Automata, Languages and Programming (Malaga), Springer-Verlag, 2002, pp. 343–354.

[Bar85] D. A. Barrington, *Bounded-width polynomial-size branching programs recognize exactly those languages in* NC^1, Proceedings of the eighteenth annual ACM symposium on Theory of computing (Berkeley, California, United States), Annual ACM Symposium on Theory of Computing, ACM Press, 1985, pp. 1–5.

[BDG88] José L. Balcázar, Josep Díaz, and Joaquim Gabarró, *Structural complexity I*, Texts in theoretical computer science. An EATCS series, Springer-Verlag, 1988.

[BDG90] ———, *Structural complexity II*, Texts in theoretical computer science. An EATCS series, Springer-Verlag, 1990.

[BJ25] M. Born and P. Jordan, Zeits. f. Phys. **34** (1925), 858.

[Bor24] M. Born, Zeits. f. Phys. (1924), no. 26, 379.

[Bor26] ———, Zeits. f. Phys. **38** (1926), 803.

[Bry86] R. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Trans. Comput. (1986), no. C-35, (8), 677–691.

[BT88] D. A. M. Barrington and D. Thérien, *Finite monoids and the fine structure of NC^1*, Journal of the ACM **35** (1988), no. 4, 941–952.

[BV97] Ethan Bernstein and Umesh Vazirani, *Quantum complexity theory*, SIAM Journal on Computing **26** (1997), no. 5, 1411–1473.

[BW96] Beate Bollig and Ingo Wegener, *Read-once projections and formal circuit verification with binary decision diagrams*, STACS 1996, 1996, pp. 491–502.

[Chu36] A. Church, *An unsolvable problem of elementary number theory*, American Journal of Mathematics (1936), no. 58, 345–363.

[Coo71] S. A. Cook, *The complexity of theorem-proving procedures*, Proc. 3rd Ann. ACM Symp. on Theory of Computing (New York), Association for Computing Machinery, 1971, pp. 151–158.

[Dir25] P. A. M. Dirac, Proc. Roy. Soc. **A109** (1925), 642.

[Dir30] ———, *Principles of quantum mechanics*, 4th ed., Oxford University Press, New York, 1958 (1st ed. 1930).

[EPR35] A. Einstein, B. Podolsky, and N. Rosen, Phys. Rev. **47** (1935), 777.

[fM] Clay Institute for Mathematics, *Millenium problems*, http://www.claymath.org/millennium/.

[FMSS03] Katalin Friedl, Frédéric Magniez, Miklos Santha, and Pranab Sen, *Quantum testers for hidden subgroup properties*, LNCS 2747, MFCS 2003, Springer-Verlag, 2003, pp. 419–428.

[FR74] M. J. Fischer and M. O. Rabin, *Super-exponential complexity of Presburger arithmetic*, Complexity of computation (R. M. Karp, ed.), American Mathematical Society, Providence, RI, 1974, pp. 27–41.

[Fre79] R. Freivalds, *Fast probabilistic algorithms*, FCT'79, LNCS 74 (Berlin, New York), Springer-Verlag, 1979, pp. 57–69.

[Gil72] J. Gill, *Probabilistic turing machines and complexity of computations*, Ph. D. Dissertation, U. C. Berkeley, 1972.

[Gil77] J. Gill, *Computational complexity of probabilistic turing machines*, SIAM Journal on Computing **6** (1977), no. 4, 675–695.

[GJ79] Michael R. Garey and David S. Johnson, *Computers and intractability*, A series of books in the mathematical sciences, W. H. Freeman and Co., New York, 1979.

[Gro97] L. K. Grover, *Quantum mechanics helps in searching for a needle in a haystack*, Phys. Rev. Lett. (1997), no. 79, 325–328.

[GSVV01] Michelangelo Grigni, Leonard Schulman, Monica Vazirani, and Umesh Vazirani, *Quantum mechanical algorithms for the nonabelian hidden subgroup problem*, Proceedings of the thirty-third annual ACM symposium on Theory of computing (Hersonissos, Greece), ACM Press, 2001, pp. 68–74.

[Ham80] Richard W. Hamming, *Coding and information theory*, Prentice Hall, Englewood Cliffs, N.J., 1980.

[Har78] Michael A. Harrison, *Introduction to formal language theory*, Addison-Wesley Series in Computer Science, Addison-Wesley, 1978.

[Haw88] Stephen E. Hawking, *A brief history of time from the big bang to the black holes*, Bantam Books, London, 1988.

[Hei25] W. Heisenberg, Zeits. f. Phys. **33** (1925), 879.

[Hei27] ———, Zeits. f. Phys. **43** (1927), 172.

[Høy97] Peter Høyer, *Conjugated operators in quantum algorithms*, Tech. report, University of Southern Denmar, 1997.

[Hro97] Juraj Hromkovič, *Communication complexity and parallel computing*, EATCS Texts in Theoretical Computer Science, Springer-Verlag, Berlin, 1997.

[HS65] J. Hartmanis and R. E. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc. (1965), no. 117, 285–306.

[HW79] G. H. Hardy and E. M. Wright, *An introduction to the theory of numbers*, 5th ed. ed., Oxford, England: Clarendon Press, 1979.

[HW02] Peter Høyer and Ronald de Wolf, *Improved quantum communication complexity bounds for disjointness and equality*, Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science, STACS'2002 (Antibes-Juan les Pins, France, March 14-16, 2002) (Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Tokyo) (Helmut Alt and Afonso Ferreira, eds.), LNCS, vol. 2285, Springer-Verlag, 2002, pp. 299–310.

[Joz97] Richard Jozsa, *Quantum algorithms and the fourier transform*, arXiv:quant-ph/9707033 (1997).

[Juk89] S. Jukna, *On the effect of null-chains on the complexity of constant schemes*, Proc. FCT'89, Lecture Notes in Computer Science, vol. 380, Springer-Verlag, 1989, pp. 246–256.

[Kar72] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of computer computation (R. E. Miller and J. W. Thatcher, eds.), Plenum Press, New York, 1972, pp. 85–103.

[KM99] M. Karpinski and R. Mubarakzjanov, *A note on Las Vegas OBDDs*, ECCC (1999), no. 99-09.

[KMW88] M. Krause, C. Meinel, and S. Waack, *Separating the eraser turing machine classes L_e, NL_e, $co - NL_e$, and P_e*, Proceedings MFCS'88 (Carlsbad, Czechoslovakia) (Michal Chytil, Ladislav Janiga, and Václav Koubek, eds.), Lecture notes in computer science, vol. 324, Springer-Verlag, August 29 - September 2 1988, pp. 405–413.

[KN97] Eyal Kushilevitz and Noam Nisan, *Communication complexity*, Cambridge University Press, Cambridge, UK, 1997.

[KSR92] U. Kebschull, E. Schubert, and W. Rosenstiel, *Multilevel logic synthesis based on functional decision diagrams*, In Proc. European Design Automation Conference, 1992, pp. 43–47.

[Lom04] Chris Lomont, *The hidden subgroup problem – review and open problems*, arXiv:quant-ph/0411037 (2004).

[Mas76] W Masek, *A fast algorithm for the string editing problem and decision graph complexity*, Master's thesis, MIT, 1976.

[ME99] M. Mosca and A. Ekert, *The hidden subgroup problem and eigenvalue estimation on a quantum computer*, arXive e-print quant-ph/9903071, 1999.

[Moo65] Gordon E. Moore, *Cramming more components into integrated circuits*, Electronics **38** (1965), no. 8.

[MS72] A. R. Meyer and L. J. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential time*, Proc. 13th Ann. Symp. on Switching and Automata Theory (Long Beach, CA), IEEE Computer Society, 1972, pp. 125–129.

[MT98] Christoph Meinel and Thorsten Theobald, *Algorithms and data structures in VLSI design OBDD - foundations and applications*, Springer-Verlag Berlin Heidelberg, 1998.

[Nag51] T. Nagell, *Introduction to number theory*, New York: Wiley, 1951.

[NC00] Michael A. Nielsen and Isaac L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, 2000.

[NHK00] Masaki Nakanishi, Kiyoharu Hamaguchi, and Toshinobu Kashiwabara, *Ordered quantum branching programs are more powerful than ordered probabilistic branching programs under a bounded-width restriction*, Computing and Combinatorics, LNCS 1858 (Sydney, Australia), 6th Annual International Conference, COCOON 2000, Springer-Verlag, July 2000, pp. 467–476.

[Pap94] Christos H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.

[Pau26] W. Pauli, Zeits. f. Phys. **36** (1926), 336.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), no. 2, 120–126.

[Sau97] M. Sauerhoff, *A lower bound for randomized read-k-times branching programs*, ECCC, TR97-019, 1997, Available at http://www.eccc.uni-trier.de/eccc/.

[Sau99] M. Sauerhoff, *Complexity theoretical results on randomized branching programs*, Ph. D. Thesis, Universität Dortmund, Shaker Verlag, Aachen, Germany, 1999.

[Sau05] ———, *Quantum vs. classical read-once branching programs*, arXiv:quant-ph/0504198 v1 (2005).

[Sch26a] Schrödinger, Ann. Physik **79** (1926), 734.

[Sch26b] E. Schrödinger, Ann. Physik **79** (1926), 361.

[Sha49] C. E. Shannon, *Communication theory of secrecy systtems*, Bell Syst. Tech. J. **28** (1949), 656–715.

[Sho94] P. W. Shor, *Algorithms for quantum computation: Discrete logarithms and factoring*, Proc. 35th Ann. Symp. on Foundations of Comp.Sci., 1994, pp. 124–134.

[Sho97] ———, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM J. on Computing (1997), no. 26(5), 1484–1509.

[SS04] M. Sauerhoff and Detlef Sieling, *Quantum branching programs and space-bounded nonuniform quantum complexity*, Theoretical Computer Science (2004), no. 334, 177–225.

[SV85] S. Skyum and L. G. Valiant, *A complexity theory based on Boolean algebra*, Journal of the ACM (1985), no. 32, 484–502.

[Tur36] A. M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proc. Lond. Math. Soc., 2, no. 42, 1936, pp. 230–265.

[vN32] Johann v. Neumann, *Mathematische grundlagen der quantenmechanik*, Berlin, 1932.

[Weg00] I. Wegener, *Branching programs and binary decision diagrams*, SIAM Monographs on Discrete Mathematics and Applications, SIAM Press, 2000.

[Yao79] A. C. Yao, *Some complexity question related to distributed computing*, Proceedings of the 11th ACM Symposium on Theory of Computing (Atlanta, Georgia), ACM, April 30 - May 2 1979, pp. 209–213.

Index

(G, K)-coset balanced input, 131
$2 -$ **EQP-BP**, 86

Aaronson, Scot, 136
Ablayev, Farid, iv, v, 57, 77, 80, 81, 84, 86, 87, 93
ACC⁰, 86
addition modulo two, 18
Agrawal, Manindra, 4
al-Khorezmi, Muhammad ibn Musa abu Abdallah, 1
algorithm
  Las Vegas, 52
$\alpha_{\Pi_{L,X}}$, $\alpha_{\Pi_{R,X}}$, 34
alphabet, 6
alphabet,word,language,class, 6
Ambainis, Andris, iv, v
amplitude, 69

Barrington, David A. Mix, 86
Bell states, 76
binary decision diagram, 27, *see* branching program
bitwise negation, 16
Bohr, Niels, 63
Bohr-Sommerfeld conditions, 65
Boolean
  circuit, 85
Boolean function, 27
Born, Max, 65
BPP, 4, 55
BPP-BP, 60
BPP-OBDD, 61
BQP, 113
BQP-BP, 85
BQP-OBDD, 86
branching program, 27–32
  as linear program, 84
  bounded width, 29
  level, 29
  nondeterministic, 57
    bounded error, 60
    one-sided error, 59
    one-sided error for 0-sink, 59
    size, 58
    unbounded error, 59
  obliviuos, 29
  quantum, 77–84
    computation, 78
    ϵ-acceptance, 82
    length, 80

oblivious, 81
output probability, 80
running time, *see* quantum branching program length
state space, 78
transition function, 78
randomized, 57–62
as linear program, 83
read-k-times-only, 29
represents function, 28
restricted, 29
size, 28
subclasses, 30
synchronous, 29
width, 29
BWBPP-BP, 85
BWBQP-BP, 85
BWEQP-BP, 85
BWP-BP, 85

$CC_1(f_n)$, 43
$\mathcal{CC}_{G,K}(A;B)$, 123
$CC(f_n,\Pi)$, 38
$CC(f_n)$, 39
$\mathcal{C}_{G,K}(A)$, 122
Chuang, Isaac, iii, 4
Church, Alonzo, 1
Church-Turing principle, *see* Church-Turing thesis
Church-Turing thesis, 1
quantum, 5
strengthened, 4
strong, 3
class, 6, 8
complement, 8
Classical, 137
code, 40
codes
instantaneous, 40
prefix, 40
coding theory, 40–41
communication, 33–46
complexity of Boolean function, 39
according to fixed partition, 38
function, 36
end message property, 36
prefix-freeness, 36
single output party property, 36
one-way, 42
complexity, 43
protocol, 43
protocol, 34
accept, reject, 38
complexity, 37, 38
computation, 37
many-round, 36
communication complexity
lower bounds, 44
communication function
end message property, 41
prefix-freeness, 40–41
single output party property, 41
communication model, 33

communication protocol
  two-party many round, 34
complementarity principle, 67
completeness equation, 72
completeness relation, *see* completeness equation
complexity, 3
complexity theory, 3
computability theory, *see* theory of computability
computational problem, 6
concatenation, 6
configuration
  Turing machine, 12
Cook, Stephen, 2
Copenhagen interpretation, 68
coRP-BP, 60
cut, 124

Davio decomposition, 31
de Wolf, Ronald, 93
decision problems, 9
decomposition
  Davio, 31
    negative, 31
    positive, 31
  Reed-Muller, 31
  Shannon, 30
Deutsch, David, 4
Dirac notation, 68
Dirac, Paul, 65
$\mathrm{DISJ}_n(x, y)$, 90

Einstein, Albert, 64, 76, 136
empty symbol, *see* λ
EPR pairs, 76
EPR paradox, 76–77
$\mathrm{EQ}_n(x, y)$, 94
EQP-BP, 85
EQP-OBDD, 86
EQUALITY, 24
Equality function, 94
essential elements of reality, 77
Euclid's algorithm, 1

fan-in, 85
FBDD, 30
Feynman, Richard, 68
FFDD, 32
finite functions, 27
Fischer, Michael J., 2
Free Binary Decision Diagram, 30
Free Functional Decision Diagram, 32
Freivalds, Rūsiņš, iv, v
Friedl, Katalin, 114
function, 10
  Boolean, 27
  communication, *see* communication function
  Equality, 94
  finite, 27
  hidden subgroup test, 114
  Periodicity, 94
  represented by the node, 30
  Semi-Simon, 94

set disjointness, *see* $\mathrm{DISJ}_n(x, y)$
Simon test, 115

Gainutdinova, Aida, iv, v, 77, 80, 81, 87
Garey, Michael R,, 3
gate, 85
MOD m, 86
Gill, John T., 56
global phase factor, 73
graph isomorphism, 154
Grigni, Michelangelo, 114
Grover, Lov, iii, 5

Hamaguchi, Kiyoharu, 77
Hartmanis, Juris, 2
Heisenberg, Werner, 64
Hermitian, 72
hidden subgroup test function, 114
Høyer, Peter, 114
Hromkovč, Juraj, 35
$\mathrm{HSP}_{G,K,X}(\sigma)$, 114
Høyer, Peter, 93

$\mathcal{IC}_{G,K}(\Pi)$, 123
$\mathcal{IC}_{G,K,A}(B)$, 123
Indexed Binary Decision Diagram, 30
input
(G, K)-coset balanced, 131
input assignment, 34
instantaneous codes, 40
intractability, 2
$\mathrm{ISA}_n(x, y)$, 90

Johnson, David S., 3
Jordan, Pascual, 65
Jozsa, Richard, 114

k – **BPP-BP**, 85
k – **BQP-BP**, 85
k – **EQP-BP**, 85
k-IBDD, 30
k-OBDD, 30
k – **P-BP**, 85
Karp, Richard, 3
Karpinski, Marek, iv, v, 57, 62, 77, 80, 81, 87
Kashiwabara, Toshinobu, 77
Kayal, Neeraj, 4
Kebschull, Udo, 32
Kitaev, Alexei, 113
Klein, Felix, 64
Kushilevitz, Eyal, 36

λ, 7
language, 6, 7
of balanced words, 15
operations, 7
recursive, 14
recursively enumerable, 14
Las Vegas algorithm, 52
Linear Speedup Theorem, 23
for space complexity, 26

Magniez, Frédéric, 114
$MAJSAT$, 53
matrix

communication, 140
Hermitian, 72
Pauli, 70
unitary, 70
"matrix mechanics", 65
measurement, 72
in a basis, 74
operator, 72
projective, 73
Meyer, Albert R., 2
model
of computation, 5
of communication, 33
MOD_p, 87
Moore's Law, 23
Moore, Christopher, 84, 86
Moore, Gordon E., 23
MOTree, 138
Mubarakzjanov, Rustam, 62
$\text{MWS}_n(x)$, 90

Nakanishi, Masaki, 77
$\mathbf{NC^i}$, 84
Nielsen, Michael, 4
Nisan, Noam, 36
NO_n, 88
nondeterministic Turing machine, 2
NP, 2, 49
NP-BP, 59
NP-completeness, 2
NP-Intermediate, 154–155
NP-OBDD, 61
NPC, 3, 139
NPI, 154
$NRow(f)$, 45
NTIME $(f(n))$, 49

O_2, 15
OBDD, 30
quantum, 80, 82, 86–89
randomized, 60–62
stable, 87
oblivious read-once quantum branching program, *see* quantum OBDD
observable, 73
Ockham, William of, 4
OFDD, 32
operator
Hermitian, 72
NOT, 75
phase flip, 75
unitary, 70
Ordered Functional Decision Diagram, 32
OTree, 138

P, 3, 21
P-BP, 29
P-OBDD, 62
partition
of input, 34
almost balanced, 39
balanced, 39
party, 33

Pauli, Wolfgang, 64
$\mathrm{Period}_{s,n}(\sigma)$, 94
Periodicity function, 94
PERM_n, 88
"pigeon hole" principle, 110
$\Pi_{L,X}$, 34
$\Pi_{R,X}$, 34
Planck's constant, 63
Planck, Max, 63
Podolsky, Boris, 76
Pollette, Christopher, 84, 86
polynomial projection, *see* projection polynomial
PP, 54
PP-BP, 60
PP-OBDD, 61
prefix codes, 40
PRIMES, 9
principle
- Church-Turing, *see* Church-Turing thesis
- "pigeon hole", 110
- uncertainty, 67

projection
- polynomial, 141
  - monotone, 141
  - multiplicity, 141

projections
- read-once, 142

property tester, 114
protocol, *see* communication protocol
PSPACE, 26

QBP, 5
quadratic congruences, 49
quantum
- Church-Turing thesis, 5
- complexity, 5
- computers, 5
- OBDD, 80, 82
- state
  - complexity, 136
  - complexity class, 137
  - separable, 137
  - tree, 136, 137
- states
  - separable, 138

quantum leaps, 64
quantum mechanics, 63–77
- Birthday of, 65
- first postulate, 68
- fourth postulate, 74
- postulates, 68–77
- probabilistic interpretation of, 66
- second postulate, 69
- third postulate, 72

quantum state, 68
- entangled, 74

quantum system
- component, 74
- composite, 74

"quantum theory", 65
qubit, 69

Röntgen, Wilhelm Conrad, 64
Rabin, Michael O., 2
read-once projections, 142
rectangular reduction, 140
recursive functions, 1
recursive language, 14
recursively enumerable language, 14
reduction, 2, 139
  Karp, 139
  rectangular, 140
Reed-Muller decomposition, 31
representation type, 59
Rev-OBDD, 87
Rosen, Nathan, 76
Rosentiel, Wolfgang, 32
RP, 51
RP-BP, 60
RP-OBDD, 61
RQP-OBDD, 86
RSA, 113
Rutherford, Ernest, 64

Santha, Miklos, 114
satisfiability, 3
Sauerhoff, Martin, 61, 78, 87, 88, 90
Saxena, Nitin, 4
Schrödinger, Erwin, 66
Schubert, E., 32
Schulman, Leonard, 114
Semi-Simon$_{s,n}(\sigma)$, 94
Semi-Simon function, 94
Sen, Pranab, 114
Shannon's decomposition, 30
Shannon, Claud, 92
Shor, Peter, iii, 5, 114, 138
Sieling, Detlef, 78, 87, 88
$\mathbf{\Sigma_i}$, 137
Simon$_{s,n}(\sigma)$, 115
Simon test function, 115
Skyum, Sven, 141
Solovay, Robert, 4
Solovay-Strassen algorithm, 4
Sommerfeld, Arnold, 64
space complexity, 25
SPACE $(f(n))$, 25
state space, 68
state vector, *see* quantum state
stationary orbits, 64
Stearns, Richard, 2
Stockmeyer, Larry, 2
Strassen, Volker, 4
string function, 16
strong Church-Turing thesis, 3
superdense coding, 74–76
superposition, 69, 137
Sure/Shor separator, 136

tensor sates hierarchy, 137
$\otimes_{\mathbf{i}}$, 137
theory of computability, 2
time complexity, 18
TIME $(f(n))$, 21
traveling salesman problem, 3
Tree, 137

Tree states, 136
TSH, 137
Turing machine, 1, 11
  accepting, 14
  accepting by clear majority, 55
  computation, 12
    partial, 12
  configuration, 12
    initial, 12
  decidable, 2
  deciding, 14
  deciding by clear majority, 55
  k-string, 17
    configuration, 18
    with input and output, 24
  nondeterministic, 2, 47–50
    accepting by majority, 53
    computability, 50
    computation tree, 48
    deciding, 49
    standard, 51
    time complexity, 49
    transition relation, 48
  operates within time, 21
  polynomial Monte Carlo, 51
    recognizing, 51
  randomized, 50–53
  space, *see* space complexity
  starting state, 11
  time, *see* time complexity
Turing machine with multiple strings,
    see k-string Turing machine
Turing machines, 26
Turing, Alan, 1

uncertainty principle, 67
  generalized, 76
undecidability, 2
uniform family, 85
unique decodability, 40
unitary, 70

Valiant, Leslie G., 141
value of a coset, 124
Vazirani, Monica, 114
Vazirani, Umesh, 114
von Lidenmann, Ferdinand, 64
von Neumann, Johann (John), 68

"wave mechanics", 66
Wegener, Ingo, 146
Wilson's camera, 66
word, 6
$\mathrm{WS}_n(x)$, 90

Yao, Andrew C. C., 33, 41

ZPP, 52
ZPP-BP, 60
ZQP-OBDD, 87