Maastricht University

UHASSELT
KNOWLEDGE IN ACTION

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Randomized computation, probabilistic proof systems and the PCP theorem

Arthur Jansen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Bart KUIJPERS

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

2020
2021

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Randomized computation, probabilistic proof systems and the PCP theorem

Arthur Jansen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Bart KUIJPERS

Acknowledgements

I would like start off by thanking my promotor prof. dr. Bart Kuijpers for all the feedback and guidance he has given me. This was of great value to me in the process of writing this thesis.

Contents

Introduction **4**

1 Background **5**
1.1 Alphabets, strings and languages . . . 5
1.2 Turing machines . . . 5
1.2.1 Non-deterministic Turing machines . . . 7
1.2.2 Representing objects . . . 7
1.3 Time complexity . . . 8
1.3.1 NP-completeness . . . 9
1.4 Space complexity . . . 12

2 Randomized computation **13**
2.1 Probabilistic Turing machines . . . 13
2.2 The classes RP and coRP . . . 14
2.3 An example: probabilistic primality testing . . . 16
2.3.1 Background . . . 16
2.3.2 Fermat primality test . . . 17
2.3.3 Miller-Rabin primality test . . . 17
2.4 The class BPP . . . 19
2.5 The class ZPP . . . 21

3 Interactive proof systems **24**
3.1 The class NP as a proof system . . . 24
3.2 The class IP . . . 25
3.2.1 Deterministic interactive proof systems . . . 25
3.2.2 Probabilistic interactive proof systems . . . 26
3.3 IP = PSPACE . . . 30
3.3.1 IP $\subseteq$ PSPACE . . . 30
3.3.2 PSPACE $\subseteq$ IP . . . 31
3.4 Public-coin proof systems . . . 36
3.5 Multi-prover interactive proof systems . . . 37

4 Zero-knowledge proof systems **39**
4.1 Background: circuit complexity . . . 39
4.2 Definition of a zero-knowledge proof system . . . 40
4.2.1 Indistinguishability of random variables . . . 40
4.2.2 Computational zero-knowledge . . . 41
4.3 The existence of zero-knowledge proofs . . . 42

5 Introduction to the PCP theorem and the hardness of approximation **44**
5.1 The PCP theorem . . . 44
5.2 The hardness of approximation . . . 46
5.2.1 Constraint satisfaction problems . . . 47
5.2.2 Examples of hardness of approximation . . . 49

5.3 Proof of the PCP theorem 51
5.3.1 Expander graphs 51
5.3.2 Constraint graph coloring problem 57
5.3.3 Overview of the proof 59
5.3.4 Proof of the preprocessing lemma 61
5.3.5 Proof of the gap amplification lemma 69

Conclusion **77**

A Nederlandstalige samenvatting **78**
A.1 Inleiding 78
A.2 Achtergrond 79
A.3 Willekeurigheid in algoritmes 80
A.4 Interactieve bewijssystemen 81
A.5 De PCP stelling 82

Introduction

Even before computers as we know them today existed, there were already problems known not to be solvable by a computer. In 1936, Alan Turing introduced a formal model of computation, now known as the Turing machine, and used it to prove that the halting problem cannot be solved by an algorithm. Later on, there was also interest in not only what problems a computer could solve, but also how efficiently they can be solved, that means, how much time and memory does an algorithm require to solve the problem. This is how the field of complexity theory emerged. One of the most important questions within complexity theory, and computer science as a whole, is known as the famous P vs. NP problem. Both P and NP are complexity classes, which are used to classify problems according to their (time or space) complexity.

In this thesis, we study some of the more advanced topics within complexity theory. The first topic we look at is randomized computation. While computers are usually seen as deterministic machines, it is still interesting to consider a model of computation where algorithms have access to randomness, formalized by *probabilistic Turing machines*. This brings us to another open problem in complexity theory, that is, does randomness allow algorithms to solve problems more efficiently?

The next topic we consider are interactive proof systems. An interactive proof system can be seen as an extension to the verifier definition of NP. A proof system consists of two Turing machines (or algorithms), a prover and a verifier. The goal of the prover is to convince the verifier of a certain claim. The prover and verifier can interact with each other by sending messages, thus making the proof system interactive. We can also extend this concept by making the verifier probabilistic. This we obtain the complexity class IP, which consists of problems with such interactive proof system. As it turns out, this is a quite powerful complexity class. In 1992, it was proven that IP = PSPACE, where PSPACE is the class of problems that can be solved by an algorithm that needs at most a polynomial amount of space (that is, memory). We also consider a special sort of interactive proof system, known as a *zero-knowledge proof system*. In such a proof system, the verifier can learn nothing besides the fact that the given claim is true.

For the final topic, we take a look at *probabilistically checkable proofs*, abbreviated as PCP. This again, can be seen as an extension to the verifier definition of NP. The difference is that a PCP verifier is probabilistic, and it will only read a small number of symbols of the proof. Because the verifier is probabilistic, we allow it to make an error sometimes, that is, accept a wrong proof, but only with a small probability that is constant (that means independent of the input). The PCP theorem states that there exists such a PCP verifier for every problem in NP. To emphasize on the importance of the PCP theorem, we note that the Gödel prize has been awarded in 2001 for the original proof of the PCP theorem as well as in 2019, for the discovery of an easier proof. We also look at an important consequence of the PCP theorem, being that, unless P = NP, there exists no polynomial-time approximation algorithms for certain optimization problems.

Chapter 1

Background

In this chapter, we give a short introduction to the background required to understand this thesis. This chapter primarily contains definitions used in basic complexity theory, the definitions and notations used in this chapter are taken from the classical textbook of Sipser [19]. We start by introducing a formal model of computation called the Turing machine. Using this model we can then define the time complexity of an algorithm, along with the classes P and NP. Finally we look at NP-complete problems, which can intuitively be seen as the hardest problems in NP.

1.1 Alphabets, strings and languages

Here, we give definitions of alphabets, strings and languages. While the connection with computation might not be clear right now, it becomes clear in the sections following this.

Definition 1.1 (Alphabet) We define an *alphabet* to be any nonempty finite set. Elements of an alphabet are called *symbols*. □

We usually use the letters Σ or Γ for alphabets.

Definition 1.2 (String) A *string* over an alphabet Σ is a finite sequence of symbols of Σ. We say the length of a string u is the amount of symbols in u and is denoted by $|u|$. A string can have length 0 and this specific string is called the *empty string*, denoted by ε. The set of all finite strings over Σ is denoted by Σ^*. □

Definition 1.3 (Language) A language over an alphabet Σ is a set of strings over Σ. □

Definition 1.4 (Complement of a language) The complemented of a language L over an alphabet Σ is denoted by $\overline{L}$ and is defined as $\overline{L} = \Sigma^* \setminus L$. □

1.2 Turing machines

Now, we are ready to introduce the Turing machine model of computation. A Turing machine formalizes the notion of an algorithm. A Turing machine has access to a component called the *tape*, serving as the memory of the machine which it can write to and read from. A tape consists of an infinite sequence of *tape cells* and each tape cell contains a symbol of the tape alphabet. We say that the first cell of this sequence is on the left side of the tape, in other words the tape is bounded at its left side. During the computation, a Turing machine has a tape head positioned on some cell of the tape and can only read from and write to the cell under the tape head. Also during the computation a Turing machine will be in some state out of a finite set of states. A computation is a sequence of steps, where each step is given by the *transition function*, usually called the delta function. The transition function decides, given the current state and the symbol currently under the tape head, the new state of the Turing machine, the symbol to write under the

tape head and the direction to move the tape head in by one cell (left or right). At the start of a computation the Turing machine is in the special state called the start state and its tape contains a string called the input string (over the input alphabet) in its leftmost cells, the other cells start with the blank symbol $\sqcup$. Finally a computation ends when either the accept state or reject state is reached.

What follows is a formal definition.

Definition 1.5 (Turing machine) A *Turing machine* is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where:

- Q is the set of states,
- Σ is the input alphabet not containing the blank symbol $\sqcup$,
- Γ is the tape alphabet where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the start state,
- $q_{accept} \in Q$ is the accept state, and
- $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

□

Definition 1.6 (Configuration) A configuration of a Turing machine M consists of a state q and two strings u and v over the tape alphabet Γ and is written as uqv. During the computation we say M is in this configuration when

- M is in state q,
- the content of the tape cells to the left of the tape head is u, and
- the content of the tape cells to the right of the tape head (including the cell under the head) is v followed by an infinite amount of blank symbols.

□

We say that a configuration C_1 yields configuration C_2 when the Turing machine can go from C_1 to C_2 in one step of the computation. There are three special cases of configurations.

- The starting configuration of M is $q_0 w$ (here, q_0 is preceded by the empty string) where w is the input string.
- An accepting configuration is any configuration of which the state is the accept state q_{accept}.
- A rejecting configuration is any configuration of which the state is the reject state q_{reject}.

Both accepting and rejection configurations are called halting configurations because when reaching either the accept state or the reject state, the computation halts. We say that a Turing machine M accepts an input string w if a sequence of configurations $C_1, \ldots, C_k$ exists, where

- C_1 is the starting configuration,
- each C_i in $C_1, \ldots, C_{k-1}$ yields (by one step of the delta function) C_{i+1}, and
- C_k is an accepting configuration.

We denote the language $L(M)$ to be the set of strings that Turing machine M accepts:

$$L(M) = \{w \mid M \text{ accepts } w\}.$$

Definition 1.7 (Turing-recognizable) We say a language L over alphabet Σ is Turing-recognizable if there exists a Turing machine M with input alphabet Σ such that $L(M) = L$. We also say that M recognizes L. □

There is an issue that could arise when working with Turing-recognizable languages. Given language L and a Turing machine M that recognizes L, we know that if some string w is an element of L, then M will halt after a finite number of steps. However when w is not an element of L, by our definition it is not guaranteed that M halts at some point (but if it ever halts, it does so in the reject state).

In the next sections, when we look at the time complexity (and in other chapters of this thesis) it would be a lot easier to only consider Turing machines that always halt. Turing machines that halt on all inputs are called deciders.

Definition 1.8 (Turing-decidable) A language L is called Turing-decidable if there exists a Turing machine M that halts on all inputs such that $L(M) = L$. We also say that M decides L. □

Note 1.9 In the remainder of this thesis we will make some assumptions when using Turing machines (unless specified otherwise):

- The input alphabet is the binary alphabet $\Sigma = \{0, 1\}$, the symbols of which are called bits, and the tape alphabet is $\Gamma = \{0, 1, \sqcup\}$.
- When we mention a Turing machine, we mean a Turing machine that halts on all inputs (a decider).

□

1.2.1 Non-deterministic Turing machines

We look at a variant of the Turing machine, called the non-deterministic Turing machine. Non-determinism is added by allowing a single configuration of the machine to yield possibly multiple configurations. Then the computation is not a sequence of configurations but it can be modelled as a tree. We say that if one of the branches of the computation tree ever reaches an accept configuration, then the machine accepts. The only change needed to the definition is in the transition function: for a non-deterministic Turing machine the transition function has the form

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

where $\mathcal{P}(S)$ denotes the power set of S.

In this context, we call a decider a non-deterministic Turing machine of which all branches of the computation halt on all inputs.

It can be shown that the set of languages decidable by non-deterministic Turing machines is equal to the set of languages decidable by deterministic Turing machines (for a proof see [19]). So, we can say that, in terms of decidability, a non-deterministic Turing machine is just as powerful as a deterministic Turing machine.

1.2.2 Representing objects

Until now we have seen that Turing machines take as input a string. While there are algorithms that work on strings, we would also like to consider algorithms working with other types of objects like numbers, graphs, sets, etc. To allow this we will encode objects to their string representation. There are many ways to encode objects into strings however the exact method used is not very important. We use the following notation: if we have some object O then the string representation of O is denoted $\langle O \rangle$. We also assume that if the input of a Turing machine is supposed to be the encoding $\langle O \rangle$ for some object O then the Turing machine will first test that the input is a valid encoding of an object and if this is not the case the Turing machine will reject.

1.3 Time complexity

In the previous section, we saw what it means for a language to be decidable. However, we did not consider the time required by the computation of a Turing machine that decides a language. It might be that for some languages it is very fast to decide membership while for other languages this might take a very long time. That is why we give a measurement for how long a computation takes, called the *time complexity*. Using this, we then show a set of languages that are considered tractable to compute.

Definition 1.10 (Deterministic time complexity) The time complexity of a deterministic Turing machine M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of length n. □

Usually, we are not interested in the exact time complexity of a Turing machine. Therefore, we use the big-O notation, where we only consider the asymptotic behaviour of the time complexity.

Definition 1.11 (Big-O notation) Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq c \cdot g(n)$. When $f(n) = O(g(n))$, we say that $g(n)$ is an *asymptotic upper bound* for $f(n)$. □

Using this we introduce complexity classes, which are sets of languages that can be computed with time complexity asymptotically bounded by the same function.

Definition 1.12 (Deterministic time complexity class) Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. We define the time complexity class $\mathrm{TIME}(t(n))$ to be the collection of all languages that are decidable by a deterministic Turing machine with time complexity $O(t(n))$. □

Next we give the definition for the time complexity class P, the class P is considered to be the class of languages that are efficiently computable, or tractable.

Definition 1.13 (The class P) P is the class of languages that are decidable in polynomial time on a deterministic Turing machine, or

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k).$$

□

Next we will define time complexity for a non-deterministic Turing machine and the corresponding time complexity class.

Definition 1.14 (Non-deterministic time complexity) The time complexity of a non-deterministic Turing machine N is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n. □

Definition 1.15 (Non-deterministic time complexity class) Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the time complexity class $\mathrm{NTIME}(t(n))$ to be the collection of all languages that are decidable by a non-deterministic Turing machine with time complexity $O(t(n))$. □

Finally we define the class NP, the equivalent to P for non-deterministic Turing machines.

Definition 1.16 (The class NP) NP is the class of languages that are decidable in polynomial time by a non-deterministic Turing machine.

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k)$$

□

We also define the class coNP, the class of languages whose complement is in NP.

Definition 1.17 (The class coNP) coNP is the class of languages whose complement is in NP, or

$$\text{coNP} = \{L \mid \overline{L} \in \text{NP}\}$$

□

We give a second definition for NP, which informally says that NP contains all languages that can be efficiently verified. What that means is shown in the following definitions.

Definition 1.18 (Polynomial time verifier) A verifier for a language L is a deterministic Turing machine V, where

$$L = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

The time complexity of V is measured only in terms of the length of w. So a polynomial time verifier has a time complexity polynomial in the length of w. We say a language L is polynomially verifiable if it has a polynomial time verifier. □

The string c in the definition is called the *certificate*, or *proof of membership*.

Definition 1.19 (Verifier definition of NP) NP is the class of languages that are polynomially verifiable. □

As an example we show that the problem of checking existence of an independent set in a graph is polynomially verifiable and thus in NP. An independent set in a graph is a set of nodes such that no pair of nodes in the set are neighbours. We define the problem as follows:

$$\text{INDSET} = \{\langle G, k\rangle \mid G \text{ is a graph with an independent set of } k \text{ nodes}\}$$

The following is a polynomial time verifier V for INDSET.

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Check if c is a set of k nodes from G.
2. Check if there is no pair of nodes in c that are neighbours in G.
3. Accept if both checks were successful, otherwise reject. "

1.3.1 NP-completeness

There are some languages in the class NP that have the interesting property that if the language is in P then all languages of NP are in P, or P = NP. These languages are called NP-complete. To show this property, we make use of *polynomial time reductions*.

Definition 1.20 (Polynomial time computable function) A function $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some deterministic polynomial time Turing machine M exists that given input w halts with $f(w)$ on its tape. We say that f is computable in polynomial time. □

Definition 1.21 (Polynomial time reduction) We say a function f is a polynomial time reduction of language L_1 to language L_2 if

- for every string $w \in \Sigma^*$ we have $w \in L_1$ iff $f(w) \in L_2$, and
- f is computable in polynomial time.

We also say L_1 is polynomial time reducible to L_2, denoted by $L_1 \leq_P L_2$. □

Definition 1.22 (NP-completeness) A language L is NP-complete if it satisfies the following conditions:

- L is in NP, and
- for every L' in NP, L' is polynomial time reducible to L.

We say a language is NP-hard if it satisfies the second condition. □

An example of a NP-complete language is 3SAT, containing all satisfiable 3CNF formulas (boolean formulas in conjunctive normal form where each clause consists of 3 literals). We will not give the proof here, but it can be found in [19].

Theorem 1.23 (Cook, 1971) 3SAT is NP-complete. □

It might seem quite challenging to prove that a certain language L is NP-complete, because we have to show that there exists a polynomial time reduction for every language in NP to L. Fortunately this is not necessary once we know of a single NP-complete language, as can be seen in the following theorem.

Theorem 1.24 If L_1 is NP-complete and $L_1 \leq_P L_2$ with $L_2 \in$ NP, then L_2 is NP-complete. □

To proof this, we use the fact that the composition of two polynomial time reductions is a polynomial time reduction itself. So, if L_1 is NP-complete we know that for every $L \in$ NP there exists a polynomial time reduction f of L to L_1. Now, if there exists a polynomial time reduction g from L_1 to $L_2 \in$ NP then $g \circ f$ is a polynomial time reduction of L to L_2. And thus C is NP-complete.

To give an example we show that INDSET is NP-complete.

Theorem 1.25 INDSET is NP-complete. □

Proof. We already know that INDSET is in NP, so we only have to show that every language in NP is polynomial time reducible to INDSET. We do this by showing a polynomial time reduction f of 3SAT to INDSET. On input a 3CNF formula ϕ, $f(\phi)$ will output $\langle G, k \rangle$ such that ϕ is satisfiable if and only if graph G has an independent set of k nodes. The graph G will be constructed as follows: for every clause in ϕ we add a group of three nodes to G, one for each literal in the clause. Two nodes in G will share an edge when they are part of the same group or when their corresponding literals are negations of each other. An example of this construction can be seen in Figure 1.1.

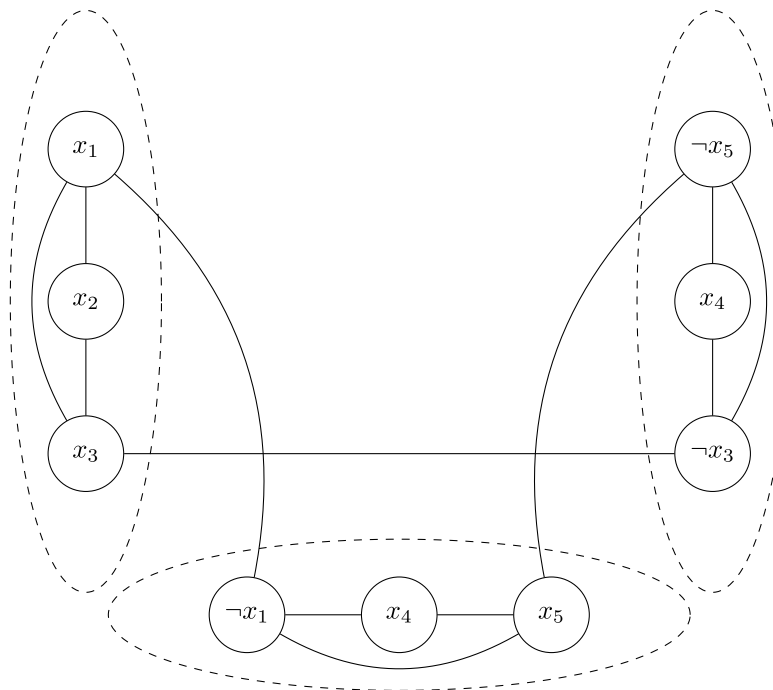


Figure 1.1: The graph constructed by the reduction of formula $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_5) \wedge (\neg x_5 \vee x_4 \vee \neg x_3)$. The dashed ellipses represent the groups and are only shown for clarity.

This reduction is correct because:

- If ϕ is satisfiable, then there exists some satisfying assignment. Then we can find a independent set of size k where k is the number of clauses in ϕ. To do this, we will select one node, from every three nodes corresponding to a clause, of which the accompanying literal is true by the satisfying assignment. There must be at least one such node for each clause because an assignment that satisfies ϕ satisfies every clause in ϕ. The selected nodes now form an independent set because we only selected a single node per clause and we did not select nodes corresponding to literals that are negations of each other because both literals could not be true given the assignment. Finally, the independent set has k nodes because we selected exactly one node per clause and there are k clauses.
- If G has an independent set of size k where k is the number of clauses in ϕ, we know ϕ is satisfiable because we can obtain a satisfying assignment from the independent set. For every node in the independent set, we simply make its corresponding literal true. Because nodes of a single clause are connected the independent set has to contain a node for every clause. And once we make some literal true we will never try to make its negation true because these nodes are connected.

It should be clear that f can be computed in polynomial time. □

To conclude this section, we define coNP-completeness and give an interesting property about this.

Definition 1.26 (coNP-completeness) A language L is coNP-complete if it satisfies the following conditions:

- L is in coNP, and
- for every L' in coNP, L' is polynomial time reducible to L.

□

Theorem 1.27 A language L is NP-complete if and only if $\overline{L}$ is coNP-complete. □

Proof. We proof the two directions.

1. Language L is NP-complete implies $\overline{L}$ is coNP-complete.

   Given an NP-complete language L, we show that for every language L' in coNP, L' is polynomial time reducible to $\overline{L}$. Because $L' \in$ coNP, $\overline{L'} \in$ NP. So, there exists a polynomial time reduction f of $\overline{L'}$ to L. Then, we know that for every string x, $x \in \overline{L'}$ if and only if $f(x) \in L$. This is equivalent to saying that for every string x, $x \in L'$ if and only if $f(x) \in \overline{L}$. We showed that f is also a polynomial time reduction from L' to $\overline{L}$.

2. Language L is coNP-complete implies $\overline{L}$ is NP-complete.

   This proof is similar to the one above. Given an coNP-complete language L, we show that for every L' in NP, L' is polynomial time reducible to $\overline{L}$. Because $L' \in$ NP, $\overline{L'} \in$ coNP. So, there exists a polynomial time reduction f of $\overline{L'}$ to L. Then, we know that for every string x, $x \in \overline{L'}$ if and only if $f(x) \in L$. This is equivalent to saying that for every string x, $x \in L'$ if and only if $f(x) \in \overline{L}$. We showed that f is also a polynomial time reduction from L' to $\overline{L}$.

□

This means we can give an alternative but equivalent definition for coNP-completeness.

Definition 1.28 (coNP-completeness) A language L is coNP-complete if $\overline{L}$ is NP-complete. □

1.4 Space complexity

In this section, we consider the amount of space a Turing machine uses instead of time. Similar to time complexity, we define the *space complexity* of a Turing machine. Finally, we show the class PSPACE and a language that is complete in this class.

Definition 1.29 (Space complexity) The space complexity of a deterministic Turing machine M is the function $f : \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum number of tape cells that M scans on any input of length n.

The space complexity of a non-deterministic Turing machine N that halts on all inputs is a function $f : \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum number of tape cells the N scans on any branch of its computation on any input of length n. □

Just like with time complexity, we are not interested in the exact space complexity of a Turing machine. Therefore we introduce space complexity classes using the big-O notation.

Definition 1.30 (Space complexity classes) Let $f : \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. The space complexity class SPACE($f(n)$) is defined as:

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is decided by an } O(f(n)) \text{ space deterministic Turing machine}\}.$$

Similarly, the class NSPACE($f(n)$) is defined as:

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is decided by an } O(f(n)) \text{ space non-deterministic Turing machine}\}.$$

□

Next, we define the class PSPACE.

Definition 1.31 (The class PSPACE) PSPACE is the class of languages that are decidable in polynomial space by a deterministic Turing machine, or

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k).$$

□

Just like with NP, we define PSPACE-completeness.

Definition 1.32 (PSPACE-completeness) A language L is PSPACE-complete if it satisfies the following conditions:

- L is in PSPACE, and
- for every L' in PSPACE, L' is polynomial time reducible to L.

□

To give an example of a PSPACE complete language, we define the language TQBF. TQBF stands for true quantified boolean formula and contains all fully quantified boolean formulas that evaluate to true. A boolean formula is fully quantified if it contains no free variables, in other words, every variable appears within the scope of some quantifier.

Theorem 1.33 TQBF is PSPACE-complete. □

We do not give the proof here, but it can be found in [19].

Chapter 2

Randomized computation

As explained in the previous chapter, when defining the class P, we attempt to define a class of problems that are computable in practice by an algorithm, or tractable. Although the existence of true randomness might be more of a philosophical question, nowadays a lot of algorithms act as if they have access to randomness by using *pseudo-random generators*. Therefore, it is useful to consider the effect of having access to randomness has on computation. If we allow a polynomial-time Turing machine to have access to randomness, can it decide more languages? In this chapter, we formalize a model of computation with access to randomness called the probabilistic Turing machine and define classes of languages that are decidable by these. One could even view these new classes as an alternative way of capturing languages that are computable in practice.

2.1 Probabilistic Turing machines

When we add the possibility of making random decisions, a Turing machine is obviously not deterministic any longer. In fact, we define a probabilistic Turing machine very similarly to how we define non-deterministic Turing machines. The difference is that we do not say that a certain input is either accepted or rejected, but we assign a probability to both outcomes. Just like with non-deterministic Turing machines, we only consider probabilistic Turing machines that, for every input, halt on all branches.

Definition 2.1 (Probabilistic Turing machine) The transition function of a probabilistic Turing machine has the same form as one of a non-deterministic Turing machine, that is:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

However, we require that in every step of the transition function, there are at most two choices. This means: for every state q and tape symbol a we have $|\delta(q, a)| \leq 2$. We call each non-deterministic step (a step with two choices) a *coin-flip* step. We define the probability that a probabilistic Turing machine M accepts an input x as the probability that an accepting branch of the computation tree of M on input x is chosen, or

$$\Pr[M \,\text{accepts}\, x] = \sum_{\text{b is an accepting branch}} \Pr[b],$$

where the probability that a branch b is chosen is $\Pr[b] = 2^{-k}$ where k is the number of coin-flip steps on the branch b. Since there are only two possible outcomes, it follows that $\Pr[M \,\text{rejects}\, w] = 1 - \Pr[M \,\text{accepts}\, w]$. The time and space complexity of a probabilistic Turing machine are defined in the same way as for a non-deterministic Turing machine. □

Let us elaborate more on the probability that a probabilistic Turing machine accepts (or rejects) an input. Say we have a probabilistic Turing machine M that, on a certain input x, has the computation tree shown in Figure 2.1. Note that the nodes in this tree represent configurations

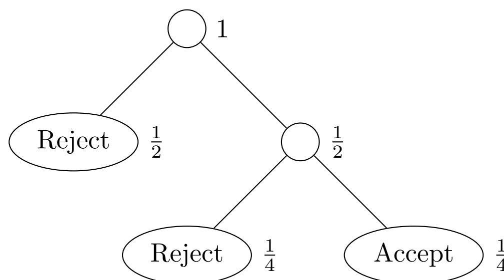


Figure 2.1: An example computation tree of a probabilistic Turing machine.

of the Turing machine. Now we give a more intuitive way of thinking about the probability that a probabilistic Turing machine accepts. Let us say that each node in the computation tree has a certain probability of being visited. The way we calculate this probability is as follows, we start at the root node with probability 1. Then, when a parent node has a single child node, the child node has the same probability as its parent node. When a parent node has two child nodes (this means we are in a coin-flip step), the probability of both of the children is half of that of the parent. Finally, we can say that the probability that a Turing machine accepts (or rejects) is the total probability that an accepting (or rejecting) configuration is visited. In Figure 2.1, the probability of visiting each node is written next to the corresponding node. Now, in our example the probability that M accepts x is the probability of visiting any accepting configuration. There is only one accepting configuration in this tree, whose probability of being visited is $\frac{1}{4}$, thus we have $\Pr[M \text{ accepts } x] = \frac{1}{4}$. For the probability that M rejects x, we have to take the sum of the probabilities that the reject configurations are reached, so we have $\Pr[M \text{ rejects } x] = \frac{1}{2} + \frac{1}{4} = \frac{3}{4}$.

On a higher level, one can view a probabilistic Turing machine as an algorithm that can, at any point, flip a coin and choose its next step based on the outcome of this coin-flip (where a coin-flip has two outcomes and each outcome has probability $\frac{1}{2}$ of occurring).

2.2 The classes RP and coRP

Now that we have defined probabilistic Turing machines, let us take a look at how powerful these are compared to deterministic Turing machines. Say we have a probabilistic, polynomial-time Turing machine M and a language L, with the following property: for every input x, if $x \in L$, then $\Pr[M \text{ accepts } x] = 1$, while if $x \notin L$, then $\Pr[M \text{ accepts } x] = 0$. It seems reasonable to say that M decides the language L. However, in this case it is not very useful for M to be probabilistic, because if M would always choose the same branch, it would be deterministic and still decide the language L (using the same time complexity). So, access to randomness on its own surely does not make our model of computation any more powerful. It becomes more interesting when we allow the probabilistic Turing machine to make an error sometimes. For example, this could mean that our probabilistic machine M sometimes (depending on the outcome of the coin-flips) rejects an input x while $x \in L$ and we still say that M decides L. It is in this way that we define randomized time complexity classes RTIME($t(n)$), where a probabilistic Turing machine is allowed to erroneously reject an input with probability at most $\frac{1}{2}$.

Definition 2.2 (The class RTIME($t(n)$)) Let $t : \mathbb{N} \to \mathbb{R}$ be a function. A language L is in the class RTIME($t(n)$) if there exists a probabilistic Turing machine M with time complexity $O(t(n))$, such that, for every input x:

- if $x \in L$, then $\Pr[M \text{ accepts } x] \geq \frac{1}{2}$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] = 1$.

□

Similar to how we defined the class P, we define the class RP.

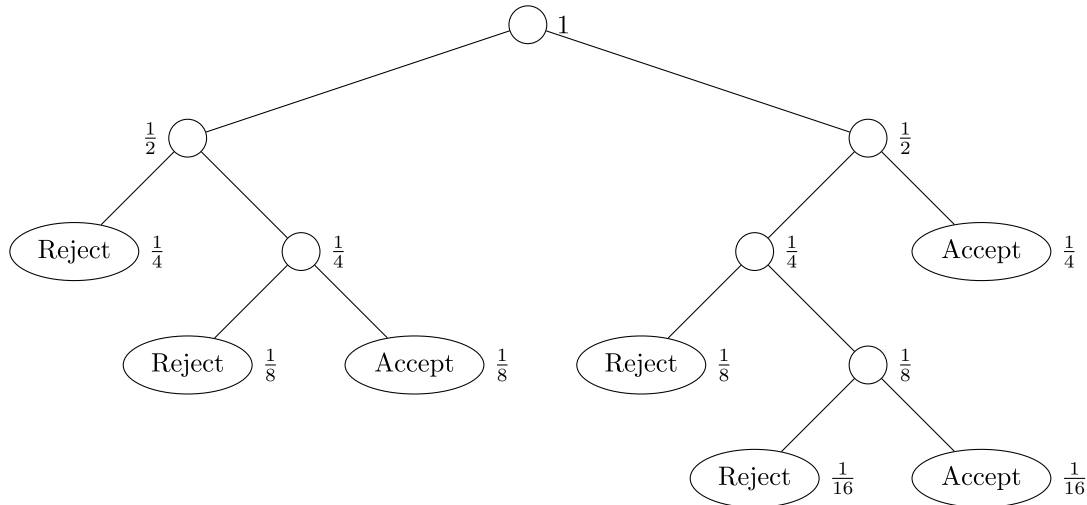


Figure 2.2: The computation tree obtained by performing error reduction.

Definition 2.3 (The class RP) We define

$$\text{RP} = \bigcup_{k \in \mathbb{N}} \text{RTIME}(n^k).$$

□

We say the error probability is the maximum probability that an input is wrongly rejected, that is, rejected while being in the language. In the above definition, the error probability can be up to $\frac{1}{2}$. However, we could have used any constant in $]0, 1[$ in the definition, without changing the class. This is because we can perform an *error reduction* on these probabilistic Turing machines. As an example, let us look back at the computation tree shown in Figure 2.1. Say this is the computation tree of a probabilistic Turing machine M on a certain input x. We calculated that the probability that M accepts x is $\frac{1}{4}$. Now, let us generalize that, say L is a language and for every input $x \in L$, the probability that M accepts x is at least $\frac{1}{4}$. On the other hand, given an input $x \notin L$, then M rejects x with probability 1. This means the error probability of M is $\frac{3}{4}$. Note that according to the definition we gave above, this does not prove that $L \in \text{RP}$. However, given M, we can construct a probabilistic Turing machine M' for L that has a lower error probability. What M' does is simply running M, if M accepts, M' does so too. If M rejects, than M' will run M one more time, and output the same as M (that is, if M accepts then M' accepts and if M rejects then M' rejects as well). Let us look at the computation tree of M', on the same input as our previous example, shown in Figure 2.2. As we can see, this computation tree is obtained by replacing the reject nodes in the original tree by a copy of the original tree itself. Again, the probability of visiting each node is written next to the nodes. We can now calculate the probability that M' accepts x, $\Pr[M \text{ accepts } x] = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = \frac{7}{16}$. So, the error probability has been reduced from $\frac{3}{4}$ to $\frac{9}{16}$. While this is still not good enough to satisfy our definition, where the error probability is at most $\frac{1}{2}$, we can simply repeat this error reduction technique until we get a sufficiently low error probability. In this case we ran M 2 times, however when we run M a number of times k, the error probability decreases exponentially in k. In this case specifically, we get an error probability of $(\frac{3}{4})^k$.

So, we can conclude that to prove that a language L is in RP, we only have to show a probabilistic Turing machine that has some constant error probability less than 1. The same holds for languages in $\text{RTIME}(t(n))$ for every function t, because running a probabilistic Turing machine with $t(n)$ time complexity k times, where k is a constant, takes $k \cdot t(n)$ time, and $k \cdot t(n) = O(t(n))$.

Similar to how we define coNP, we can define the class coRP.

Definition 2.4 (The class coRP) A language L is in coRP if and only if $\overline{L}$ is in RP. □

An interesting property of coRP is that we could have defined it in an alternative way. This definition is similar to the one of RP, but the error is allowed when $x \notin L$.

Property 2.5 A language L is in the class coRP if and only if there exists a probabilistic, polynomial-time Turing machine M, such that, for every input x:

- if $x \in L$, then $\Pr[M \text{ accepts } x] = 1$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] \geq \frac{1}{2}$.

□

Proof. Proving this is straightforward, given a language $L \in$ RP, by definition, we have $\overline{L} \in$ coRP. Let M be a probabilistic Turing machine that decides L, thus we have that for every input x, if $x \in L$ then $\Pr[M \text{ accepts } x] \geq \frac{1}{2}$ and if $x \notin L$, then $\Pr[M \text{ rejects } x] = 1$. Now if we modify obtain the probabilistic Turing machine M' by swapping the accept and reject states of M, then clearly M' decides L and L satisfies the conditions of Property 2.5. The other way can be proved in the same way. □

2.3 An example: probabilistic primality testing

A classic example of a problem solved by probabilistic algorithms is primality testing, that is, given a number test if it is a prime number. Before showing an algorithm to test primality, we first give a short background of number theory.

2.3.1 Background

A natural number greater than 1 is called a prime number (or just a prime) if it is not a product of two smaller natural numbers. A natural number that is not a prime number is called a composite number. Two natural numbers, a and b, are called coprime if their greatest common divisor equals 1, that is: $\gcd(a, b) = 1$. It follows that a prime number is coprime to all natural numbers (excluding 0) smaller than itself. We say two integers, a and b, are equivalent modulo n, for some natural number n greater than 1, if their difference is divisible by n, in other words, there exists an integer k such that $a - b = k \cdot n$. When integers a and b are equivalent modulo n, we write $a \equiv b \pmod{n}$. Now, we show Fermat's little theorem, which we use later in the primality testing algorithm.

Theorem 2.6 (Fermat's little theorem) If p is a prime number and a is coprime to p, then $a^{p-1} \equiv 1 \pmod{p}$. □

We also show Euclid's lemma, which we need for the proof of the next property.

Lemma 2.7 (Euclid's lemma) If a product of two integers, $a \cdot b$ is divisible by a prime number p, then at least one of the two integers, a and b, is divisible by p. □

We say an integer a is a square root of an integer b modulo n, if $a^2 \equiv b \pmod{n}$. The following property about the square roots of 1 modulo a prime number, will also be used in the primality testing algorithm.

Property 2.8 When p is an odd prime number and x is a square root of 1 modulo p, then either $x \equiv 1 \pmod{p}$ or $x \equiv -1 \pmod{p}$. □

Proof. Let p be an odd prime number and let x be a square root of 1 modulo p. Then we have $x^2 \equiv 1 \pmod{p}$. From this follows:

$$x^2 - 1 = (x - 1) \cdot (x + 1) \equiv 0 \pmod{p}.$$

So, there exists an integer k such that $(x - 1) \cdot (x + 1) = k \cdot p$, in other words, $(x - 1) \cdot (x + 1)$ is divisible by p. By Euclid's lemma, this means that at least one of the two factors, $(x - 1)$ and $(x + 1)$, must be divisible by p.

- If $x - 1$ is divisible by p, then $x - 1 \equiv 0 \pmod{p}$ and thus $x \equiv 1 \pmod{p}$.
- On the other hand, if $x + 1$ is divisible by p, then $x + 1 \equiv 0 \pmod{p}$ and $x \equiv -1 \pmod{p}$.

□

Example 2.9 Let us take a prime number, $p = 5$. Clearly, 1 and -1 are trivial square roots of 1 modulo 5. There are many more square roots of 1 modulo 5, for example $6^2 = 36 \equiv 1 \pmod{5}$ or $4^2 = 16 \equiv 1 \pmod{5}$. However, all these square roots are equivalent to either 1 or -1 modulo 5, following our example, $6 \equiv 1 \pmod{5}$ and $4 \equiv -1 \pmod{5}$.

Let us show that Property 2.8 does not hold for square roots of 1 modulo a composite number. Take the composite number $n = 8$. One of the square roots of 1 modulo 8 is 3 because $3^2 = 9 \equiv 1 \pmod{8}$. And this while 3 is not equivalent to 1 or -1 modulo 8. □

2.3.2 Fermat primality test

Using Fermat's little theorem, we can perform a very simple primality test, called the *Fermat primality test*. On input a number x, we randomly choose a number $a \in \{2, \ldots, x - 1\}$ and test if $a^{x-1} \equiv 1 \pmod{x}$. If the test fails, we know that x is not prime. So, this test will accept prime numbers with probability 1. However, if the test does not fail, we do not know that for sure that x is a prime. In fact, there exist composite numbers that pass the Fermat primality test for any a that is coprime to x. Such numbers are known as Carmichael numbers.

Definition 2.10 (Carmichael number) A Carmichael number is a composite number x, such that for every number a coprime to x, we have:

$$a^{x-1} \equiv 1 \pmod{x}.$$

□

Example 2.11 Let us give an example of the Fermat primality test. Say we want to test the number 11 for primality (in this case, the number is indeed prime). So, we have to choose a random $a \in \{2, \ldots, 11 - 1 = 10\}$, let us say we picked $a = 4$. Then we verify that $4^{11-1} \equiv 1 \pmod{11}$, which is true. And thus the test accepts x as a prime.

Now, let us apply the Fermat primality test to a composite number, say 6. We choose a random $a \in \{2, \ldots, 5\}$, let us say 2. Then we compute $2^{6-1} = 32$, clearly $32 \not\equiv 1 \pmod{6}$, and the test will reject 6.

As a final example, let us perform the test on a Carmichael number. The smallest Carmichael number is 561. Again, we chose a random $a \in \{2, \ldots, 560\}$, say 326. Now we compute 326^{560}, we do not include the result here because this is a very large number. However, it happens that $326^{560} \equiv 1 \pmod{561}$. So, the test wrongly accepts 561 as a prime number. □

Because there exist an infinite number of Carmichael numbers (see [2]), the Fermat primality test has a serious flaw. Therefore, we show a more accurate primality test in the next subsection.

Note 2.12 While there is indeed a flaw with this test, in practice the Fermat primality test is sometimes used before using more accurate, but also more time-consuming, primality tests. This way, some composite numbers can already be rejected by the relatively fast Fermat primality test before having to perform the more time-consuming tests. □

2.3.3 Miller-Rabin primality test

The more accurate probabilistic algorithm we describe here is called the Miller-Rabin primality test [15]. This test works by checking that the input number is a *strong probable prime*, as defined below, to a randomly chosen base.

Definition 2.13 (Strong probable prime) Let $n > 2$ be an odd integer. We can write n as $d \cdot 2^s + 1$, where d and s are integers and d is odd. Then n is called a strong probable prime to a base a if one of the following conditions hold:

1. $a^d \equiv 1 \pmod{n}$, or
2. $a^{d \cdot 2^r} \equiv -1 \pmod{n}$ for some r in $\{0, \ldots, s-1\}$.

□

This test will accept primes with probability 1, as can be seen from the following property.

Property 2.14 An odd prime p, is a strong probable prime to every base $a \in \{2, \ldots, p-1\}$. □

Proof. Let p be an odd prime. Because p is odd, we can write p as $d \cdot 2^s + 1$, with d and s integers. By Fermat's little theorem, we know:

$$a^{d \cdot 2^s} \equiv 1 \pmod{p}$$

for every $a \in \{2, \ldots, p-1\}$. Now, $a^{d \cdot 2^{s-1}}$ is a square root of $a^{d \cdot 2^s}$ modulo p and thus a square root of 1 modulo p. By Property 2.8, we know that a square root of 1 modulo p is equivalent to 1 or -1 modulo p. If $a^{d \cdot 2^{s-1}} \equiv -1 \pmod{p}$, then the second condition of the definition holds, and p is a probable prime to base a. Otherwise we know that $a^{d \cdot 2^{s-1}} \equiv 1 \pmod{p}$, and then $a^{d \cdot 2^{s-2}}$ is a square root of 1 modulo p. Then, we can continue this reasoning. In general, either there exists an $r \in \{0, \ldots, s-1\}$ such that $a^{d \cdot 2^r} \equiv -1 \pmod{p}$, or $a^d \equiv 1 \pmod{p}$. In both cases, the definition is satisfied. □

On the other hand, unlike with the Fermat primality test, we can bound the probability that the Miller-Rabin test (wrongly) accepts a composite number. This is shown in the following property.

Property 2.15 A composite number n is a strong probable prime to at most $\frac{1}{4}$ of the bases in $\{2, \ldots, n-1\}$. □

We do not give a proof here, see [15].

Now, let us define the language that contains all prime numbers.

$$\text{PRIMES} = \{\langle n \rangle \mid n \text{ is a prime number}\}.$$

Using the Miller-Rabin test, we can construct a Turing machine, M, for the language PRIMES. Note that we encode numbers using their binary representation, which means that the length of an input number n is logarithmic in n, that is $|\langle n \rangle| = \lfloor \log n \rfloor + 1$.

"On input $\langle n \rangle$, with n a natural number greater than 1

1. If $n = 2$, accept.
2. If n is even, reject.
3. Compute integers d and s such that $n = d \cdot 2^s + 1$, where d is odd.
4. Choose a random a in $\{2, \ldots, n-1\}$.
5. If $a^d \equiv 1 \pmod{n}$, accept.
6. Repeat for r in $0, \ldots, s$:
   (a) If $a^{d \cdot 2^r} \equiv -1 \pmod{n}$, accept.
7. If we reach this point, reject. "

We know by Property 2.14 that the Miller-Rabin test always accepts primes. Thus, if $x \in$ PRIMES, then $\Pr[M \text{ accepts } x] = 1$. On the other hand, Property 2.15 tells us that for at least $\frac{1}{4}$ of the bases we choose, the test will reject a composite number. In other words, if $x \notin$ PRIMES, then $\Pr[M \text{ accepts } x] \leq \frac{1}{4}$. From this, we conclude PRIMES $\in$ coRP. Note that this means that the complement of PRIMES,

$$\text{COMPOSITES} = \{\langle n \rangle \mid n \text{ is a composite number,}\}$$

is in the class RP.

Note 2.16 Historically, it has long been an open question whether primality testing can be done in (deterministic) polynomial time. Until in 2002, Agrawal, Kayal and Saxena showed a polynomial time algorithm for testing primality, proving that PRIMES is in P, this algorithm is known as the AKS primality test [1]. □

2.4 The class BPP

The classes RP and coRP are defined with *one-sided* error probabilities. We can also allow *two-sided* error, this gives us the class BPP (bounded-error probabilistic polynomial-time).

Definition 2.17 (The class BPTIME($t(n)$)) Let $t : \mathbb{N} \to \mathbb{R}$ be a function. A language L is in the class BPTIME($t(n)$) if there exists a probabilistic Turing machine M with time complexity $O(t(n))$, such that, for every input x:

- if $x \in L$, then $\Pr[M \text{ accepts } x] \geq \frac{2}{3}$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] \geq \frac{2}{3}$.

□

Again, we define the class BPP similar to how we defined the class RP.

Definition 2.18 (The class BPP) We define

$$\text{BPP} = \bigcup_{k \in \mathbb{N}} \text{BPTIME}(n^k).$$

□

Since we know that the constants used in the definition of RP (and therefore coRP) can be changed without changing the class, we could change the error probability to at most $\frac{1}{3}$. Then, it is easy to see that RP $\subseteq$ BPP and coRP $\subseteq$ BPP.

Just as with RP, the error probability can be reduced. However, error reduction for BPP works slightly different than error reduction for RP. Say we are given a probabilistic Turing machine M and a language L such that for every input x, we have:

- if $x \in L$, then $\Pr[M \text{ accepts } x] \geq c$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] \geq c$,

for some constant $c \in \,]\frac{1}{2}, 1[$. Then we say that the two-sided error probability is $1 - c$. To reduce this error, we construct a new probabilistic Turing machine, M', that runs M a constant number of times, say k times. We let M' accept when more than half of the k runs of M have accepted. Otherwise, when more than half of the k runs of M have rejected, M' rejects. Let us assume that k is an odd number so we do not have to deal with the case where the number of accepting runs is equal to the number of rejecting runs. So, we can describe M' as follows.

"On input x

1. Store two counters for the number of times that M accepted and rejected respectively.
2. Repeat k times:

Run 1	Run 2	Run 3	Probability
accept	accept	accept	$\frac{27}{64}$
accept	accept	reject	$\frac{9}{64}$
accept	reject	accept	$\frac{9}{64}$
accept	reject	reject	$\frac{3}{64}$
reject	accept	accept	$\frac{9}{64}$
reject	accept	reject	$\frac{3}{64}$
reject	reject	accept	$\frac{3}{64}$
reject	reject	reject	$\frac{1}{64}$

Table 2.1: Probabilities for each possible outcome of 3 runs of M.

- Run M on input x.
- If M accepted, increment the counter for the number of accepted runs, and likewise when M rejected.

3. If the number of accepted runs is greater than the number of rejected runs, accept.
4. Otherwise, reject. "

Example 2.19 Say we have a probabilistic Turing machine M for a language L that is in BPP. Now, imagine that, on a certain input x, the probability that M accepts is $\frac{3}{4}$. Let us look at the probability of accepting after performing two-sided error reduction. We construct a new probabilistic Turing machine M' that runs the original Turing machine, M, 3 times, thus we have $k = 3$. Now, the probability that, for example, the first run of M accepts, the second run rejects and the third run accepts equals $\Pr[M \text{ accepts } x] \cdot \Pr[M \text{ rejects } x] \cdot \Pr[M \text{ accepts } x]$, which is $\frac{9}{64}$. In Table 2.1 we have calculated the probabilities for all these possible outcomes of k runs.

Using this table, we can calculate the probability that, for example, 2 of the k runs accept and one run rejects, to do this we take the sum of the probabilities of such outcomes (see the second, third and fifth row in Table 2.1), which is $\frac{9}{64} + \frac{9}{64} + \frac{9}{64} = \frac{27}{64}$. The probability for each possibility is shown in Table 2.2. Remember that when doing error reduction with a two-sided error, we accept if and only if the number of accepting runs is greater than the number of rejecting runs. Now, we can finally calculate the probability that our new Turing machine, M', accepts x. This is simply the sum of the probabilities of obtaining an outcome where the number of accepting runs is greater than the number of rejecting runs. In this case (see the first two rows of Table 2.2), we have

$$\Pr[M' \text{ accepts } x] = \frac{27}{64} + \frac{27}{64} = \frac{54}{64}.$$

So, if our original error probability was $1 - \frac{3}{4} = \frac{1}{4}$, we reduced the error to $1 - \frac{54}{64} = \frac{10}{64} = \frac{5}{32}$.

□

Note 2.20 We stated that the two-sided error probability can not be greater than $\frac{1}{2}$. To see why this is necessary, let us consider the class of languages that have a probabilistic Turing machine which has error probability $\frac{1}{2}$ on both sides. We call this class $\text{BPP}_{\frac{1}{2}}$, so a language L is in $\text{BPP}_{\frac{1}{2}}$ if there exists a probabilistic Turing machine M, such that, on every input x:

- if $x \in L$, then $\Pr[M \text{ accepts } x] \geq \frac{1}{2}$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] \geq \frac{1}{2}$.

Accepting runs	Rejecting runs	Probability
3	0	$\frac{27}{64}$
2	1	$\frac{27}{64}$
1	2	$\frac{9}{64}$
0	3	$\frac{1}{64}$

Table 2.2: Probabilities for the number of accepting and rejecting runs out of 3 runs of M.

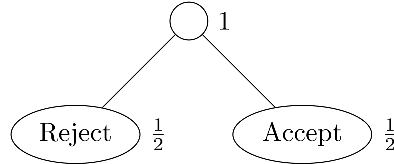


Figure 2.3: A computation tree of a probabilistic Turing machine with two-sided error probability $\frac{1}{2}$.

Now, consider a Turing machine M that, on every input x, has the same computation tree shown in Figure 2.3. As we can see, M performs a coin-flip and either rejects or accepts, depending on the outcome of the coin-flip. So, for every input x, we have $\Pr[M \text{ accepts } x] = \frac{1}{2}$ and $\Pr[M \text{ rejects } x] = \frac{1}{2}$. This means that if we take any language L, then for every input x, we have:

- if $x \in L$, then $\Pr[M \text{ accepts } x] = \frac{1}{2}$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] = \frac{1}{2}$.

Which means that every language is in $\text{BPP}_{\frac{1}{2}}$, even undecidable ones, for example. So, allowing a two-sided error probability of exactly $\frac{1}{2}$ does not make sense. □

2.5 The class ZPP

We said earlier, that without allowing an error probability, a probabilistic, polynomial-time Turing machine gains no power over a deterministic one. However, another way to make a probabilistic Turing machine potentially more powerful is to loosen the time complexity bound. For example, we could allow a probabilistic Turing machine to, sometimes, use more than polynomial time, as long as the average time needed is polynomial. Therefore we define the expected running time of a probabilistic Turing machine.

Definition 2.21 (The expected running time of a probabilistic Turing machine) Given a probabilistic Turing machine M, we denote by $t_M(x)$ the running time of M on input x. Now, $t_M(x)$ is a random variable because the amount of steps needed to halt varies over the different branches in the computation tree. We define $t_M(x, b)$ to be the running time of M on input x when following the branch b, in other words, this is the depth of branch b in the computation tree. The expected running time of M on input x, is defined as

$$\sum_b t_M(x, b) \cdot \Pr[b],$$

where $\Pr[b] = 2^{-k}$ where k is the number of coin-flip steps on the branch b, as we saw earlier. Note that the sum is taken over all branches of the computation tree of M on input x.

Finally, we say M has expected running time $t(n)$ if, for every input x, the expected running time of M on input x is most $t(|x|)$. □

Using this, we define the *zero-error probabilistic time complexity class* $\text{ZTIME}(t(n))$.

Definition 2.22 (The class ZTIME($t(n)$)) Given a function $t : \mathbb{N} \to \mathbb{R}$, a language L is in the class ZTIME($t(n)$) if and only if there exists a probabilistic Turing machine M, with expected running time $t(n)$, such that, for every input x:

- if $x \in L$, then $\Pr[M \text{ accepts } x] = 1$, and
- if $x \notin L$, then $\Pr[M \text{ rejects } x] = 1$.

□

We define ZPP as the class of languages that can be decided in expected polynomial time with zero error probability.

Definition 2.23 (The class ZPP) We define

$$\text{ZPP} = \bigcup_{k \in \mathbb{N}} \text{ZTIME}(n^k)$$

□

Something very noteworthy about this class is shown in the following property.

Property 2.24 ZPP = RP ∩ coRP. □

Proof. We prove the two inclusions.

- We first prove RP ∩ coRP ⊆ ZPP. Let L be a language in RP ∩ coRP. Let M_{RP} be a probabilistic, polynomial-time Turing machine that decides L according to the definition of RP and let M_{coRP} be one that decides L according to the definition of coRP. We construct a probabilistic Turing machine M_{ZPP}, that works by alternating between running M_{RP} and M_{coRP} until either M_{RP} accepts or M_{coRP} rejects, as shown below.

  "On input x:

  1. Keep repeating:

     1.1. Run M_{RP} until it halts. If it accepted, then accept.

     1.2. Run M_{coRP} until it halts. If it rejected, then reject.

     "

  Clearly, M_{ZPP} has zero error probability. Now, M_{RP} and M_{coRP} have time complexity that is at most a polynomial in $|x|$, say this is $t(|x|)$. The probability that M_{ZPP} has not halted after running both machines for $k \cdot t(|x|)$ steps, is at most $\left(\frac{1}{2}\right)^k$. The expected amount of steps that M_{ZPP} has to simulate is thus at most

  $$\sum_{k=1}^{\infty} 2kt(|x|) \cdot \left(\frac{1}{2}\right)^k = 2t(|x|) \cdot \sum_{k=1}^{\infty} k \left(\frac{1}{2}\right)^k$$

  Now, we prove that

  $$\sum_{k=1}^{\infty} k \left(\frac{1}{2}\right)^k = 2.$$

  Consider the function

  $$f(x) = \sum_{k=1}^{\infty} x^k,$$

  if $|x| < 1$ we get a geometric series and thus

  $$f(x) = \frac{1}{x-1} - 1 = \frac{x}{1-x}.$$

Now, if we multiply x with the derivate of f, we get

$$x \cdot f'(x) = x \cdot \sum_{k=1}^{\infty} k \cdot x^{k-1} = \sum_{k=1}^{\infty} k \cdot x^k.$$

But since $f(x) = \frac{x}{1-x}$, we have:

$$x \cdot f'(x) = x \cdot \frac{1}{(1-x)^2}.$$

If we take $x = \frac{1}{2}$, then we get the desired result:

$$x \cdot f'(x) = \sum_{k=1}^{\infty} k \left(\frac{1}{2}\right)^k = \frac{1}{2} \cdot \frac{1}{\left(\frac{1}{2}\right)^2} = 2.$$

So, M_{ZPP} has to simulate at most $4t(|x|)$ steps, which is a polynomial in $|x|$.

- Now we prove ZPP $\subseteq$ RP $\cap$ coRP. Let L be a language in ZPP and let M_{ZPP} be a probabilistic Turing machine for L with expected polynomial time, and zero error probability. Let $t(|x|)$ be the expected running time of M_{ZPP} on input x, thus $t(|x|)$ is a polynomial in $|x|$. We construct a probabilistic Turing machine M_{RP}, that works as follows.

  "On input x:

  1. Run M_{ZPP} on x for $2t(|x|)$ steps.
  2. If M_{ZPP} has halted, then we accept if it accepted and reject if it rejected.
  3. Otherwise, if M_{ZPP} has not halted, we reject. "

  Clearly, M_{RP} runs in polynomial time. Also, when given an input $x \notin L$, M_{RP} will reject x with probability 1. Now, for the error probability, we can use Markov's inequality (see Lemma 2.25 below), which implies that the probability that M_{ZPP} runs for longer than $2t(|x|)$ steps is at most $\frac{1}{2}$.

  Lemma 2.25 (Markov's inequality) If X is a non-negative random variable and $a > 0$, then

  $$\Pr[X \geq a \cdot \mathrm{E}[X]] \leq \frac{1}{a},$$

  where $\mathrm{E}[X]$ denotes the expected value of the random variable X. □

  So, the probability that x is rejected when $x \in L$ is at most $\frac{1}{2}$. This proves $L \in$ RP and thus ZPP $\subseteq$ RP. It is easy to see that ZPP is closed under complementation, and thus $\overline{L} \in$ ZPP $\subseteq$ RP. This implies that $L \in$ coRP and concludes the proof.

□

Chapter 3

Interactive proof systems

We have seen in Chapter 1 that every language in NP can be efficiently (i.e., in polynomial time) verified when given access to a certificate (or proof of membership). In this chapter, we extend this idea of proving membership to a verifier. This way, we explore the possibility of more powerful *interactive proof systems* where we imagine a prover and a verifier that can interact with each other by sending messages. With these proof systems, we define the class IP. As proved by Shamir in 1992 [17], we show that IP equals PSPACE. Since currently it is not known whether NP is a proper subset of PSPACE, there exist languages that have such interactive proof systems while it is not known whether they are in NP. This chapter is based on Chapter 9 from the textbook of Goldreich [10] and on Chapter 8 from the textbook of Arora and Barak [4].

3.1 The class NP as a proof system

We know from Chapter 1 that the class NP can be defined using a polynomial-time verifier that is given a certificate. In this section, we restate this definition with a slightly different notation. This notation will be useful later on.

A proof system consists of two entities: a prover and a verifier. We will say these are both Turing machines, however one could also think of them as two humans, for example, one trying to convince the other of a certain claim. Here, such claims will always be of the same form, being that a certain string is an element of a certain language. For example, a claim could be that some propositional formula ϕ is satisfiable (i.e., the representation of ϕ is an element of the language containing the representations of all satisfiable formulas). It is important that the verifier and prover can communicate with each other. Therefore, we allow them to send *messages* to each other. Messages are strings over the alphabet we are using (usually that is the binary alphabet $\{0,1\}$). Of course, using strings over a finite alphabet, we can encode all sorts of objects, like numbers, graphs, functions, etc. In this chapter, when we describe an interactive proof system, the verifier will expect the messages it receives to have a certain form. When a prover sends a message to the verifier that is not in the expected form, the verifier should simply reject.

In this section, we only allow one message to be sent from the prover to the verifier. We denote the message (a message is a string) sent by prover P on input x by $P(x)$. After the verifier receives a message from the prover, it can choose to accept or reject. When a verifier V accepts after interaction with a prover P on input x, we write '$V \leftrightarrow P$ accepts x'. Similarly, when a verifier V rejects after interaction with a prover P on input x, we write '$V \leftrightarrow P$ rejects x'.

We can now reformulate the definition of NP using this notation.

Definition 3.1 (NP as a proof system) A language L is in the class NP exactly when there exists a polynomial-time verifier V and prover P such that for every input x:

- if $x \in L$ then $V \leftrightarrow P$ accepts x, and

- if $x \notin L$ then for every prover P', $V \leftrightarrow P'$ rejects x.

□

Note 3.2 As seen in the definition above, when defining a proof system, we do so from the perspective of the verifier. We make no assumptions about the prover and thus the prover is computationally unbounded, unlike the verifier who must have a polynomial time complexity. □

Example 3.3 Let us look at a proof system for 3SAT. We know from Chapter 1 that a satisfying assignment can be used as a certificate (or proof of membership) for 3SAT. So, in a proof system, the prover will, given a 3CNF formula ϕ, simply send a satisfying assignment (if there exists one) to the verifier. Then the verifier can check if this assignment is actually satisfying and accept or reject accordingly.

We now take a concrete example, say $\phi = (\neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$. The prover, trying to convince the verifier that ϕ is satisfiable, will send a satisfying assignment. In this case, this could be the assignment that sets x_1 to false, x_2 to true and x_3 to false. Then finally, the verifier can see that ϕ is indeed satisfiable. This process is visualised in Figure 3.1.

Let us consider the case where we are given a unsatisfiable formula, say $\phi = (\neg x_1) \wedge (x_1)$. Then, when a prover sends an assignment, the assignment will not satisfy ϕ and the verifier will reject. Because the verifier expects a variable assignment, when a prover sends a message that is not a variable assignment, the verifier will always reject. Thus, there is no prover that could convince the verifier that ϕ is satisfiable. □

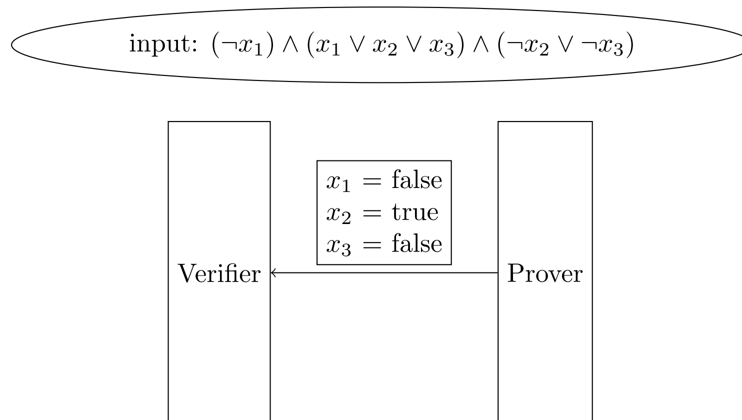


Figure 3.1: Proof system for 3SAT.

3.2 The class IP

3.2.1 Deterministic interactive proof systems

To make a proof system more interactive we allow communication to happen in *rounds*. In the first round, the prover sends a message to the verifier. Then, in the second round, the verifier sends a message to the prover. Then the prover will send a message again, and this process continues until the verifier decides to accept or reject. We allow both the prover and the verifier to read the complete message history when computing the next message to send. The notation is as follows, say a verifier, V, sends a message m_i in the i'th round, then we write $m_i = V(x, m_1, \ldots, m_{i-1})$ where x is the input and $m_1, \ldots m_{i-1}$ is the message history, this means that in the first round, the prover has sent the message m_1, in the second round, the verifier has sent the message m_2, and so on. Now, given verifier V, prover P and input x, we say that $V \leftrightarrow P$ accepts x if there exists a message history $m_1, \ldots, m_k$ such that:

- for every even $i \leq k$, $V(x, m_1, \ldots, m_{i-1}) = m_i$, and

- for every odd $i \leq k$, $P(x, m_1, \ldots, m_{i-1}) = m_i$, and
- after k rounds, the verifier V accepts.

Note 3.4 When we say a verifier V in an interactive proof system has polynomial time complexity, we mean that $V(x, m_1, \ldots, m_i)$ is computable in time bounded by a polynomial in the size of the input, $|x|$. □

We define the class DIP as the class of languages that have a deterministic interactive proof system.

Definition 3.5 (The class DIP) A language L is in the class DIP exactly when there exists an interactive polynomial-time verifier V and prover P such that for every input x:

- if $x \in L$ then $V \leftrightarrow P$ accepts x, and
- if $x \notin L$ then for every prover P', $V \leftrightarrow P'$ rejects x.

□

Example 3.6 As an example, let us look at a proof system with multiple rounds for the 3SAT language. We can let the verifier ask the prover for a truth value of a variable each time the verifier sends a message. Then the prover answers with "true" or "false". Once the verifier has obtained a truth value for every variable, the verifier can check if this is a satisfying assignment. If this is the case, the verifier will accept, otherwise it will reject.

Next, let us show such an interaction for the formula used in Example 3.3, $\phi = (\neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$. Since in our definition the prover sends the first message, but we want the verifier to ask questions to the prover, so we simply let the prover send the empty string ε in the first round, or $m_1 = P(\langle\phi\rangle) = \varepsilon$. In the second round, it is the verifiers turn and let us say the verifier wants to ask a truth value for the variable x_1. To do this, we let the verifier simply sent the name (using some representation) of that variable, thus $m_2 = V(\langle\phi\rangle, \epsilon) = \langle x_1\rangle$. Now the prover has to answer with a truth value, and since the prover is trying to convince the verifier of the fact that ϕ is satisfiable, the prover should use truth values from a satisfying assignment. We use the same assignment as in Example 3.3, and thus $m_3 = P(\langle\phi\rangle, \epsilon, \langle x_1\rangle) =$ "false". And this process is repeated for the other variables, the full interaction is shown in Figure 3.2. Finally, the verifier will have received the full assignment and can check that ϕ is indeed satisfiable □

The interactive proof system seen in Example 3.6 does not seem very useful, because, as we know, 3SAT can be verified using a proof system with just a single round, so we do not need interaction for the 3SAT language. Now, the question arises: are there cases where interaction is useful? This would mean that interactive proof systems with multiple rounds can recognize more languages than proof systems with just one round. As it turns out, this is not the case, as seen in the following property.

Property 3.7 DIP = NP □

To realize this, we have to remember that we put no bounds on the time and space complexity of the prover. This means that, given an interactive proof system with multiple rounds, one can construct a proof system with only one round by letting the prover calculate all the messages that the verifier would send beforehand by simulating the verifier, and then sending all questions and answers in a single message. Clearly, the proof system with one round recognizes the same language as the one with multiple rounds. So, adding interaction on its own does not make proof systems more powerful.

3.2.2 Probabilistic interactive proof systems

In our search for more powerful proof systems let us consider probabilistic verifiers. When using a probabilistic verifier, we cannot simply convert a k-round interactive proof system into one with a single round in the same way as above. This is because the prover would have to answer all

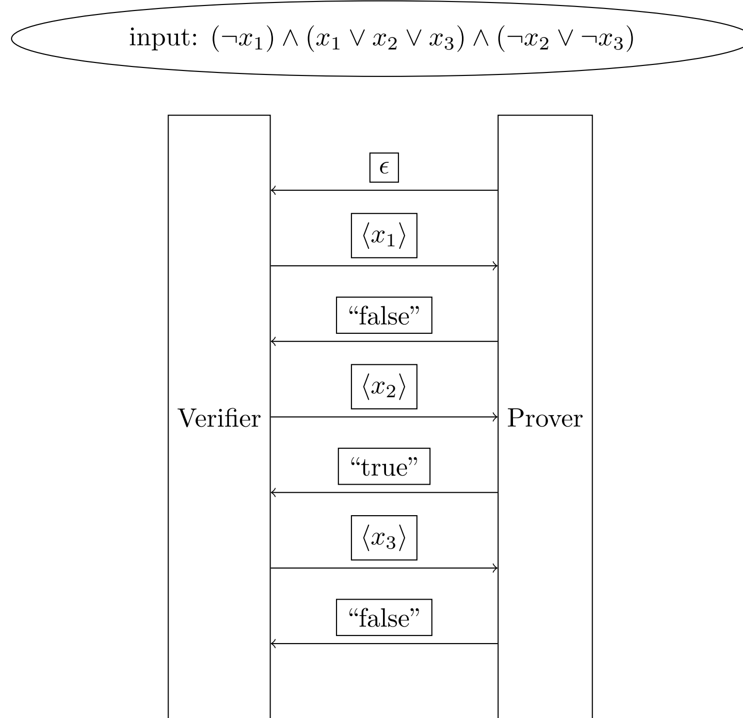


Figure 3.2: Interactive proof system for 3SAT.

possible questions the verifier could ask. Since the verifier runs is polynomial time, it can also use a polynomial amount of random coins. This means the amount of possible questions (and answers) could be exponential in the size of the input. A polynomial-time verifier would not be able to read an exponential amount of answers.

Using probabilistic verifiers requires us to adjust our notation again. Now, we denote the probability that a verifier V accepts after interaction with a prover P on input x by $\Pr[V \leftrightarrow P \text{ accepts } x]$. And the probability that verifier V sends the message m in the i-th round is denoted by $\Pr[V(x, m_1, \ldots, m_{i-1}) = m]$.

Next, we define the class $\mathrm{IP}(k)$, consisting of languages that have k-round interactive proof systems.

Definition 3.8 (The class IP(k)) Given a function $k : \mathbb{N} \to \mathbb{N}$, a language L is in the class $\mathrm{IP}(k)$ if and only if there exists a probabilistic polynomial-time verifier V and prover P such that for every input x:

- an interaction of the verifier, V, with the prover, P, takes at most $k(|x|)$ rounds, and
- if $x \in L$ then $\Pr[V \leftrightarrow P \text{ accepts } x] \geq \frac{2}{3}$, and
- if $x \notin L$ then for every prover P', $\Pr[V \leftrightarrow P' \text{ accepts } x] \leq \frac{1}{3}$.

□

One could ask why the constants $\frac{2}{3}$ and $\frac{1}{3}$ are used. In fact these constants are quite arbitrary. Let us define the class $\mathrm{IP}_{c,s}(k)$.

Definition 3.9 Given a function $k : \mathbb{N} \to \mathbb{N}$, a language L is in the class $\mathrm{IP}(k)$ if and only if there exists a probabilistic polynomial-time verifier V and prover P such that for every input x:

- an interaction of the verifier, V, with the prover, P, takes at most $k(|x|)$ rounds, and
- if $x \in L$ then $\Pr[V \leftrightarrow P \text{ accepts } x] \geq 1 - c$, and

- if $x \notin L$ then for every prover P', $\Pr[V \leftrightarrow P' \text{ accepts } x] \leq s$.

□

In the definition above, we say c is the completeness error and s is the soundness error. So, in our definition of $\mathrm{IP}(k)$, both the completeness error and the soundness error are $\frac{1}{3}$.

Property 3.10 For all constants $c, s < \frac{1}{2}$, $\mathrm{IP}_{c,s}(k) = \mathrm{IP}(k)$. □

In other words, a language that has a k-round interactive proof system with completeness error c and soundness error s, also has a k-round interactive proof system with both completeness error and soundness error $\frac{1}{3}$, for all constants c, s greater than $\frac{1}{2}$.

Proof. We proof the two inclusions.

1. We start with $\mathrm{IP}_{c,s}(k) \subseteq \mathrm{IP}(k)$. Say we are given a language $L \in \mathrm{IP}_{c,s}(k)$. This means L has an interactive proof system with completeness error c and soundness error s with, say, a verifier V and prover P. Using this, we construct a new interactive proof system with with both completeness error and soundness error $\frac{1}{3}$ with a verifier V' and P'. The new proof system works by running the original proof system a number of times, say n times. To keep the number of rounds the same, we will not run the proof system n times consecutively (this would require $n \cdot k$ rounds) but in parallel, this means a message in the new proof system will consist of n messages used to simulate the original proof system. Say the input is x. In the first round, the prover P' computes $P(x)$ to obtain the message m_1 and sends it to the verifier. In the second round, the verifier V' runs $V(x, m_1)$ independently n times to obtain messages $m_{2,1}, m_{2,2}, \ldots, m_{2,n}$ and sends the message $m_2 = \langle m_{2,1}, m_{2,2}, \ldots, m_{2,n} \rangle$ to the prover. Then in the third round, the prover runs $P(x, m_1, m_{2,i})$ for each i from 1 to n to obtain messages $m_{3,1}, \ldots, m_{3,n}$ and sends the message $m_3 = \langle m_{3,1}, \ldots, m_{3,n} \rangle$ to the verifier. This continues, until the final round k. After that, the verifier V' will run V on input x and message history $m_1, m_{2,i} \ldots, m_{k,i}$ for each i from 1 to n and count the number of times V accepts and the number of times V rejects, out of the i runs. Finally if the number of accepting runs is greater than the number of rejecting runs, V' will accept. Otherwise, if the number of rejecting runs is greater than the number of accepting runs, V' will reject. For simplicity, let us assume that n is odd to avoid the case where the number of accepting runs equals the number of rejecting runs.

   Now, let us analyze the completeness and soundness error of the constructed proof system.

   - Say $x \in L$, we give an upper bound for the probability that V' rejects x. By definition we know that the probability that V is is wrong is at most c, in other words, $\Pr[V \text{ rejects } x] \leq c$. Now, let us look at the probability that V' is wrong (that is, rejects x). The constructed verifier V' is wrong when it rejects r times and accepts a times out of n times such that $r > a$. The probability of this happening with a accepting runs and r rejecting runs is at most $(1-c)^a \cdot c^r$. Let us call an outcome of n runs a rejecting outcome when there are more rejecting runs then accepting runs of the n runs. For a rejecting outcome, we have $(1-c)^a \cdot c^r < c^n$, because $1 - c < c$ and $a + r = n$ (we assumed n is odd). If we take the sum of this probability over all rejecting outcomes, we get an upper bound for the probability that V' rejects. The number of rejecting outcomes is obviously less than the number of total outcomes possible, which is 2^n, thus

     $$\Pr[V' \text{ rejects } x] \leq 2^n \cdot c^n = (2 \cdot c)^n.$$

     Now, we can find a value for n such that $(2 \cdot c)^n \leq \frac{1}{3}$, that is when $n \geq \log_{2c}(\frac{1}{3})$.

   - For the other case, when $x \notin L$ we can use the same reasoning, just using outcomes with more accepting runs than rejecting runs instead of the other way around. Then we get the following inequality,

     $$\Pr[V' \text{ accepts } x] \leq (2 \cdot s)^n.$$

     And we can take $n \geq \log_{2s}(\frac{1}{3})$ like before.

Finally if we take $n \geq \max(\log_{2c}(\frac{1}{3}), \log_{2s}(\frac{1}{3}))$, we get the desired completeness and soundness error, and this concludes the proof of the first inclusion.

2. The other inclusion, $\text{IP}_{c,s}(k) \subseteq \text{IP}(k)$, can be proven in the same way as above, using parallel repetition, we only have to swap the constant c with $\frac{1}{3}$ and s with $\frac{1}{3}$ as well. Then we have to use n parallel runs where $n \geq \max(\log_{2/3}(c), \log_{2/3}(s)$ to obtain both completeness and soundness error at most $\frac{1}{3}$.

□

We also define the class IP, which consists of languages that have interactive proof systems with a polynomial amount of rounds.

Definition 3.11 (The class IP) We define

$$\text{IP} = \bigcup_{c \in \mathbb{N}} \text{IP}(n^c).$$

□

From the examples above, we can see that NP ⊆ IP. This also follows from the definition because NP is defined using a deterministic verifier, which is a special case of a probabilistic verifier (with both completeness error and soundness error equal to 0) and there is only round of interaction, so NP ⊆ IP(1) ⊆ IP.

Example 3.12 In this example, we show an interactive proof system for GNI, a language not known to be in NP. We define the language GNI as follows,

$$\text{GNI} = \{\langle G_0, G_1 \rangle \mid \ G_0 \text{ and } G_1 \text{ are non-isomorphic graphs}\}.$$

We say two graphs, G_0 and G_1, are isomorphic (we denote this by $G_0 \cong G_1$) if and only if there exists a bijection $f : V(G_0) \to V(G_1)$ such that for all $u, v \in V(G_0)$ it holds that $(u, v) \in E(G_0)$ iff $(f(u), f(v)) \in E(G_1)$. Here, $V(G)$ is the set of vertices of graph G and $E(G)$ is the set of edges of graph G. We will use $\text{ISO}(G)$ to denote the set of graphs isomorphic to graph G that have the vertex set $\{1, \ldots, |V(G)|\}$, or

$$\text{ISO}(G) = \{H \mid G \cong H \text{ and } V(H) = \{1, \ldots, |V(G)|\}\}.$$

We now describe the proof system.

1. The verifier chooses a random $a \in \{0, 1\}$, chooses a graph $H \in \text{ISO}(G_a)$ at random, and sends $\langle H \rangle$ to the prover.
2. After receiving $\langle H \rangle$, the prover has to figure out which of the two graphs (G_0 or G_1) was used to construct H. If the prover chooses the graph G_b, he will send $\langle b \rangle$ to the verifier.
3. When the verifier receives $\langle b \rangle$, he will accept when $a = b$, and reject otherwise.

Note that the verifier can chose a random graph from $\text{ISO}(G_b)$ by choosing a random bijection $f : V(G_b) \to \{1, \ldots, |V(G_b)|\}$ and then constructing the graph H such that $V(H) = \{1, \ldots, |V(G_b|\}$ and $E(H) = \{(f(u), f(v)) \mid (u, v) \in E(G_b)\}$.

Let us now show that this proof system satisfies the definition.

If $G_0 \ncong G_1$, then $\text{ISO}(G_0)$ and $\text{ISO}(G_1)$ are disjunct sets and thus the prover can simply send $a = 0$ when $G_0 \cong H$ and $a = 1$ when $G_1 \cong H$. Because the prover can always find the correct answer, the verifier will accept with probability 1.

If $G_0 \cong G_1$, then $\text{ISO}(G_0) = \text{ISO}(G_1)$ and because of this, the prover cannot distinguish between graphs isomorphic to G_0 and graphs isomorphic to G_1, so it can not do anything more than guess the answer. This means the prover will be wrong with probability at least $\frac{1}{2}$. Now, the verifier will accept with probability $\frac{1}{2}$, while in the definition we require an error of at most $\frac{1}{3}$. We can fix this

by repeating the protocol one more time (this makes the proof system require 4 rounds in total), then the error becomes $\left(\frac{1}{2}\right)^2 = \frac{1}{4} < \frac{1}{3}$.

Finally, we note that the verifier can be implemented in probabilistic polynomial time. From this example we conclude that GNI $\in$ IP(4). □

3.3 IP = PSPACE

3.3.1 IP ⊆ PSPACE

In this subsection, we prove that every language decidable by an interactive proof system can also be decided by a polynomial space (deterministic) Turing machine. This proof is inspired by the one found in the textbook of Sipser [19].

Theorem 3.13 IP $\subseteq$ PSPACE. □

Proof. Given an interactive verifier V for a language $L \in$ IP, we construct a polynomial space Turing machine M. When given an input x, M will calculate the maximum probability that V will accept with after interaction with some prover P, or

$$p = \max_P \Pr[V \leftrightarrow P \text{ accepts } x].$$

By Definition 3.8 we know that:

- if $p \geq \frac{2}{3}$, then $x \in L$, and
- otherwise $p \leq \frac{1}{3}$ and $x \notin L$.

So, calculating p allows M to effectively decide membership of x in L. We show how to compute p recursively on the message history, therefore we define

$$p_h = \max_P \Pr[V \leftrightarrow P \text{ accepts } x \text{ starting with message history } h].$$

Where $h = m_1, \ldots, m_i$ is a history of messages with $|h| = i$ the number of messages, and

$$\Pr[V \leftrightarrow P \text{ accepts } x \text{ starting with message history } h]$$

is the probability that after the message history h has occurred, the verifier V will accept after interacting with the prover P on input x. We also define h_0 to be the empty message history, thus $p = p_{h_0}$. We denote the extension of message history h with message m by $h, m = m_1, \ldots, m_i, m$. Now, the following equations hold:

- when $|h|$ is even: $p_h = \max_m p_{h,m}$, and
- when $|h|$ is odd: $p_h = \sum_m \Pr[V(x,h) = m] \cdot p_{h,m}$.

Above, $V(x,h)$ is the message that the verifier will send to the prover after message history h has occurred, i.e., when $h = m_1, \ldots, m_i$, then $V(x,h) = V(x, m_1, \ldots, m_i)$.

The first equation holds because when $|h|$ is even, the prover will send a message to the verifier so we want to find the best message the prover can send, which means finding an m that maximizes $p_{h,m}$. The second equation occurs when the verifier sends a message to the prover, here we have to account for all possible messages the verifier could send and thus calculate a weighted average of probabilities.

Our Turing machine M will compute p recursively, and because there at most a polynomial amount of messages sent between the verifier and prover, the number of levels in this recursion will be at most polynomial in the size of the input. On each level, M only requires a polynomial amount of space, to show this we distinguish between the two cases when calculating p_h.

- When $|h|$ is even, we need to compute $\max_m p_{h,m}$. To do this, we iterate over all possible messages. We can bound the length of the messages by a polynomial in the size of the input because a polynomial-time verifier cannot read longer messages. We only need to keep a single message in memory while iterating over the messages.
- When $|h|$ is odd, we need to compute $\sum_m \Pr[V(x,h) = m] \cdot p_{h,m}$. In this case, we iterate over all possible outcomes of coin-flips that the verifier does. Because the verifier runs in polynomial time, the number of coin-flip steps is also bounded by a polynomial in the size of the input. Again, we only keep a single outcome of the coin-flip steps in memory while iterating over the outcomes of random coins.

Finally, since the amount of recursive levels is at most polynomial and the amount of space used by each level is polynomial, M can compute p using polynomial space. Thus IP $\subseteq$ PSPACE. □

3.3.2 PSPACE ⊆ IP

Now we proof the other inclusion, this result was first prover by Shamir in 1992 [17]. However, we show a simplified version by Shen [18], that is also found in the textbook of Goldreich [10] and the textbook of Arora and Barak [4].

Theorem 3.14 (Shamir, 1992 [17]) PSPACE $\subseteq$ IP □

Arithmetization of boolean formulas

Central to the proof is a technique called *arithmetization* of boolean formulas. For a boolean formula ϕ with variables $x_1, \ldots x_n$ we can construct a polynomial function $f_\phi : \mathbb{N}^n \to \mathbb{N}$, with corresponding variables $y_1, \ldots, y_n$, that has the following property. Given an assignment $a_1, \ldots a_n$ to variables $x_1, \ldots x_n$: if the assignment satisfies ϕ, then $f_\phi(t(a_1), \ldots, t(a_n)) > 0$, and if ϕ is not satisfied by the assignment, $f_\phi(t(a_1), \ldots, t(a_n)) = 0$, where t is a function that maps true to 1 and false to 0.

The conversion of ϕ to f works as follows:

- A positive literal $l = x_i$ is converted to $f_l = y_i$.
- A negative literal $l = \neg x_i$ is converted to $f_l = 1 - y_i$.
- A clause with n literals, $c = (l_1 \vee \cdots \vee l_n)$ is converted to $f_c = \sum_{i=1}^n f_{l_i}$.
- Finally, a conjunction of clauses $c_1 \wedge \cdots \wedge c_m$ is converted to $\prod_{i=1}^m f_{c_i}$.

To see that this conversion indeed satisfies the above property:

- It holds trivially for literals.
- When a clause $c = (l_1 \vee \ldots \vee l_n)$ is satisfied, one the literals must be true, thus one or more of $f_{l_1}, \ldots, f_{l_n}$ must be greater than 0 and the other equal 0, so $f_c > 0$. Otherwise, when c is not satisfied, than none of the literals are satisfied, so all of $f_{l_1}, \ldots, f_{l_n}$ are 0 and $f_c = 0$.
- When a conjunction of clauses $c_1 \wedge \cdots \wedge c_m$ is satisfied, all of the clauses are satisfied ($f_{c_i} > 0$ for all c_i) and $\prod_{i=1}^m > 0$. When a conjunction is not satisfied, there exists at least one clause c_i that is not satisfied, therefore $f_{c_i} = 0$ and $\prod_{i=1}^m = 0$.

Let us give an example, say $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$, then we get

$$f(y_1, y_2, y_3) = (y_1 + y_2 + (1 - y_3)) \cdot ((1 - y_1) + (1 - y_2) + y_3).$$

The truth table of ϕ, combined with the outcome of f is shown in Table 3.1. As we can see, the outcome of f_ϕ is 2 when ϕ is satisfied and 0 otherwise.

x_1	x_2	x_3	ϕ	f_ϕ
false	false	false	true	2
false	false	true	false	0
false	true	false	true	2
false	true	true	true	2
true	false	false	true	2
true	false	true	true	2
true	true	false	false	0
true	true	true	true	2

Table 3.1: The truth table of $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ combined with the outcome of f_ϕ.

coNP $\subseteq$ IP

We start by proving a weaker theorem. Later on, we extend this proof to the stronger version.

Theorem 3.15 coNP $\subseteq$ IP. □

Proof. We show an interactive proof system for $\overline{\text{3SAT}}$, which is coNP-complete.[1]

First, we show a property of arithmetization of boolean formulas. Given a boolean formula ϕ, we can convert it into the polynomial function f_ϕ. We know that, given an assignment $a_1, \ldots, a_n \in \{\text{true}, \text{false}\}$ to the variables $x_1, \ldots, x_n$ of ϕ, then ϕ is not satisfied by assignment if and only if $f_\phi(t(a_1), \ldots, t(a_n)) = 0$. So, if for for every assignment $b_1, \ldots b_n \in \{0,1\}$ to the variables $y_1, \ldots, y_n$ of f_ϕ, $f_\phi(b_1, \ldots, b_n) = 0$, then ϕ is not satisfiable. Because f_ϕ is always positive when given inputs in $\{0,1\}$, summing f_ϕ over all assignments to $y_1, \ldots, y_n$ is 0 if and only if ϕ is unsatisfiable. So, ϕ being not satisfiable is equivalent to the following equality:

$$\sum_{y_1 \in \{0,1\}} \cdots \sum_{y_n \in \{0,1\}} f_\phi(y_1, \ldots, y_n) = 0.$$

So, in this proof we actually show an interactive proof for claims like the equality above. Note that a polynomial-time verifier could not evaluate the left hand side of above equality because this would take an exponential amount of time in n, because there are an exponential amount of terms. Let us define the function g_i, obtained by stripping off the first i summations, as

$$g_i(y_1, \ldots, y_i) = \sum_{y_{i+1} \in \{0,1\}} \cdots \sum_{y_n \in \{0,1\}} f_\phi(y_1, \ldots, y_n).$$

We can see that

$$g_i(y_1, \ldots, y_i) = g_{i+1}(y_1, \ldots, y_i, 0) + g_{i+1}(y_1, \ldots, y_i, 1).$$

Now we can describe the proof system, this will happen in $n+1$ phases, where n is the number of variables of ϕ. The first phase looks as follows:

1. The prover sends the representation of $h_1 = g_1$ to the verifier. We note that g_1 is a univariate polynomial function with degree m, so can be represented in the way polynomials are usually written, that is,
$$g_1(y_1) = a_1 y_1^m + \cdots + a_n y_1 + \cdots a_{n+1},$$
where $a_1, \ldots a_{n+1}$ are the coefficients.

[1] Note that technically, $\overline{\text{3SAT}}$ not only contains 3CNF formulas that are not satisfiable, but also all strings that are not 3CNF formulas. However, we ignore this because a verifier can easily check that the input is a 3CNF formula and accept if this is not the case.

2. The verifier computes the value of $h_1(0) + h_1(1)$ (this can be done in polynomial time using the representation shown in the previous point). If $g_1(0) + g_1(1) \neq 0$, the verifier will reject, otherwise it will continue.

3. Finally, the verifier sends the prover a randomly chosen integer $r_1 \in [0, 9m^2]$.

In phase i, where $0 < i \leq n$:

1. The prover sends h_i, where $h_i(y_i) = g_i(r_1, \ldots, r_{i-1}, y_i)$, to the verifier.

2. The verifier computes $h_i(0) + h_i(1)$ and rejects if the outcome does not equal $h_{i-1}(r_{i-1})$.

3. The verifier sends the prover a randomly chosen integer $r_i \in [0, 9m^2]$.

Finally, in phase $n + 1$:

1. The verifier checks that $h_n(r_n) = f_\phi(r_1, \ldots, r_n)$. If true, the verifier accepts and rejects otherwise.

Let us now verify that this proof system is correct. When the prover follows the protocol described above and ϕ is not satisfiable, then clearly the verifier will always accept. In other words, there exists a prover P such that for every ϕ, if $\langle\phi\rangle \in \overline{\text{3SAT}}$ then $\Pr[V \leftrightarrow P \text{ accepts } \langle\phi\rangle] = 1$.

But what happens when ϕ is satisfiable (and thus not in $\overline{\text{3SAT}}$)? If the prover follows the protocol, then the verifier will reject in the first phase because $g_1(0) + g_1(1) \neq 0$). However, we can not assume that the prover will follow the protocol. The verifier has to reject with probability at least $\frac{2}{3}$ when interacting with any prover. So, say the prover does not follow the protocol and sends some h_1 different from g_1 in the first phase, such that $h_1(0) + h_1(1) = 0$. After the verifier has chosen r_1, we can distinguish two cases:

- When $h_1(r_1) = g_1(r_1)$, then the prover can send $h_2(y_2) = g_2(r_1, y_2)$ in the second phase and just continue the protocol as described above. This will make the verifier accept (wrongly).
- When $h_1(r_1) \neq g_1(r_1)$, then if the prover sends $h_2 = g_2(r_1, y_2)$ in the second phase, the verifier will reject because

$$h_2(0) + h_2(1) = g_2(r_1, 0) + g_2(r_1, 1) = g_1(r_1) \neq h_1(r_1).$$

  So, when trying to make the verifier accept, the prover must send some h_2, different from g_2, such that $h_2(0) + h_2(1) = h_1(r_1)$.

Luckily, the probability of the first case occurring is small. Consider the function $d(y_1) = h_1(y_1) - g_1(y_1)$, since h_1 and g_1 both have degree at most m (when h_1 has degree larger than m, the verifier should reject), d also has degree at most m. This implies there are at most m values for y_1 such that $d(y_1) = 0$ (remember we are assuming $h_1 \neq g_1$), and definitely at most m values in $\{0, \ldots, 9m^2\}$. Thus, when randomly choosing $r_1 \in \{0, 9m^2\}$, we have

$$\Pr[h_1(r_1) = g_1(r_1)] \leq \frac{m}{9m^2}.$$

In general, if the functions $h_i(y_i)$ and $g_i(r_1, \ldots, r_{i-1}, y_1)$ are different, then

$$\Pr[h_i(r_i) = g_i(r_1, \ldots, r_i)] \leq \frac{m}{9m^2},$$

when randomly choosing $r_i \in \{0, \ldots 9m^2\}$. Here, $r_1, \ldots r_{i-1}$ are already chosen and thus constant in this context. Let us denote E_i by the event that $h_i(r_i) = g_i(r_1, \ldots, r_{i-1})$ in phase i, then

$$\Pr[E_i] \leq \frac{m}{9m^2},$$

given that the functions h_i and g_i are different. Now, if we denote the event that the verifier accepts by A, then

$$\Pr[A] = \Pr[E_1] + (1 - \Pr[E_1]) \cdot \Pr[E_2] + \cdots + (1 - \Pr[E_1]) \cdots (1 - \Pr[E_{n-1}]) \cdot \Pr[E_n].$$

In other words, the verifier will accept when E_1 occurs, or if E_1 does not occur but E_2 does, or if E_1 and E_2 both do not occur but E_3 does occur, and so on. Since, we know that for every i, $\Pr[E_i] \leq \frac{m}{9m^2}$, we can give an upper bound on the probability that the verifier accepts:

$$\Pr[A] \leq \Pr[E_1] + \cdots + \Pr[E_n] \leq n \cdot \frac{m}{9m^2}.$$

Finally, since $n \leq 3m$,

$$\Pr[A] \leq \frac{3m^2}{9m^2} = \frac{1}{3}.$$

This should explain why we chose to sample numbers between 0 and $9m^2$. Of course, we could choose a larger set to sample from, in that case the probability of error becomes smaller but we require more coin-flip steps. On the other hand, by using a smaller set, the acceptance probability becomes higher (although we only gave an upper bound here). □

Example 3.16 Let us look at an example of the proof system shown above. Say $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$, a clearly unsatisfiable formula. The number of clauses, m, is 4 and the number of variables, n, is 2. After arithmetization, we get:

$$f_\phi(y_1, y_2) = (y_1 + y_2) \cdot ((1 - y_1) + y_2) \cdot (y_1 + (1 - y_2)) \cdot ((1 - y_1) + (1 - y_2)).$$

Now, we look at the interaction between the verifier and the prover.

1. The prover starts by sending the function h_1, where
$$h_1(y_1) = g_1(y_1) = \sum_{y_2 \in \{0,1\}} f_\phi(y_1, y_2) = 2y_1^4 - 4y_1^3 - 2y_1^2 + 4y_1.$$
2. When the verifier receives h_1, it checks that $h_1(0) + h_1(1) = 0$ (in our case, this is true).
3. The verifier sends a random number $r_1 \in \{0, \ldots, 9m^2 = 144\}$, as an example let us take $r_1 = 17$.
4. Now, the prover sends the function h_2, where
$$h_2(y_2) = g_2(r_1, y_2) = f_\phi(r_1, y_2) = y_2^4 - 2y_2^3 + 545y_2^2 + 546y_2 + 73440.$$
5. The verifier will check that $h_2(0) + h_2(1) = h_1(r_1)$ (again, this is true in our example).
6. Then the verifier will again choose a random number $r_2 \in \{0, \ldots, 144\}$ (since we are in the last round, the verifier does not need to send r_2 to the prover). Let us take $r_2 = 107$ as an example.
7. Finally, the verifier will accept if $h_2(r_2) = f_\phi(r_1, r_2)$. We can verify that this is the case, thus the verifier will accept.

□

Extending the proof system to TQBF

We show that the proof system for $\overline{\text{3SAT}}$ can be extended to obtain a proof system for TQBF. TQBF is the language containing all true quantified boolean formulas. We assume all quantified boolean formulas are in the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi,$$

where each $Q_i \in \{\exists, \forall\}$ is a quantifier and ϕ is a propositional formula with variables $x_1, \ldots x_n$. This form, where all quantifiers are written at the beginning of the formula, is called the *prenex normal form*. Every quantified boolean formula can be converted into an equivalent formula in prenex normal form. We also consider only consider quantified formulas where the propositional part is in 3CNF, any quantified formula in prenex normal form can be converted into one of which the propositional part is in 3CNF (and the formula itself is still in prenex normal form). Since

TQBF is a PSPACE-complete language, showing that TQBF $\in$ IP, would proof that PSPACE $\subseteq$ IP.

A naive way to extend our proof system is to extend arithmetization to quantified formulas in the way described below.

- A quantifier-free formula ϕ is converted to f_ϕ as before.
- A formula $\forall x_i \phi$ is converted to $\prod_{y_i \in \{0,1\}} f_\phi(y_1, \dots y_i)$.
- A formula $\exists x_i \phi$ is converted to $\sum_{y_i \in \{0,1\}} f_\phi(y_1 \dots y_i)$.

We note that given a quantified boolean formula ψ, the arithmetized formula f_ψ has no free variables and thus evaluates to a constant. However, as before, this constant can not be computed in polynomial time given the arithmetized formula, because there are an exponential amount of terms.

This way of arithmetization works, because ψ is true if and only if $f_\psi > 0$. Now, if we define the function g_1 for a formula $\psi = \forall x_1 \exists x_2 \cdots \forall x_n \phi$ similar to before (that is, by stripping of the first summation or product, in this case, we strip off a product), we get the following,

$$g_1(y_1) = \sum_{y_2 \in \{0,1\}} \cdots \prod_{y_n \in \{0,1\}} f_\phi(y_1, \dots y_n).$$

Remember that the degree (of each variable) of f_ϕ is at most m, where m is the number of clauses in ϕ. Every time a product is applied to f_ϕ, the degree can double, so the degree of g_1 for variable x_1 could become exponentially large in n, to be more specific: up to $2^n \cdot m$ (n is the number of variables). Obviously, a polynomial function with a degree that is exponential in the input size cannot be read by a polynomial-time verifier, because there could be an exponential amount of coefficients. So, this naive way of arithmetization does not give us a proof system for TQBF.

To solve this, we use a *linearization* operation, which given a function $f(y_1, \dots y_k)$, linearization on variable y_i produces a function $f'(y_1, \dots, y_k)$ such that

1. f' is linear (i.e., has degree at most 1) in y_i, and
2. for $y_i \in \{0,1\}$, $f'(y_1, \dots, y_k) = f(y_1, \dots, y_k)$.

In other words, linearization on y_i produces a function, linear in y_i, that is equivalent to the original on inputs in $\{0,1\}$. Applying linearization is quite simple. Given a function $f(y_1, \dots, y_k)$ that we wish to linearize on y_i, then the outcome is

$$f'(y_1, \dots, y_k) = (1 - y_i) \cdot f(y_1, \dots, y_{i-1}, 0, \dots, y_k) + y_i \cdot f(y_1, \dots, y_{i-1}, 1, \dots, y_k).$$

Clearly, f' is linear in y_i, and when $y_i = 0$, then $f'(y_1, \dots, y_k) = f(y_1, \dots, y_{i-1}, 0, y_k)$ and if $y_i = 1$ then $f'(y_1, \dots, y_k) = f(y_1, \dots, y_{i-1}, 1, y_k)$.

So, when performing arithmetization of a $\forall$ quantifier, we first linearize the input function on all free variables and then take the product, just like in the naive way. This way the result is a polynomial with degree at most 2.

Example 3.17 Say we are given a quantified boolean formula,

$$\psi = \forall x_1 \exists x_2 (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2).$$

We show how to arithmetize this formula using the linearization operation as described above. First, consider the propositional part,

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2).$$

We arithmetize this in the usual way,

$$f_\phi(y_1, y_2) = (y_1 + y_2) \cdot ((1 - y_1) \cdot (1 - y_2)) = -y_1^2 - 2y_1y_2 + 2y_1 - y_2^2 + 2y_2.$$

Next, we consider $\psi_1 = \exists x_2 \phi$, this formula can be arithmetized in the naive way (the $\exists$ quantifier did not introduce problems regarding the degree of the outcome).

$$f_{\psi_1}(y_1) = \sum_{y_2 \in \{0,1\}} f_\phi(y_1, y_2) = -2y_1^2 + 2y_1 + 1.$$

Finally we get to $\psi = \forall x_1 \psi_1$, so we first we have to linearize f_{ψ_1} on y_1, let us call the resulting function f'_{ψ_1}.

$$f'_{\psi_1}(y_1) = (1 - y_1) \cdot f_{\psi_1}(0) + y_1 \cdot f_{\psi_1}(1) = (1 - y_1) + y_1 = 1.$$

Coincidentally, the resulting function is constant in y_1, but a constant function is also a linear function. Now, the final step is the actual arithmetization of the $\forall$ quantifier as seen in the naive way.

$$f_\psi = \prod_{y_1 \in \{0,1\}} f'_{\psi_1}(y_1) = 1.$$

Because $f_\psi > 0$, we now know that ψ is a true quantified boolean formula. □

Now, we can modify the proof system for $\overline{\text{3SAT}}$ to one for TQBF by

1. allowing quantified boolean formulas, and
2. in each round, instead of stripping off a summation, strip off one of the possible operations (being either a summation, a product or a linearization). Now, the number of rounds required is at most n^2 with n the number of variables of ψ, because for every $\forall$ quantifier, we require up to n linearization operations.

This concludes the proof of theorem 3.14. More specifically, TQBF $\in$ IP(n^2).

3.4 Public-coin proof systems

In Example 3.12, we showed an interactive proof system for graph non-isomorphism. Looking back, it seems quite important that the outcome of the coin-flips of the verifier can not be seen by the prover. In fact, if the prover could see the outcome of the coin-flips, he could make the verifier accept any input with probability 1. In this section, we consider proof systems where the prover has access to the outcome of the coin-flips of the verifier, such proof systems are called *public-coin proof systems*. These proof systems have an interesting property. Say we let the verifier send, along with a message, the outcome of coin-flips the verifier used, then there is actually no need to send anything but the outcome of the coin-flips because the prover can simply compute the message by himself, given the random coins.

Definition 3.18 An interactive proof system is called a *public-coin proof system* when the messages of the verifier only consist of random bits (i.e., the outcome of coin-flip steps). □

Using this definition, we define a corresponding complexity class, AM(k), first defined by Babai [6] as the class of languages that have so-called Arthur vs. Merlin games (which we call public-coin proofs) where Merlin (the prover) has to convince Arthur (the verifier) of a certain claim.

Definition 3.19 (The class AM(k)) Given a function $k : \mathbb{N} \to \mathbb{N}$, language L is in AM(k) is and only if there exists a public-coin proof system with at most $k(|x|)$ rounds on any input x. □

The following theorem follows directly from the definition, because a public-coin proof system is a special case of an interactive proof system.

Theorem 3.20 For every $k : \mathbb{N} \to \mathbb{N}$, AM($k$) $\subseteq$ IP(k). □

Interestingly, there are also relations between AM(k) and IP(k) in the other direction.

Theorem 3.21 IP $\subseteq \bigcup_{c \in \mathbb{N}}$ AM(n^c). □

In other words, every language in IP has a public-coin proof system with a number of rounds at most polynomial in the size of the input.

Proof. The idea of this proof is that the proof system shown for TQBF in Section 3.3 can easily be transformed into a public-coin proof system. The verifier can just send the random bits used to choose the r_i's and let the prover compute the r_i's, this will not change the probability that the verifier accepts wrongly. So, TQBF $\in$ AM(n^2).

Now, we give the actual proof. Given a language L in IP, we know that L is in PSPACE by Theorem 3.13. We also know that there exists a polynomial-time reduction f of L to TQBF, because TQBF is PSPACE-complete. Because TQBF $\in$ AM(n^2), we can construct a public-coin proof for L using at most $|f(x)|^2$ rounds on any input x. Finally, since $|f(x)|$ is polynomial in $|x|$, $|f(x)|^2$ is also polynomial in $|x|$, so our public coin proof system for L requires at most a polynomial amount of rounds. □

Now, we know that, for example, graph non-isomorphism also has a public-coin proof system. However, this public-coin proof system could require a polynomial amount of rounds, while in Example 3.12, we showed an interactive proof system with only a constant number of rounds. The following theorem is stronger than Theorem 3.21, and implies that there exists a public-coin proof with only a constant number of rounds for graph non-isomorphism, for example.

Theorem 3.22 (Goldwasser and Sipser, 1987 [13]) IP(k) $\subseteq$ AM($k+2$). □

3.5 Multi-prover interactive proof systems

In this chapter, so far, we have shown potentially more powerful proof systems than the ones used to define the class NP. These interactive proofs could be used as an alternative definition to what one might consider to be an efficient proof system. However, we said these interactive proof systems are potentially more powerful than NP proof systems because currently, it is not known if IP = PSPACE or not. In this section, we show a type of proof system that is provably more powerful than NP proof systems. To do this, we add multiple provers to an interactive proof system to obtain a *multi-prover interactive proof system*. In a multi-prover interactive proof system, the verifier will interact with provers $P_1, \ldots P_n$ where in each odd round the verifier sends a (possibly different) message to every prover and in even rounds each prover sends a message back to the verifier. The amount of provers, l, must be bounded by a polynomial in the size of the input. We do not allow the provers to communicate between each other, a prover P_i only has as access to the input string and the message history between himself and the verifier. We denote the probability that a verifier V accepts after interaction with provers $P_1, \ldots P_l$ on input x by $\Pr[V \leftrightarrow P_1, \ldots P_l]$. Now, we can define the class MIP(k) for multi-prover interactive proofs with k rounds, similar to how we defined IP(k).

Definition 3.23 (The class MIP(k)) Given a function $k : \mathbb{N} \to \mathbb{N}$, a language L is in the class MIP(k) if and only if there exists a probabilistic polynomial-time verifier V and provers $P_1, \ldots P_l$ such that, on every input x:

- an interaction of the verifier with the provers takes at most $k(|x|)$ rounds, and
- if $x \in L$ then $\Pr[V \leftrightarrow P_1, \ldots P_l \text{ accepts } x] \geq \frac{2}{3}$, and
- if $x \notin L$ then for all provers $P'_1, \ldots P'_l$, $\Pr[V \leftrightarrow P'_1, \ldots P'_l] \leq \frac{1}{3}$.

□

We also define MIP, the class of languages with a polynomial amount of rounds.

Definition 3.24 (The class MIP) We define

$$\text{MIP} = \bigcup_{c \in \mathbb{N}} \text{MIP}(n^c).$$

□

The following theorem shows the power of multi-prover interactive proof systems. It states that every language decidable in non-deterministic exponential time has a multi-prover interactive proof system (and this with only two provers and a constant number of rounds). Let us first define the class of languages decidable in non-deterministic exponential time, NEXPTIME $= \bigcup_{c \in \mathbb{N}} \text{NTIME}(2^{n^c})$.

Theorem 3.25 (Babai, Fortnow and Lund, 1992 [7]) MIP = NEXPTIME □

This is interesting, because we can separate NP from NEXPTIME using the non-deterministic time hierarchy theorem, as seen below. We first give the definition of time-constructable functions, which are used in the theorem.

Definition 3.26 (Time-constructable function) A function $f : \mathbb{N} \to \mathbb{R}$, where $f(n)$ is at least $O(n \log n)$, is called time-constructable if there exists a Turing machine, M, that, given a string 1^n, halts with the binary representation of $f(n)$ on its tape in time at most $O(f(n))$. □

Theorem 3.27 (Non-deterministic time hierarchy theorem [8]) Given functions $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$, if g is a time-constructable function and $f(n+1) = o(g(n))$, then NTIME$(f) \subsetneq$ NTIME(g). □

Thus, due to the non-deterministic time hierarchy, we know that NP $\subsetneq$ NEXPTIME = MIP.

Chapter 4

Zero-knowledge proof systems

In the previous chapter, when defining interactive proof systems, we always assumed that the verifier does not trust the prover. This means that we have to choose the verifier such that no prover could convince (with high probability) the verifier of a false claim. Now, what happens if the prover does not trust the verifier either? Let us say we have a prover trying to convince a verifier of the claim that a certain 3CNF formula is satisfiable. The prover could simply send a satisfying assignment to the verifier. This way, the verifier will be convinced that the formula indeed is satisfiable, but the verifier also learns a satisfying assignment for the formula. However, maybe the prover wants to keep this assignment secret. Now, we get to the question we focus on in this chapter: is it possible to prove the satisfiability of a 3CNF formula such that the verifier learns nothing but the fact that the formula is satisfiable? Such proof systems are called *zero-knowledge proof systems*. This chapter is based on Chapter 9 from the textbook of Goldreich [10], Chapter 8 from the textbook of Arora and Barak [4] and on the paper of Goldwasser, Micali and Rackoff [12], where zero-knowledge proof systems were first defined.

4.1 Background: circuit complexity

Here, we show an alternative way of formalising computation. This section serves as background to the remainder of this chapter. The definitions used are from the textbook of Sipser [19].

Definition 4.1 (Boolean circuit) A Boolean circuit is a collection of *gates* and *inputs* connected by *wires*. Cycles are not permitted. Gates take three forms: AND gates, OR gates and NOT gates. □

The wires in a circuit carry Boolean values. The AND, OR and NOT gates have one or more input wires and one or more output wires. They set the value of the output wire to the value obtained by applying the logical operators AND, OR and NOT as we know them to the input value(s). One of the gates of the circuit functions as the output gate. A Boolean circuit C with n inputs, outputs the value $C(a_1, \dots a_n)$ where $a_1, \dots a_n \in \{0, 1\}$ are the input values.

Definition 4.2 (Circuit family) A circuit family C is an infinite list of circuits $(C_0, C_1, C_2, \dots)$, where C_n has n inputs. We say that C decides a language L if for every string $w \in \{0, 1\}^*$, we have $w \in L$ if and only if $C_{|w|}(w) = 1$. □

Definition 4.3 (Circuit size) The size of a Boolean circuit is the number of gates that it contains. □

Definition 4.4 (Size complexity) The size complexity of a circuit family C is the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the size of C_n. □

We say a circuit family C is polynomial-sized if the size complexity of C is bounded by some polynomial.

4.2 Definition of a zero-knowledge proof system

It seems quite difficult to give a definition a proof systems where the verifier learns nothing. Instead, we define a zero-knowledge proof as one where the verifier learns nothing that it cannot compute by itself (that is, in polynomial time). To formalise this, we consider the messages that the verifier sees when interacting with the prover, the messages a verifier sees are random (because the verifier is probabilistic), so these messages have a certain probability distribution. Now, one can see that a verifier does not gain any knowledge when the verifier can take samples from this distribution by itself. While we could define zero-knowledge proofs in this way, we use a less strict definition, that is, we say that a verifier learns nothing if it can take samples from a distribution that is indistinguishable from the original by a polynomial-time algorithm. We define what it means that two random variables are indistinguishable in the following subsection.

4.2.1 Indistinguishability of random variables

We do not consider random variables on their own, but *probability ensembles*, which are sets of random variables. Such probability ensemble is always related to a language such that for each string in the language, the ensemble contains a random variable.

Definition 4.5 (Probability ensemble) A probability ensemble over a language L is a set $U = \{U(x) \mid x \in L\}$, where $U(x)$ is a random variable for $x \in L$. □

Now, we can define indistinguishability of these probability ensembles. To do this, we first define the concept of a negligible function, which (intuitively) is a function that asymptotically decreases faster than any polynomial.

Definition 4.6 (Negligible function) A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if and only if for every $c \in \mathbb{N}$, there exists a $x_0 \in \mathbb{N}$ such that $f(x) < \frac{1}{x^c}$ for all $x \geq x_0$. □

Now, we get to the most important concept needed when defining zero-knowledge proofs. We will see later that the messages a verifier sees in an interactive proof system form a probability ensemble. Informally, for every input, the verifier will see a different distribution of messages that he receives from the prover. Say we have a probability ensemble U, which represents the messages our verifier sees. So, when given input x the verifier sees samples of $U(x)$. Now, when can we say that the verifier learns nothing from these samples? Well, this is the case when the verifier could generate samples from a random variable $U'(x)$ (this means U' is a probability ensemble as well), which is *indistinguishable* from $U(x)$. Two probability ensembles U, U' are indistinguishable if no probabilistic Turing machine, on input x, can distinguish between samples of $U(x)$ and samples from $U'(x)$ with non-negligible probability. We specifically consider the case when the probability ensembles cannot be distinguished in polynomial time, this is called *computational indistinguishability*.

We define two probability ensembles to be computationally indistinguishable if it is not possible to distinguish samples of the ensembles in polynomial time with a non-negligible probability. In the definition, we use polynomial-sized circuits instead of a polynomial-time Turing machine. Note that this gives us a stricter definition. Given a polynomial-sized circuit family C, a probability ensemble U over a language L and a string $y \in L$, we denote the probability that C accepts x, when x is sampled from $U(y)$, as $\Pr_{x \leftarrow U(y)}[C \text{ accepts } x]$.

Definition 4.7 (Computational indistinguishability) Two probability ensembles U and U' over a language L are called computationally indistinguishable if for all polynomial-sized circuit families C, the function d, defined as below, is negligible.

$$d(n) = \max_{y \in L_n} |\Pr_{x \leftarrow U(y)}[C \text{ accepts } x] - \Pr_{x \leftarrow V(y)}[C \text{ accepts } x]|$$

where $L_n = \{x \mid x \in L \text{ and } |x| = n\}$ (that is, the set of strings in L with length n). □

We have said that a verifier gains no knowledge when it can take samples from a probability ensemble that is indistinguishable from the probability ensemble of the messages sent by the prover

(there is a different random variable for each input). So, we still have to define when a distribution can be sampled from in polynomial time.

Definition 4.8 (Simulator) A simulator S is a probabilistic, polynomial-time Turing machine with an input and output tape. We say the output of S on input x is the string found on the output tape after S halts, when starting with x on its input tape. Because the simulator is probabilistic, the output is a random variable, we denote this random variable by $S(x)$. □

Definition 4.9 (Computational approximability) A probability ensemble U over a language L is computationally approximable if there exists a simulator S such that the probability ensemble $\{S(x) \mid x \in L\}$ and U are computationally indistinguishable. □

4.2.2 Computational zero-knowledge

We now formalise the distribution of messages the verifier sees when interacting with the prover, we call this the *view* of a verifier.

Definition 4.10 (The view of a verifier) When a verifier V interacts with a prover in a k-round interactive proof system, we say the view of the verifier is the tuple containing all exchanged messages, $(m_1, \ldots m_k)$. Because a verifier is probabilistic, the view of the verifier is the outcome of a random variable. For verifier V and input x, we use $\text{VIEW}_V(x)$ to denote this random variable. □

As we can see, we can use the view of the verifier to obtain a probability ensemble over a language L: $\{\text{VIEW}_V(x) \mid x \in L\}$. When this ensemble is computationally approximable, the verifier does not learn anything from interacting with the prover. We could now define a zero-knowledge proof system this way. However, there is a problem with this definition. Say, a verifier V interacts with prover P, we know that the verifier learns nothing from a single run of the proof system, but what happens if the verifier interacts a second time with the prover, and keeps the messages of the previous run. In that case, can we still ensure the verifier learns nothing? Not with the proposed definition, therefore we change the definition of the verifier and his view to formalise the idea that a verifier can have some extra information at hand (the message history of previous runs is a possibility, but we do not restrict it to that). We let the verifier have an extra read-only tape that contains an *auxiliary input*, which can be any string with length polynomial in the length of the actual input. Using this, we change the definition of the view of a verifier.

Definition 4.11 (Auxiliary-input view) For verifier V, input x and auxiliary input z, we say $\text{VIEW}_V(x, z)$ is the random variable that represents the possible outcomes of the view of V. □

Now, we can finally give the definition of a zero-knowledge proof, which is defined as a interactive proof system with the added constraint that no verifier can make the prover output messages according to a distribution that is not computationally approximable.

Definition 4.12 (Zero-knowledge proof system) A zero-knowledge proof system is an interactive proof system with a verifier V and prover P with one additional constraint: for every verifier V' and constant $c \in \mathbb{N}$, the probability ensemble $\{\text{VIEW}_{V'}(x, z) \mid x \in L \text{ and } |z| \leq |x|^c\}$ is computationally approximable. □

We also define the class of languages that have zero-knowledge proofs.

Definition 4.13 (The class ZK) A language L is in the class ZK if L has a zero-knowledge proof system. □

Note 4.14 In the definition above, we only compare the distributions for cases where the input is in the language (that is, $x \in L$). This is because we use the probability ensemble $\{\text{VIEW}_{V'}(x, z) \mid x \in L \text{ and } |z| \leq |x|^c\}$. This means that, for the zero-knowledge constraint, we only consider an honest prover (that is, a prover that is not trying to proof a wrong claim). □

Example 4.15 Let us give an example of a proof system that is not a zero-knowledge proof system. Take the (non-interactive) proof system shown in Example 3.3 for the 3SAT language, where, on input a formula ϕ, the prover simply sends a satisfying assignment for ϕ. Since in this example the verifier does not send any message, the view of all possible verifiers are the same. Now, if we want to approximate the probability ensemble associated with the view using a simulator, that, on input ϕ, has to compute a satisfying assignment for ϕ. It is not known whether this is possible in polynomial time. So, unless P = NP, the proof system in Example 3.3 is not a zero-knowledge proof system. Note that if P = NP, all languages in NP trivially have zero-knowledge proof systems, because the prover does not even need to send any messages since the verifier can decide membership by itself (that is, in polynomial time). □

4.3 The existence of zero-knowledge proofs

Now, let us get back to the question we asked ourself in the introduction, is there a zero-knowledge proof system for the 3SAT language? Currently, this is not known. However, it is known that under a certain assumption, there are zero-knowledge proofs for every language in NP. This assumption is the existence of a non-uniform one-way function. Let us start by defining a one-way function, as hinted to by the name, intuitively, a one-way function is a function that is easy to evaluate but hard to invert.

Definition 4.16 (One-way functions) A polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is one-way if for every probabilistic polynomial-time algorithm M, we have that, given an input $f(x)$, M cannot compute a string y such that $f(y) = f(x)$ with a non-negligible probability. In other words, the function $p(n) = \Pr[f(M(f(x))) = f(x)]$ (where $M(f(x))$ denotes the output of M when given input $f(x)$ and x is chosen uniform at random from $\{0,1\}^n$) is a negligible function. □

Property 4.17 The existence of a one-way function implies P $\neq$ NP. □

Proof. We prove the contraposition: if P = NP, then there exists no one-way function. Assume, by contradiction, that f is a one-way function and P = NP. Let $L = \{\langle 1^l, x_0, y\rangle \mid \exists x : |x| = l, f(x) = y \text{ and } x_0 \text{ is a prefix of } x\}$. Clearly, L is in NP, because we can take x as a certificate, a polynomial-time verifier then only has to check that $|x| = l$, $f(x) = y$ (by definition, one-way functions are computable in polynomial time) and x_0 is a prefix of x. Because of our assumption, we have $L \in$ P. Say M is a polynomial Turing machine that decides L. Now, consider the following algorithm, A:

"On input $\langle y, 1^l\rangle$:

1. Let $x_0 = \epsilon$.
2. For every i from $1, \ldots l$, do:
   - 2.1. Run M on input $\langle 1^i, x_0 0, y\rangle$.
   - 2.2. If M accepts, set $x_0 = x_0 0$.
   - 2.3. Otherwise, set $x_0 = x_0 1$.
3. Output x_0. "

Note that $x_0 0$ means the string x_0 appended with the symbol 0, and likewise for $x_0 1$. It should be clear that, on input $\langle y, 1^l\rangle$, A outputs a value x such that $f(x) = y$, if there exists such x of length l. Also, A uses only a polynomial amount of time, because M also only uses a polynomial amount of time. Now, look at the function $p(l) = \Pr[f(A(\langle f(x), 1^l\rangle)) = f(x)]$, where x is sampled uniformly at random from the set of all strings of length l. Then, $A(f(x), 1^l)$ will output a x' such that $f(x') = f(x)$, so $p(l) = 1$, for every l. This is clearly a non-negligible function and thus f is not a one-way function. This is a contradiction and concludes the proof. □

A non-uniform one-way function is defined using families of boolean circuits instead of probabilistic polynomial-time Turing machines.

Definition 4.18 (Non-uniform one-way functions) A polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is one-way if for every polynomial-sized circuit family C, we have that, given an input $f(x)$, C cannot compute a string y such that $f(y) = f(x)$ with a non-negligible probability. In other words, the function $p(n) = \Pr[f(C(f(x))) = f(x)]$ (where $C(f(x))$ denotes the output of C when given input $f(x)$ and x is chosen uniform at random from $\{0,1\}^n$) is a negligible function. □

Theorem 4.19 (Goldreich, Micali and Widgerson, 1991 [11]) If there exists a non-uniform one-way function then NP $\subseteq$ ZK. □

Chapter 5

Introduction to the PCP theorem and the hardness of approximation

In the definition of the class NP, we have seen the concept of a verifier. In this chapter, we introduce a new type of verifier called a PCP verifier (PCP stands for *probabilistically checkable proof*). Using these PCP verifiers an alternative definition of the class NP is possible. This result is known as the PCP theorem, proven by Arora et al. in 1998 [5]. Remarkably, they have been awarded the Gödel prize for this result in 2001. One of the reasons why the PCP theorem is important is that it was used to prove the hardness of approximation for many problems. We will show some examples of problems that have such a hardness of approximation. This chapter is based on chapter 9 from the textbook of Goldreich [10] and on chapter 11 from the textbook of Arora and Barak [4]. Some of the definitions are taken from [9].

5.1 The PCP theorem

The PCP theorem gives an alternative definition of the class NP. This definition is similar to the verifier definition of the class NP seen in Chapter 1 because it also uses a verifier. The difference is that the PCP theorem states that every language in NP can be probabilistically verified by only looking at a portion of the proof of membership (or certificate). We call such a verifier a PCP verifier. A PCP verifier is a probabilistic Turing machine. We allow a PCP verifier to have a one-sided error similar to coRP. This means that even when the input is not in the language, the verifier may still accept with some probability called the *soundness error*.

We say that the *random complexity* of a probabilistic Turing machine M is bounded by $r : \mathbb{N} \to \mathbb{N}$ if, for every input x, the computation tree of M on input x contains at most $r(|x|)$ coin-flip steps on a single branch. Intuitively, the random complexity puts an upper bound on the amount of coin tosses during the computation.

A PCP verifier also does not have to read the entire proof of membership but it can choose which bits to read. To make this possible we allow random access to the proof. When we say a Turing machine M has random access to proof π, denoted by M^π, we modify M by adding a tape called the address tape. When M writes a string u over the alphabet $\{0, 1\}$ on the address tape it will receive the value of $\pi[i]$ where $\pi[k]$ is the k-th bit of π and $i \in \mathbb{N}$ is the number with binary representation u. We say that u is the address of the i-th bit of π. The action of reading a bit is called a *query*.

We say that the *query complexity* of a Turing machine M^π with access to proof π is bounded by $q : \mathbb{N} \to \mathbb{N}$ if, for every input x, M makes at most $q(|x|)$ queries to the proof.

Definition 5.1 (PCP verifier) We say a language L has a (r, q)-PCP verifier with soundness error s if there exists a polynomial-time probabilistic Turing machine V such that on input x:

- V has random complexity $O(r(|x|))$,
- V has query complexity $O(q(|x|))$,
- if $x \in L$ then there exists a proof π such that $\Pr[V^\pi \text{ accepts } x] = 1$, and
- if $x \notin L$ then for every proof π, $\Pr[V^\pi \text{ accepts } x] \leq s$.

□

Note 5.2 We say a PCP verifier is non-adaptive if the addresses it queries from the proof only depend on the input and the outcome of random coins. This means that an answer to some query cannot have an effect on the addresses that will be queried later on by the verifier. Given a non-adaptive (r, q)-PCP verifier and an input x the amount of possible addresses the verifier could query is at most $q(|x|) \cdot 2^{r(|x|)}$. Because of this we can assume without loss of generality that the length of a proof is at most $q(|x|) \cdot 2^{r(|x|)}$. From now on we assume that all PCP verifiers are non-adaptive. □

Next, we define the complexity class $\mathrm{PCP}_s(r, q)$.

Definition 5.3 (Complexity class $\mathbf{PCP}_s(r, q)$) $\mathrm{PCP}_s(r, q)$ is the class of languages that have a (r, q)-PCP verifier with soundness error s. □

We note that changing the soundness error does not change the complexity class, as shown in the following theorem.

Theorem 5.4 For every $s_1, s_2 \in]0, 1[$, $\mathrm{PCP}_{s_1}(r, q) = \mathrm{PCP}_{s_2}(r, q)$. □

Proof. Take s_1 and s_2 in $]0, 1[$. We assume without loss of generality that $s_1 \leq s_2$. We prove the two directions.

- $\mathrm{PCP}_{s_1}(r, q) \subseteq \mathrm{PCP}_{s_2}(r, q)$. Let L be a language in $\mathrm{PCP}_{s_1}(r, q)$. Let V be a (r, q)-PCP verifier with soundness error s_1 that decides L. When a verifier has soundness error (at most) s_1 then it clearly also has soundness error at most $s_2 \geq s_1$. So L is also in $\mathrm{PCP}_{s_2}(r, q)$.
- $\mathrm{PCP}_{s_1}(r, q) \supseteq \mathrm{PCP}_{s_2}(r, q)$. Let L be a language in $\mathrm{PCP}_{s_2}(r, q)$. Let V be a (r, q)-PCP verifier with soundness error s_2 that decides L. We can construct a (r, q)-PCP verifier V' with soundness error at most s_1 that decides L. The verifier V' works by running V a constant number of times, c. It will accept if and only if all of the c runs accept. Because the c outcomes of these runs are independent of each other, the soundness error of V' is ${s_2}^c$. So if we take $c = \lceil \log_{s_2}(s_1) \rceil$, we get soundness error ${s_2}^{\lceil \log_{s_2}(s_1) \rceil} \leq s_1$.

□

Because of this we simply use the notation $\mathrm{PCP}(r, q)$ to denote the class of languages decided by a (r, q)-PCP verifier with some constant soundness error $s \in]0, 1[$.

Example 5.5 (A language with a PCP verifier) In this example we use the language GNI (graph non-isomorphism), defined as

$$\mathrm{GNI} = \{\langle G_0, G_1 \rangle \mid G_0 \text{ and } G_1 \text{ are not isomorphic}\}.$$

Where $\langle G_0, G_1 \rangle$ is the binary representation of graphs G_0 and G_1. The exact representation used is not very important as long as it is a reasonable one. Graphs G_0 and G_1 are isomorphic is there exists a bijection $f : V_{G_0} \to V_{G_0}$ such that for all $u, v \in V_{G_0}$ it holds that $(u, v) \in E_{G_0}$ iff $(f(u), f(v)) \in E_{G_1}$. Here, V_G is the set of vertices of graph G and E_G is the set of edges of graph G.

We show a PCP verifier V for GNI. The verifier V works as follows, on input $x = \langle G_0, G_1 \rangle$ and random access to proof π:

1. Choose, uniformly at random, $a \in \{0,1\}$ and say $b = 1 - a$ (so b is the element that was not chosen).
2. Choose, uniformly at random, a bijection $f : V_{G_a} \to V_{G_b}$.
3. Construct a new graph H with $V_H = \{f(u) \mid u \in V_{G_a}\}$ and $E_H = \{(f(u), f(v)) \mid (u,v) \in E_{G_a}\}$.
4. Query the proof π at location i where i is the natural number that has $\langle G_a \rangle$ as its binary representation.
5. If the result of the query, $\pi[i]$, equals a then accept, otherwise reject.

The correctness of this verifier relies on the fact the graph isomorphism is a transitive property, this means that if graphs G_1 and G_2 are isomorphic and G_2 and graph G_3 are isomorphic then G_1 and G_3 are also isomorphic. Because graphs are isomorphic exactly when there exists a bijection between them satisfying the aforementioned condition, the constructed graph H is taken uniformly at random from the set of all graphs (with vertex set V_{G_b}) isomorphic to the chosen graph G_a. So, if G_0 and G_1 are not isomorphic then the prover can distinguish isomorphisms of G_0 from isomorphisms of G_1 and thus create a proof that will always make the verifier accept by putting a 0 on locations that have an address equal to the binary representation of some isomorphism of G_0 and similarly putting a 1 on locations with addresses coinciding with the representation of an isomorphism of G_1. This is possible because if G_0 and G_1 are not isomorphic then no graph is isomorphic with both G_0 and G_1. However, if G_0 and G_1 are isomorphic then all graphs isomorphic with G_0 are also isomorphic with G_1. Because both graphs have an equal chance of being chosen in the first step, there is no proof that would make the verifier accept with probability higher than $\frac{1}{2}$.

The amount of random coins used in step 1 is 1 and in step 2 is at most $O(n^2)$ with $n = |x|$ the size of the input. The remaining steps do not use any random coins so the total amount of random coins used is polynomial in n so $r = O(n^2)$. The amount of bits read from the proof is exactly one so $q = O(1)$. Finally it should be clear that the verifier has time complexity at most polynomial in the size of the input.

From this example, we conclude: $\text{GNI} \in \text{PCP}(n^k, 1)$ for some constant k. □

Now we are ready to show the PCP theorem. This theorem says that every language in NP can be probabilistically verified using only $O(\log n)$ random coins and $O(1)$ queries.

Theorem 5.6 (PCP theorem [5]) $\text{NP} = \text{PCP}(\log n, 1)$ □

Because the proof of this theorem is quite long we will not see it here but in Section 5.3.

5.2 The hardness of approximation

One of the reasons the PCP theorem is so important is that it is used to prove the hardness of approximation for many optimization problems. Until now we have only seen decision problems where the output is a binary yes or no (or accept and reject in a Turing machine). On the other hand, an optimization problem is solved by finding a solution to some input such that an objective function is either maximized or minimized.

Definition 5.7 (Optimization problem) An optimization problem Π is defined by:

- a function sol_Π that maps an input to the set of solutions for that input, and
- an objective function obj_Π that maps an input x and a solution y for x to a real number.

For a maximization problem the goal is, given an input x, to find a solution y such that $\text{obj}_\Pi(x,y)$ is maximal. Then the optimal value of the objective function for x is

$$\text{opt}_\Pi(x) = \max_{y \in \text{sol}_\Pi(x)} \text{obj}_\Pi(x,y).$$

For a minimization problem the goal is to find a solution such that the objective function is minimal so the optimal value of the objective function is

$$\mathrm{opt}_\Pi(x) = \min_{y \in \mathrm{sol}_\Pi(x)} \mathrm{obj}_\Pi(x, y).$$

□

To give an example of an optimization problem, we take a look at the decision problem INDSET that asks if there exists an independent set of some size in a graph.

$$\mathrm{INDSET} = \{\langle G, k\rangle \mid G \text{ is a graph with an independent set of } k \text{ nodes}\}.$$

The optimization version of this problem would be to find the maximum independent set of a graph, called max-INDSET. Here the solutions of an input graph G are all independent sets of G. The objective function, given a graph and an independent set, outputs the number of nodes in the independent set.

It should be clear that, unless P = NP, there are no polynomial time algorithms solving the optimization version of an NP-hard problem. Now, an interesting question to ask is whether there exist polynomial time algorithms that approximate these optimization problems. This is what we look at in this section. Let us first define what an approximation algorithm is.

Definition 5.8 (Approximation algorithm) Given an optimization problem Π, an algorithm A is a ρ-optimal approximation of Π with $\rho \in \,]0, 1[$ if on input x, A outputs $A(x) = y$ such that y is a solution for x and

- for maximization problems: $\mathrm{obj}_\Pi(x, y) \geq \rho \cdot \mathrm{opt}_\Pi(x)$, or
- for minimization problems: $\mathrm{obj}_\Pi(x, y) \leq \frac{1}{\rho} \cdot \mathrm{opt}_\Pi(x)$.

□

Example 5.9 Consider the optimization version of 3SAT, called max-3SAT, in which we want to find a variable assignment to the input formula ϕ such that the fraction of satisfied clauses is maximal. Thus, max-3SAT is a maximization problem where the solutions are variable assignments and the objective function is the fraction of satisfied clauses. As it turns out, there is a very straightforward $\frac{1}{2}$-optimal approximation algorithm for max-3SAT. This algorithms works by outputting the variable assignment that sets all variables to either true or false, whichever satisfies the most clauses. One can see that this will always satisfy half of the clauses because each clause is satisfied by at least one of the two possible assignments. Because it always satisfies half of the clauses, it definitely satisfies at least $\frac{1}{2}$ times the maximal fraction of satisfiable clauses.

To give a concrete example, let

$$\phi = (\neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3).$$

Consider the assignment a_1 that maps all variables to true, and a_2 that maps all variables to false. Now, the number of clauses of ϕ that are satisfied by a_1 is 1, while for a_2, the number of satisfied clauses is 2, which is more than half. Note that ϕ is satisfiable. □

5.2.1 Constraint satisfaction problems

Before going further, we define constraint satisfaction problems which we will use in a theorem equivalent to the PCP theorem. This equivalent theorem is then used to prove the hardness of the approximations of some problems. The definitions used are from [9].

Definition 5.10 (Constraint) Given a set of variables $V = \{v_1, \ldots, v_n\}$ and an alphabet Σ, a q-ary constraint over alphabet Σ and variables V is a tuple $(R, i_1, \ldots, i_q)$, where:

- $R \subseteq \Sigma^q$ is the set of tuples of variable assignments that would satisfy the constraint, and
- $i_1, \ldots, i_q \in \mathbb{N}$ are indices to the variables that the constraint depends on.

A constraint is satisfied by an assignment $a : V \to \Sigma$ if $(a(v_{i_1}), \ldots, a(v_{i_q})) \in R$. □

Definition 5.11 (Constraint satisfaction problem) The constraint satisfaction problem over an alphabet Σ (CSP_Σ) is the problem of, given a set of constraints $\{c_1, \ldots, c_m\}$ over alphabet Σ and variables V, deciding if there exists an assignment to V that satisfies all constraints. □

The accompanying optimization problem is called max-CSP_Σ which is the problem of finding an assignment of the variables that maximizes the amount of constraints satisfied by that assignment. We call the *unsat-value* of a CSP_Σ instance C, denoted by $\text{UNSAT}(C)$, the minimum fraction of constraints that are not satisfied over all variable assignments. A CSP instance C is satisfiable if and only if $\text{UNSAT}(C) = 0$. The solutions to a CSP instance are assignments to the variables of the instance and the objective function, given a CSP instance and a variable assignment, outputs the fraction of constraints satisfied by the assignment. The optimal value is thus the maximal fraction of constraints satisfied by a single assignment, which is 1 in the case where the CSP instance is satisfiable.

We denote q-CSP_Σ to be the constraint satisfaction problem where all constraints are q-ary. Respectively we denote max-q-CSP_Σ as the problem of maximizing the amount of satisfied constraints in a CSP instance where all constraints are q-ary.

Now we can show the following theorem which is equivalent to the PCP theorem and we will call this the hardness of approximation version of the PCP theorem.

Theorem 5.12 There exists a q and an alphabet Σ such that for every language $L \in \text{NP}$ there exists a polynomial-time reduction f of L to q-CSP_Σ such that:

- $x \in L \implies \text{UNSAT}(f(x)) = 0$, and
- $x \notin L \implies \text{UNSAT}(f(x)) \geq \frac{1}{2}$.

□

The following is a proof that Theorem 5.6 and Theorem 5.12 are equivalent.

Proof. First we prove that Theorem 5.6 implies Theorem 5.12.

Let L be any language in NP. Because of Theorem 5.6 there exists a (r, q)-PCP verifier V for L where $r = O(\log n)$ and $q = O(1)$. Now, consider the reduction f that, given input x, constructs the q-CSP instance $f(x) = C$ where C contains a constraint c_w for every possible outcome w of r random coins. The constraint $c_w = (R, i_1, \ldots i_q)$ has as relation $R \subseteq \Sigma^q$ that contains all combinations of query answers that would make V accept given random coins w, and $i_1, \ldots, i_q$ are the indices of symbols in the proof that would be queried by V given random coins w.

This set of constraints can be constructed in polynomial time, because the amount of constraints equals the amount of possible outcomes of the random coins, which is $2^r = 2^{O(\log n)}$ and thus is polynomial in $n = |x|$. Each constraint is computed by simulating V for every possible combination of q query answers, there are $2^q = 2^{O(1)} = O(1)$ such combinations and it is given that V can be simulated in polynomial time.

It should also be clear that when $x \in L$ then the CSP instance C is satisfiable and $\text{UNSAT}(C) = 0$. In the case of $x \notin L$ we know that there is no proof that makes the verifier accept on more than half the possible outcomes of random coins. It follows that no variable assignment will satisfy more than half of the constraints of C so $\text{UNSAT}(C) \geq \frac{1}{2}$.

Now we proof the other direction: Theorem 5.12 implies Theorem 5.6.

Say L is a language in NP. We have to proof that there exists a $(\log n, 1)$-PCP verifier V for L given that there exists a polynomial-time reduction f from L to max-q-CSP such that:

- $x \in L \implies \text{UNSAT}(f(x)) = 0$, and
- $x \notin L \implies \text{UNSAT}(f(x)) \geq \frac{1}{2}$.

The verifier works as follows.

$V =$ "On input x and random access to proof π:

1. Run f to on x to obtain the CSP instance $C = f(x)$.
2. Randomly select one of the constraints $c = (R, i_1, \dots, i_q)$ of C.
3. Query the proof at locations $i_1, \dots, i_q$ and call the answers $a = (a_1, \dots, a_q)$.
4. If $a \in R$ accept, otherwise reject. "

In other words, the proof is a variable assignment for the CSP obtained by the reduction f. The verifier will select a random constraint and check if it is satisfied by the variable assignment.

The verifier runs in polynomial time because it is given that f is computable in polynomial time and the other steps are also possible in time at most polynomial in the input size $n = |x|$.

For the soundness and completeness requirements, we have:

- If $x \in L$ then $f(x)$ is satisfiable so there exists a proof (variable assignment) such that V always accepts.
- If $x \notin L$ then $\text{UNSAT}(f(x)) \geq \frac{1}{2}$ so for every variable assignment the probability that a randomly selected constraint is unsatisfied is at least $\frac{1}{2}$. It follows that for every proof the verifier accepts with at most probability $\frac{1}{2}$.

□

This theorem implies what we call *hardness of approximation* for max-q-CSP, the reason should become clear from the following corollary.

Corollary 5.13 Assuming P $\neq$ NP, there exists a q such that there exists no polynomial time $\frac{1}{2}$-optimal approximation algorithm for max-q-CSP. □

Proof. We proof this by contradiction. Assume P $\neq$ NP and that for every q there exists a polynomial time $\frac{1}{2}$-optimal approximation algorithm for max-q-CSP. Now we can construct a polynomial time algorithm that can decide every language in NP given the reduction from Theorem 5.12 called f:

"On input x:

1. Construct CSP instance $C = f(x)$.
2. Run A on C.
3. If the outcome of A is an assignment satisfying more then a $\frac{1}{2}$ fraction of the constraints of C, accept.
4. Otherwise reject. "

Where A is the polynomial time $\frac{1}{2}$-optimal approximation algorithm guaranteed by our assumption. Of course, when we have a polynomial time algorithm for every language in NP, then P = NP and this is a contradiction. □

5.2.2 Examples of hardness of approximation

Now that we have shown Theorem 5.12 to be equivalent to the PCP theorem, we can use it to prove hardness of approximation for other problems. To do this we use *gap-preserving reductions*, that given input instances with a gap in their optimal values output instances also with a gap in their optimal values.

The first problem we look at is the optimization version of 3SAT, max-3SAT. Max-3SAT is the problem of finding a variable assignment for which the amount of satisfied clauses is maximal. Like with CSP we denote $\text{UNSAT}(\phi)$ to be the minimum fraction of unsatisfied clauses of the 3CNF

formula ϕ over all variable assignments. Again as with CSP the solutions are variable assignments and the objective function is the amount of satisfied clauses.

Theorem 5.14 There exists a $\rho \in]0,1[$ such that for every language $L \in$ NP there exists a polynomial-time reduction f of L to 3SAT such that:

$x \in L \implies \text{UNSAT}(f(x)) = 0$, and

$x \notin L \implies \text{UNSAT}(f(x)) \geq \rho$. □

Proof. We use Theorem 5.12 so we know that for every language L in NP there exists a polynomial-time reduction f mapping strings to q-CSP instances, for some constant q, such that:

- $x \in L \implies \text{UNSAT}(f(x)) = 0$, and
- $x \notin L \implies \text{UNSAT}(f(x)) \geq \frac{1}{2}$.

We show a gap-preserving reduction g of such q-CSP instances to 3CNF formulas, the composition of g and f will then be the reduction needed. Reduction g works as follows, given as input q-CSP instance C. We can convert a constraint $c = (R, i_1, \ldots, i_q)$ to at most 2^q CNF clauses each having q literals. We can do this by converting $\overline{R}$ to DNF clauses and taking the negation of the resulting DNF formula. Let ϕ be the conjunction of these clauses of every constraint of C, so ϕ contains at most $m \cdot 2^q$ clauses in total, each with q literals, where m is the amount of constraints in C. It is important to note that when any variable assignment to C violates at least a fraction b of the constraints then at least a $\frac{b}{2^q}$ fraction of the clauses of ϕ are violated given any assignment. Finally we convert ϕ into a 3CNF formula ϕ' which contains at most $q \cdot m \cdot 2^q$ clauses. We do this by converting every clause $c = (x_1 \vee \cdots \vee x_n)$ to one or more clauses with 3 literals as follows:

- If $n = 3$ we simply keep c itself.
- If $n < 3$ we repeat one of the literals in c until there are 3 literals. For example we convert $c = (x_1 \vee x_2)$ to $c' = (x_1 \vee x_2 \vee x_2)$.
- If $n > 3$ we add variables $y_1, \ldots, y_{n-2}$ and convert c to the following clauses: $(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge \cdots \wedge (\neg y_{n-2} \vee x_{n-1} \vee x_n)$

Note that this conversion does not produce an equivalent formula but has the property that ϕ' is satisfiable if and only if ϕ is satisfiable. Now when any assignment violates at least fraction c of the clauses of ϕ then at least a fraction $\frac{c}{q}$ of the clauses of ϕ' are always violated. So ϕ' violates at least a fraction $\frac{b}{q \cdot 2^q}$ of its clauses where b is the unsat-value of C.

Now the composition of reductions g and f gives us the reduction we need:

- If $x \in L$ then the CSP instance $f(x)$ is satisfiable and so is the 3CNF formula $g(f(x))$.
- If $x \notin L$ then $\text{UNSAT}(f(x)) \geq \frac{1}{2}$ and $\text{UNSAT}(g(f(x))) \geq \frac{1}{2 \cdot q \cdot 2^q}$.

□

As a second example we show a similar result for max-INDSET. We define IS as the function that maps a graph to the fraction of nodes that are member of the largest independent set.

Theorem 5.15 There exists a $\rho \in]0,1[$ such that the ρ-optimal approximation of max-INDSET is NP-hard. There exists a $\rho \in]0,1[$ such that for every language $L \in$ NP there exists a reduction f of L to INDSET such that:

- $x \in L \implies \text{IS}(f(x)) = \frac{1}{3}$, and
- $x \notin L \implies \text{IS}(f(x)) < \frac{\rho}{3}$.

□

Proof. We use the same reduction used to proof NP-completeness of INDSET in Chapter 1. Let us call this reduction f. Remember that $f(\phi) = \langle G, k\rangle$ where G is a graph and k is the number

of clauses in ϕ. For every clause in ϕ there is a group of 3 nodes in G, one for each literal in the clause. Nodes are connected when they are part of the same group or when their associated literals are negations of each other.

Let L be some language in NP. We will use the reduction g of 3SAT to L shown in Theorem 5.14. If we compose the reductions g and f we can see that:

- If $x \in L$ then $\text{UNSAT}(g(x)) = 0$, let m be the number of clauses in $g(x)$ and $G = f(g(x))$. The number of nodes in G is $3m$ and the number of nodes in the largest independent set is m so $\text{IS}(f(g(x)) = \frac{1}{3}$.
- If $x \notin L$ then $\text{UNSAT}(g(x)) \geq \rho$ for some constant ρ as guaranteed by Theorem 5.14. Again let m be the number of clauses in $g(x)$ and $G = f(g(x))$. Because the maximum fraction of satisfied clauses is less than ρ, the number of nodes in the largest independent set of G is less than $\rho \cdot m$. So the fraction of nodes in the largest independent set is less than $\frac{\rho}{3}$.

□

5.3 Proof of the PCP theorem

5.3.1 Expander graphs

Intuitively, expander graphs are graphs that are 'well-connected' while at the same time being 'sparse'. Note that we do not say of a single graph that it is an expander graph (or not). Instead, we use a certain property of a graph that measures how 'good' of an expander graph it is. In this section, we define two such *expansion properties* of a graph, and how they are related to each other. This section is based on a survey of expander graphs by Hoory, Linial and Widgerson [14].

From now on, when talking about graphs, we mean undirected *multigraphs*. In a multigraph, we allow there to be multiple edges between any two nodes. We also allow loops, a loop in a graph is an edge that connects a node to itself. See Figure 5.1 for an example of an undirected multigraph with loops, there are two edges between node B and node C, and both node A and node D have a loop.

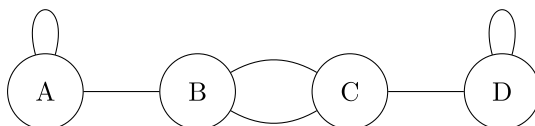


Figure 5.1: An example of an undirected multigraph with loops.

The degree of a node is the number of edges connected to it. We say a graph is *regular* if all of its nodes have the same degree. More specifically, we say a graph is d-regular if all of its nodes have degree d. As an example, let us look at the degree of the nodes of the graph shown in Figure 5.1.

- The node A has one loop, and is connected to node B by one edge, thus the degree of A is 2.
- The node B is connected to node A with one edge and connected to node C with two edges, thus its degree is 3.
- The node C is connected to node D with one edge and connected to node B with two edges, so its degree is 3.
- The node D is connected to node C with one edge and has one loop, so its degree is 2.

Edge expansion ratio

Given a graph $G = (V, E)$, and two sets of nodes, $S, T \subseteq V$, we denote the number of edges connecting a node in S with a node in T by $E(S, T)$. The *edge boundary* of a set of nodes $S \subseteq V$ is

the number of edges connecting a node in S to a node outside S, which can be written as $E(S, \overline{S})$, where $\overline{S} = V \setminus S$.

Next, we define the first expansion property, known as the *edge expansion ratio* of a graph.

Definition 5.16 (Edge expansion ratio) Given a graph $G = (V, E)$, the edge expansion ratio of G, denoted by $h(G)$, is defined as:

$$h(G) = \min_{S \in H} \frac{E(S, \overline{S})}{|S|},$$

where

$$H = \{S \mid S \subseteq V \text{ and } 0 < |S| \leq \frac{|V|}{2}\}.$$

□

Intuitively, a graph $G = (V, E)$ has a high edge expansion ratio if every subset $S \subseteq V$ has a lot of outgoing edges, that is, S has a large edge boundary.

Example 5.17 Let us calculate the edge expansion ratio of the graph shown in Figure 5.1. We have the node set $V = \{A, B, C, D\}$. If we consider the subset $S = \{A\}$, than we have

$$\frac{E(S, \overline{S})}{|S|} = \frac{1}{1} = 1.$$

One can verify that is the minimum for any subset of nodes of size less than $\frac{|V|}{2}$, thus the edge expansion ratio of the graph equals 1. □

In the introduction, we stated that 'good' expander graphs are 'well-connected'. Consider the graph shown in Figure 5.2, this graph is disconnected and should thus be a 'bad' expander. Take the set of nodes $S = \{D, E\}$ of one of the connected components in the graph. Then clearly we have $E(S, \overline{S}) = 0$ and thus the edge expansion ratio of the graph is 0, which is the smallest possible value for an edge expansion ratio. It should be easy to see that, in general, every disconnected graph has an edge expansion ratio of 0.

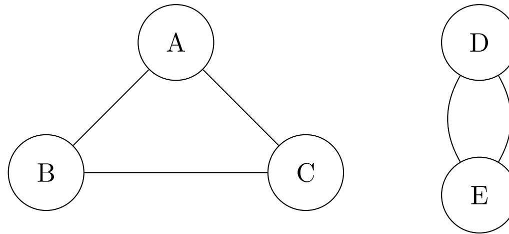


Figure 5.2: An example of a disconnected graph.

As an example of a 'well-connected' graph, let us take the complete graph with 5 nodes, K_5, shown in Figure 5.3. Because there are 5 nodes, we only have to consider subsets of nodes with a size up to 2. For every subset of nodes of size 1, the edge boundary is 4 and for every subset of nodes of size 2, the edge boundary equals 6. Thus

$$h(K_5) = \min \left\{ \frac{4}{1}, \frac{6}{2} \right\} = 3.$$

We also stated that good expander graphs should be 'sparse', and by that we mean that the ratio of edges to nodes is low. Therefore, we do not consider a graph on its own, but a family of graphs.

Definition 5.18 (Family of graphs) A family of graphs is a set $\mathcal{F} = \{G_1, G_2, G_3, \dots\}$, which for every $i \in \mathbb{N}$ contains a graph G_i where G_i has exactly i nodes. □

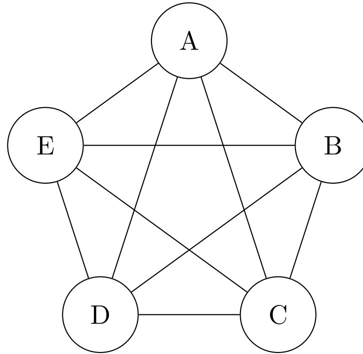


Figure 5.3: The complete graph K_5.

To get sparse graphs, we consider families of graphs of which all graphs are d-regular for some constant d. Now, we can define families of expander graphs, these consists of d-regular graphs for some constant d and have a edge expansion ratio greater than some constant.

Definition 5.19 (Family of expander graphs) Given $d \in \mathbb{N}$ and $h_0 \in \mathbb{R}$ with $h_0 > 0$, a family of d-regular graphs $\mathcal{F}$ is called a *family of h_0-expander graphs* if for every $G_i \in F$, we have $h(G_i) \geq h_0$. □

Now, it is interesting to consider graphs that can be efficiently constructed, therefore we define polynomial-time constructible families of graphs.

Definition 5.20 (Polynomial-time constructible family of graphs) A family of graphs $\mathcal{F} = \{G_1, G_2, G_3, \dots\}$ is constructable in polynomial-time if there exists an algorithm that, given as input a number $i \in \mathbb{N}$, outputs the graph G_i and runs in time at most a polynomial in i. □

The following theorem states that there exist polynomial-time constructible families of expander graphs. We do not prove this here, but an example of such constructible family can be found in [16].

Theorem 5.21 There exist a $d \in \mathbb{N}$ and a $h_0 \in \mathbb{R}$ with $h_0 > 0$, such that there exists a polynomial-time constructible family of d-regular graphs $\mathcal{F} = \{G_1, G_2, G_3, \dots\}$, with $h(G_i) \geq h_0$ for every $G_i \in F$, that is, a family of h_0-expander graphs. □

Eigenvalue gap

Let us move on to the second expansion parameter, called the *eigenvalue gap*. This parameter is derived from the eigenvalues of the adjacency matrix of a graph.

From now on, we assume that there is some order associated with the nodes of a graph. So, when we say the i-th node, we mean the node that comes on the i-th place in that order. Now, we define the adjacency matrix of a graph.

Definition 5.22 The adjacency matrix of a graph $G = (V, E)$, denoted by $A(G)$, is an $n \times n$ matrix with $n = |V|$, such that $A(G)_{ij}$ equals the number of edges between the i-th node and the j-th node. □

Example 5.23 Let us look at the adjacency matrix of the graph, G, shown in Figure 5.1. We order the nodes alphabetically, so the order is A, B, C, D. We get:

$$A(G) = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Note that:

- the elements on the diagonal of the adjacency matrix are equal to the number of loops on the corresponding nodes;
- since we are dealing with undirected graphs, the adjacency matrix is always symmetrical; and
- the sum of the elements on the i-th row is equal to the degree of the i-th node, and, because the matrix is symmetric, the sum of the elements on the i-th column is also equal to the degree of the i-th node.

□

Next, we define the eigenvalues of a symmetric matrix.

Definition 5.24 Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$, the eigenvalues of A are $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \in \mathbb{R}$ where, for each eigenvalue λ_i, there exists a vector $\vec{v}_i \in \mathbb{R}^n$, with $\vec{v}_i \neq \vec{0}$, called an eigenvector, such that $A\vec{v}_i = \lambda_i\vec{v}_i$, this means: the result of the multiplication of matrix A with vector $\vec{v}_i$ is equal to the result of the (scalar) multiplication of λ_i with $\vec{v}_i$. □

Note 5.25 In the following, when we talk about eigenvalues of a graph, we mean the eigenvalues of the adjacency matrix of the graph. □

There are some interesting properties of the eigenvalues of a graph. First, we show that we can bound the absolute value of the eigenvalues of regular graphs.

Property 5.26 Given a d-regular graph G, if λ is an eigenvalue of $A(G)$ then $|\lambda| \leq d$. □

Proof. Say G is a d-regular graph with adjacency matrix $A = A(G)$. Let n be the number of nodes of G. Let λ be an eigenvalue of A and let $\vec{v} \in \mathbb{R}^n$ be a corresponding eigenvector. Thus we have

$$A\vec{v} = \lambda\vec{v}.$$

Now, take v_m to be the largest component (in absolute value) of $\vec{v}$ and let $\vec{u} = \frac{\vec{v}}{v_m}$ (v_m can not be zero since that would imply $\vec{v} = \vec{0}$ which is not an eigenvector). Then, the largest component of $\vec{u}$, say u_k, equals 1, where the position of u_k in $\vec{u}$ is k. Since $\vec{u}$ is also an eigenvector, we have

$$A\vec{u} = \lambda\vec{u}.$$

From this follows that

$$|(A_{k1}, A_{k2}, \ldots, A_{kn}) \cdot \vec{u}| = |\lambda u_k| = |\lambda|.$$

In other words, the absolute value of the dot product of the k-th row of A with the vector $\vec{u}$ equals $|\lambda|$. Then we have:

$$\begin{aligned}
|\lambda| &= |(A_{k1}, A_{k2}, \ldots, A_{kn}) \cdot \vec{u}| \\
&= \left|\sum_{i=1}^{n} A_{ki}u_i\right| \\
&\leq \left|\sum_{i=1}^{n} A_{ki}u_k\right| \\
&= \left|\sum_{i=1}^{n} A_{ki}\right| = |d| = d.
\end{aligned}$$

The last equality holds because the sum of the components of a row in the adjacency matrix equals the degree of the corresponding node. Because in this case the graph is d-regular, this is d. □

Not only are the absolute values of the eigenvalues of regular graphs bounded by the degree, we can also show that the largest eigenvalue of a regular graph is equal to the degree.

Property 5.27 Given a d-regular graph G and its adjacency matrix $A = A(G)$, with $\lambda_1 \leq \lambda_2, \cdots \leq \lambda_n$ the eigenvalues of A, then $\lambda_1 = d$. □

Proof. Say G is a d-regular graph with adjacency matrix $A = A(G)$. Let n be the number of nodes of G. Let $\vec{1} \in \mathbb{R}^n$ be the vector whose elements are all 1. Consider the result of $A \cdot \vec{1}$:

$$A \cdot \vec{1} = \left(\sum_{i=1}^{n} A_{1i}, \sum_{i=1}^{n} A_{2i}, \ldots, \sum_{i=1}^{n} A_{ni} \right).$$

Because the sum of each row in the adjacency matrix equals the degree, d, we have:

$$A \cdot \vec{1} = (d, d, \ldots d) = d\vec{1}.$$

So, d is an eigenvalue of A with corresponding eigenvector $\vec{1}$. By Property 5.26, we know that there is no eigenvalue greater than d, thus d is the largest eigenvalue of A, or $\lambda_1 = d$. □

Definition 5.28 Given a d-regular graph G and its adjacency matrix $A = A(G)$, with $\lambda_1 \leq \lambda_2, \cdots \leq \lambda_n$ the eigenvalues of A, the *eigenvalue gap* of G is defined as $d - \lambda_2$. □

Note that since $\lambda_1 = d$, the eigenvalue gap is equal to $\lambda_1 - \lambda_2$, which explains the name 'eigenvalue gap'.

Example 5.29 Consider the graph shown in Figure 5.4, which is obtained by adding an edge from node A to node D to the graph shown in Figure 5.1. The result is a 3-regular graph. The adjacency matrix of this graph is shown below.

$$A(G) = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

The eigenvalues of the adjacency matrix are:

$$\begin{aligned} \lambda_1 &= 3 \\ \lambda_2 &= 1 \\ \lambda_3 &= \sqrt{2} - 1 \\ \lambda_4 &= -1 - \sqrt{2}. \end{aligned}$$

Thus the eigenvalue gap of this graph is $\lambda_1 - \lambda_2 = 2$.

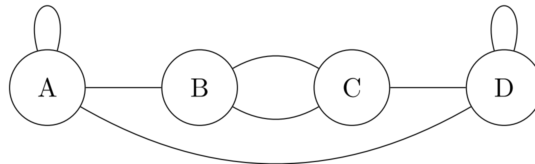


Figure 5.4: The graph shown in Figure 5.1 with an added edge to make it 3-regular.

□

The following theorem shows that the eigenvalue gap and the edge expansion ratio of a graph are related to each other. This means that the eigenvalue gap can also be used as a measure for how 'well-connected' the graph is. We do not give a proof here, but it can be found in [3].

Theorem 5.30 If G is a d-regular graph with edge expansion ratio $h(G)$ and eigenvalue gap $d - \lambda_2$, then:

$$\frac{d - \lambda_2}{2} \leq h(G) \leq \sqrt{2d(d - \lambda_2)}.$$

□

Because disconnected graphs have an edge expansion ratio of 0, it follows that disconnected graphs have an eigenvalue gap of 0 as well.

Example 5.31 Let us look back at the disconnected graph shown in Figure 5.2. Clearly, this graph is 2-regular. The adjacency matrix is show below.

$$A(G) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

The eigenvalues of the adjacency matrix are:

$$\begin{aligned} \lambda_1 &= 2 \\ \lambda_2 &= 2 \\ \lambda_3 &= -1 \\ \lambda_4 &= -1 \\ \lambda_5 &= -2. \end{aligned}$$

And as we can see, the eigenvalue gap is indeed $\lambda_1 - \lambda_2 = 0$. □

Because of the relation between the edge expansion ratio and the eigenvalue gap, when we can construct graphs with an edge expansion ratio greater than some constant, we obtain graphs with an eigenvalue gap greater than some constant. Thus, using Theorem 5.20 we can show that there exists a polynomial-time constructible family of graphs with an eigenvalue gap greater than some constant.

Theorem 5.32 There exists a $d \in N$ and a $\lambda \in \mathbb{R}$ with $\lambda > 0$, such that there exists a polynomial-time constructible family of d-regular graphs $F = \{G_1, G_2, G_3, \dots\}$, where, for every $G_i \in F$, the eigenvalue gap of G_i is at least λ. □

Random walks on expander graphs

One of the nice properties of expander graphs is that when taking a random walk on an expander graph, the resulting probability distribution converges rapidly to the uniform distribution over the nodes of the graph. Let us start by defining (random) walks on a graph.

Definition 5.33 A *walk* on a graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \dots \in V$ such that, for every index i, the nodes v_i and v_{i+1} are connected with an edge. □

Definition 5.34 A *random walk* on a graph $G = (V, E)$ is a walk $v_1, v_2, \dots$ such that v_{i+1} is selected uniformly at random from the neighbors of v_i and v_1 is chosen from some initial probability distribution π_1. With a random walk we associate probability distributions $\pi_1, \pi_2, \dots$ such that, for every index i, $\pi_i(x)$ is the probability that $v_i = x$ with $x \in V$, in other words, $\pi_i(x)$ is the probability that node x is reached in the i-th step of the random walk. □

The following theorem states that there is relation between the behaviour of a random walk on a graph and the eigenvalues of the graph, we do not give a proof here. Given a d-regular graph G with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, we define $\lambda(G) = \max(|\lambda_2|, |\lambda_n|)$, that is, the second-largest eigenvalue in absolute value.

Theorem 5.35 Given a d-regular graph $G = (V, E)$ with $\lambda(G) = \lambda$ and $F \subseteq E$ a subset of the edges without loops. Let π_1 be the probability distribution obtained when randomly choosing a edge from F and then choosing one of the two endpoints of that edge at random. In other words, $\pi_1(x)$ equals the fraction of edges incident to the node x that are in F, divided by 2. Then, the

probability that a random walk with initial distribution π_1 chooses an edge in F on the $i+1$-th step, is at most

$$\frac{|F|}{|E|} + \left(\frac{\lambda}{d}\right)^i.$$

□

However, now, we require a bound on the absolute value of the second and last eigenvalue, instead of just on the second eigenvalue. Fortunately, there also exist polynomial-time constructible graph families where $\lambda(G)$ is at most some constant for every graph G in the family.

Theorem 5.36 There exists a $d \in \mathbb{N}$ and a $\lambda \in \mathbb{R}$ with $\lambda < d$, such that there exists a polynomial-time constructible family of d-regular graphs $F = \{G_1, G_2, G_3, \ldots\}$, where, for every $G_i \in F$, we have $\lambda(G_i) \leq \lambda$. □

5.3.2 Constraint graph coloring problem

The problem we will be working with in the proof is called the *constraint graph coloring problem*, abbreviated as CGC. The input of the constraint graph coloring problem is a constraint graph over some alphabet. It can be seen as a generalisation of the 3-coloring problem of graphs where:

1. we allow any number of colors, where the set of colors is called the alphabet, and
2. instead of allowing only inequality constraints on the edges (in the 3-coloring problem, colors of nodes that are connected with an edge must be different), we allow any (binary) constraint on the edges.

Definition 5.37 A constraint graph over the alphabet Σ is a tuple (G, C), where:

- $G = (V, E)$ is an undirected multigraph, and
- the function $C : E \to \mathcal{P}(\Sigma \times \Sigma)$ associates with each edge, $e \in E$, a constraint $C(e) \subseteq \Sigma \times \Sigma$, where $C(e)$ consists of satisfying assignments to the two endpoints of the edge.

□

We say an assignment $a : V \to \Sigma$ satisfies a constraint graph $((V, E), C)$ over the alphabet Σ if for every edge $e = (u, v) \in E$, we have $(a(u), a(v)) \in C(e)$. We say a constraint graph $((V, E), C)$ over Σ is satisfiable if there exists an assignment $a : V \to \Sigma$ that satisfies the constraint graph.

Now, the constraint graph coloring problem over an alphabet Σ can be defined as follows:

$$\text{CGC}_\Sigma = \{\langle (G, C)\rangle \mid (G, C) \text{ is a constraint graph over } \Sigma \text{ and } (G, C) \text{ is satisfiable } \}.$$

We can prove that this language is NP-complete when the alphabet contains 3 or more symbols. To do this, we reduce from the language 3COLOR, which is a typical example of an NP-complete language.

$$\text{3COLOR} = \{\langle G\rangle \mid \text{the graph } G \text{ is colorable with 3 colors}\}.$$

Property 5.38 For every alphabet Σ with $|\Sigma| \geq 3$, CGC_Σ is NP-complete. □

Proof. We have to proof that CGC_Σ is in NP, and that every language in NP is polynomial-time reducible to CGC_Σ.

1. First we show that $\text{CGC}_\Sigma \in \text{NP}$ for every alphabet Σ, this is straightforward. A polynomial-time verifier can take as certificate a satisfying assignment for the constraint graph given as input.

2. Let Σ be an alphabet such that $|\Sigma| \geq 3$. Take 3 distinct symbols from Σ and call them r, g, and b respectively. We describe a reduction, f, from 3COLOR to CGC_Σ. On input a graph $G = (V, E)$, the reduction will output a constraint graph $f(G) = (G, C)$ over the alphabet Σ, where the constraints are such that, for every edge $e \in E$, the associated constraint is

$$C(e) = \{(r, b), (r, g), (b, r), (b, g), (g, r), (g, b)\},$$

that is, the inequality constraint. Then it is easy to see that if a graph G is 3-colorable, then the constraint graph $f(G)$ is satisfiable. While on the other hand, if G is not 3-colorable, then the constraint graph $f(G)$ is not satisfiable, note that $f(G)$ can not be satisfied by an assignment that uses elements in $\Sigma \setminus \{r, g, b\}$ because these elements do not occur in the constraints. Finally, f is clearly computable in polynomial time.

□

Now, let us consider the optimization version of this problem, called max-CGC_Σ, when using alphabet Σ. Similarly to the unsat-value of a constraint satisfaction problem, we define the unsat-value of a constraint graph $A = (G, C)$ as the minimum amount of unsatisfied constraints over all possible assignments, denoted by $\text{UNSAT}(A)$.

As we saw in Section 5.2, we know that Theorem 5.12, restated below, is equivalent to the PCP theorem.

Theorem 5.12 There exists a q and an alphabet Σ such that for every language $L \in \text{NP}$ there exists a polynomial-time reduction f of L to q-CSP_Σ such that:

- $x \in L \implies \text{UNSAT}(f(x)) = 0$, and
- $x \notin L \implies \text{UNSAT}(f(x)) \geq \frac{1}{2}$.

□

Now, because we know that CGC_Σ is NP-complete when $|\Sigma| \geq 3$, if we can show a polynomial-time reduction f of CGC_Σ to q-CSP, for some q, such that, on input a constraint graph $A = (G, C)$:

- if A is satisfiable, then $\text{UNSAT}(f(A)) = 0$, and
- if A is not satisfiable, then $\text{UNSAT}(f(A)) \geq \frac{1}{2}$,

then we have proven the PCP theorem.

But first, let us proof the following lemma.

Lemma 5.39 For every Σ with $|\Sigma| \geq 3$, there exists a polynomial-time reduction f of CGC_Σ to 2-CSP_Σ, that, on input a constraint graph A over Σ, outputs a 2-CSP_Σ instance C such that: $\text{UNSAT}(A) = \text{UNSAT}(C)$. □

Proof. There is a trivial correspondence between constraint graphs and binary CSP instances over the same alphabet. Then, any assignment for the constraint graph can be converted into one for the CSP instance such that the same number of constraints are satisfied, and vice-versa. Thus the unsat-value of the constraint graph is equal to the unsat-value of the CSP instance. □

This means that, to prove the PCP theorem, we only have to show, for some Σ with $|\Sigma| \geq 3$, a polynomial-time reduction of the CGC_Σ problem to itself such that, the unsat-value of satisfiable instances stays 0, while the unsat-value of unsatisfiable instances becomes becomes at least $\frac{1}{2}$.

5.3.3 Overview of the proof

In this subsection, we give an overview of the complete proof. Like we said in the previous subsection, to prove the PCP theorem, we show, for some alphabet Σ with $|\Sigma| \geq 3$, a reduction of CGC_Σ to itself such that the unsat-value of unsatisfiable constraint is greater than some constant. This reduction consists of 3 important steps, each with a corresponding lemma explained below. But first, let us define the 'size' of a graph as the sum of the number of nodes and edges, so, given a graph $G = (V, E)$ we write $\mathrm{size}(G) = |V| + |E|$. This proof given here is based on the proof by Dinur [9].

The first lemma we use, shows that any constraint graph can be converted, in polynomial time, to a constraint graph where the underlying graph is a d-regular expander graph, for some constant d. It is important that the size of the resulting graph is only a constant factor greater than the input graph. This lemma is used as a sort of preprocessing step, therefore we call it the preprocessing lemma.

Lemma 5.40 (The preprocessing lemma) There exist constants $d \in \mathbb{N}$, $\lambda \in \mathbb{R}$ and $\beta_1 \in \mathbb{R}$ with $\lambda < d$ and $0 < \beta_1 < 1$, such that there exists a polynomial-time algorithm that, on input a constraint graph (G, C) over an alphabet Σ, outputs a constraint graph (G', C') over Σ, such that:

- the graph G' is d-regular with at least one loop on every node;
- $\lambda(G') \leq \lambda$;
- $\mathrm{size}(G') = O(\mathrm{size}(G))$;
- $\mathrm{UNSAT}(G') \geq \beta_1 \cdot \mathrm{UNSAT}(G')$; and
- if $\mathrm{UNSAT}(G) = 0$, then $\mathrm{UNSAT}(G') = 0$ (in other words, when the input graph is satisfiable, so is the output graph).

□

The proof of this lemma is given in Subsection 5.3.4.

Note that when we apply the preprocessing step, the unsat-value of the resulting graph is less than or equal to the unsat-value of the original graph, and this while we are trying to increase the unsat-value, at least for unsatisfiable constraint graphs. However, this problem is solved using the next step, where we increase the unsat-value of unsatisfiable constraint graphs. Again, we do this while only increasing the size of the graph by a constant factor. Since we are increasing, or amplifying, the gap between the unsat-values of satisfiable and unsatisfiable constraint graphs, the next step is called the *gap amplification* step.

Lemma 5.41 (Gap amplification lemma) Given $\lambda \in \mathbb{R}$, $d \in \mathbb{N}$ and an alphabet Σ, with $\lambda < d$, there exists a $\beta_2 > 0$ and a polynomial-time algorithm that, on input a number $t \in \mathbb{N}$ and a constraint graph (G, C) where G is a d-regular graph with at least one loop on every node and where $\lambda(G) \leq \lambda$, outputs a constraint graph (G', C') over the alphabet $\Sigma^{d^{\lceil t/2 \rceil}}$ such that:

- $\mathrm{UNSAT}(G') \geq \beta_2 \sqrt{t} \cdot \min(\mathrm{UNSAT}(G), \frac{1}{t})$,
- if $\mathrm{UNSAT}(G) = 0$, then $\mathrm{UNSAT}(G') = 0$, and
- $\mathrm{size}(G') = O(d^t \cdot \mathrm{size}(G'))$.

□

The proof of this lemma is given in Subsection 5.3.5.

Note that the type of constraint graph required in the gap amplification step can be given by the preprocessing step. While this step does indeed amplify the gap in unsat-values, we are left with a problem: the output constraint graph uses a different alphabet, one whose size depends on:

1. the alphabet, Σ, of the input constraint graph,
2. the degree, d, of the input constraint graph, and

3. the value of the parameter t.

This is problematic because, as we see later, these steps will be run multiple times in order to amplify the gap up to constant. Then however, the alphabet could grow to a non-constant size, while we need a reduction that transforms an input constraint graph over some alphabet into an output constraint graph over the same alphabet. Fortunately, we can solve this problem using an *alphabet reduction* step, show in the lemma below.

Lemma 5.42 (Alphabet reduction lemma) There exists a $0 < \beta_3 < 1$, an alphabet Σ_0, with $|\Sigma_0| \geq 3$, and a polynomial-time algorithm that, on input a constraint graph (G, C) over some alphabet Σ, outputs a constraint graph (G', C') over Σ_0 such that:

- $\text{size}(G') = O(\text{size}(G))$,
- $\text{UNSAT}(G') \geq \beta_3 \cdot \text{UNSAT}(G)$, and
- if $\text{UNSAT}(G) = 0$, then $\text{UNSAT}(G') = 0$.

□

For details of the proof of this lemma, we refer to [9]. Note that, just like with the preprocessing step, the gap between unsat-values can decrease during this step.

Now, using the 3 steps described above, we can show that the unsat-value of a constraint graph can be doubled (up to a maximum), while only increasing the size of the graph by a constant factor. This is shown in the following lemma.

Lemma 5.43 There exists an alphabet Σ, with $|\Sigma| \geq 3$, and a constant $0 < \alpha < 1$, such that there exists a polynomial-time algorithm that, on input a constraint graph (G, C) over Σ, outputs a constraint graph (G', C') over Σ such that:

- $\text{size}(G') = O(\text{size}(G))$,
- $\text{UNSAT}(G') \geq \min(2 \cdot \text{UNSAT}(G), \alpha)$, and
- if $\text{UNSAT}(G) = 0$, then $\text{UNSAT}(G') = 0$.

□

Proof. Take Σ as the Σ_0 guaranteed to exist by Lemma 5.42. Now, the algorithm works as follows:

"On input a constraint graph (G, C) over Σ:

1. Apply the preprocessing step to (G, C) to obtain the constraint graph (G_1, C_1) over Σ.
2. Apply the gap amplification step to (G_1, C_1) to obtain the constraint graph (G_2, C_2) over $\Sigma^{d^{\lceil t/2 \rceil}}$, where d is the degree of G_1 (see below for the choice of t).
3. Finally, apply the alphabet reduction step to (G_2, C_2) to obtain the constraint graph (G_3, C_3) over Σ, this is our output. "

Now, we have that $\text{size}(G_1) = O(\text{size}(G))$, $\text{size}(G_2) = O(d^t \cdot \text{size}(G_1))$ and $\text{size}(G_3) = O(\text{size}(G_2))$. Thus, because we choose d and t as constants, $\text{size}(G_3) = O(\text{size}(G))$. We also know that $\text{UNSAT}(G_3) = 0$ if $\text{UNSAT}(G) = 0$. The only requirement left to prove is:

$$\text{UNSAT}(G_3) \geq \min(2 \cdot \text{UNSAT}(G), \alpha),$$

for some constant α. By Lemma 5.42, Lemma 5.41 and Lemma 5.40, we have, respectively:

$$\begin{aligned}
\text{UNSAT}(G_3) &\geq \beta_3 \cdot \text{UNSAT}(G_2) \\
&\geq \beta_3 \cdot \beta_2 \sqrt{t} \cdot \min(\text{UNSAT}(G_1), \frac{1}{t}) \\
&\geq \beta_3 \cdot \beta_2 \sqrt{t} \cdot \min(\beta_1 \cdot \text{UNSAT}(G), \frac{1}{t})
\end{aligned}$$

Now, if we take

$$t = \left\lceil \left(\frac{2}{\beta_1\beta_2\beta_3}\right)^2 \right\rceil,$$

then we get

$$\begin{aligned} \text{UNSAT}(G_3) &\geq \beta_3 \cdot \beta_2\sqrt{t} \cdot \min(\beta_1 \cdot \text{UNSAT}(G), \frac{1}{t}) \\ &\geq \frac{2}{\beta_1} \cdot \min(\beta_1 \cdot \text{UNSAT}(G), \frac{1}{t}) \\ &= \min(2 \cdot \text{UNSAT}(G), \frac{2}{t\beta_1}) \end{aligned}$$

And thus we get

$$\alpha = \frac{2}{t\beta_1} = \frac{2}{\left\lceil \left(\frac{2}{\beta_1\beta_2\beta_3}\right)^2 \right\rceil \cdot \beta_1}.$$

Note that α is a constant because β_1, β_2 and β_3 are constants. □

5.3.4 Proof of the preprocessing lemma

The preprocessing step exists of 2 steps itself. In the first step, we convert the input constraint graph into a d-regular constraint graph for some constant d. In the second step, we convert the d-regular constraint graph into an expander graph.

Converting to a regular graph

Here, we show a method to convert an input constraint graph into a d-regular constraint graph, for some d, such that if the input graph is satisfiable, so is the output graph, and when the input graph is unsatisfiable, the unsat-value of the output graph is at most some factor smaller than the unsat-value of the input graph, where the factor only depends on d, and thus is constant in function of the input graph. We start by describing the method and the intuition behind it, and prove its correctness afterwards.

First, let us show how we convert an input graph into a regular graph, ignoring the constraints for now. For each node v in the original graph, we create a group $g(v)$ of nodes in the output graph. The number of nodes in the group $g(v)$ equals the degree of v. In other words, with each edge e incident to v, we associate a node (v, e) in $g(v)$. Formally, when given as input a graph $G = (V, E)$, the output graph $G' = (V', E')$ will have the node set

$$V' = \{g(v) \mid v \in V\},$$

where

$$g(v) = \{(v, e) \mid e \in E \text{ and } e \text{ is incident to } v\}.$$

For the edges, we distinguish between two (multi)sets of edges, E_1 and E_2, such that $E' = E_1 \cup E_2$.

- E_1 consists of edges between nodes of different groups. We connect two nodes of different groups if and only if these two nodes are associated to the same edge in the original graph. Formally:

  $$E_1 = \{\{(v, e), (u, e)\} \mid e = \{v, u\} \text{ and } e \in E\}.$$

  Note that $\{(v, e), (u, e)\}$ in the above equation is the (undirected) edge between the nodes (v, e) and (u, e), where (v, e) is the node in $g(v)$ associated with edge e (of the original graph G), as explained before.

- E_2 consists of edges between nodes of the same group. We connect nodes of a group such that every node in the group is connected to d_0 other nodes in the same group, for some d_0. We explain more specifically how to do this when considering constraints, later on.

It should be clear now, that the resulting graph is d-regular, with $d = d_0 + 1$. We can also already look at the size of the output graph. The number of nodes in the output graph $G' = (V', E')$, when the input graph is $G = (V, E)$, is at most the number of edges in G multiplied by two, note that this is exact when there are no loops in G. The number of edges in G' equals at most the amount of nodes in G' times the degree of G' (because G' is regular). So we have:

$$|V'| \leq 2 \cdot |E|$$
$$|E'| \leq d \cdot |V'|$$

where d is the degree of G'. Now, consider the size of G':

$$\begin{aligned} \text{size}(G') &= |V'| + |E'| \\ &\leq 2 \cdot |E| + d \cdot |V'| \\ &\leq (2 + 2d) \cdot |E| \\ &\leq (2 + 2d) \cdot \text{size}(G), \end{aligned}$$

proving that the size of the output graph is only a constant factor larger than the size of the input graph (assuming d is a constant, at least).

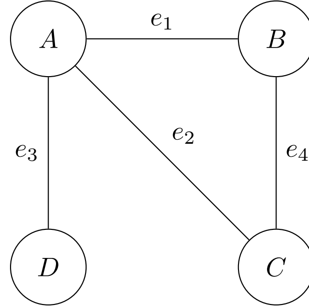


Figure 5.5: An example input graph, the edges are given names e_1 to e_4.

Before considering the constraints on the resulting graph, let us give an example of such output graph. Take the graph shown in Figure 5.5, with nodes $\{A, B, C, D\}$ and edges $e_1 = \{A, B\}$, $e_2 = \{A, C\}$, $e_3 = \{A, D\}$ and $e_4 = \{B, C\}$. Now consider, for example, the group $g(A)$, because the node A is incident to the three edges e_1, e_2 and e_3, the group $g(A)$ will consist of the three nodes (A, e_1), (A, e_2) and (A, e_3). Because $e_1 = \{A, B\}$, the node (A, e_1) is connected to the node (B, e_1), similarly, because $e_2 = \{A, C\}$, nodes (A, e_2) and (C, e_2) are connected, and so on. The output graph is shown in Figure 5.6, the edges connecting nodes within the same group are omitted.

Now, let us consider the constraints we put on the output graph. We keep working with the same alphabet as the input constraint graph. The constraints on edges between nodes of different groups will have the same constraint as the associated edge in the input constraint graph. Thus, for example, if we have the input constraint graph (G, C) where G is the graph shown in Figure 5.5 and we have the output graph (G', C') with G' the graph as shown in Figure 5.6, the constraint on the edge between nodes (A, e_1) and (B, e_1) will be the same as the constraint of e_1, that is, $C'((A, e_1), (B, e_1)) = C(e_1)$.

For the edges between nodes in the same group, we use the equality constraint. As an example, the equality constraint over the alphabet $\{r, g, b\}$ is:

$$\{(r, r), (g, g), (b, b)\}.$$

In general, the equality constraint over an alphabet Σ is

$$\{(\sigma_1, \sigma_2) \mid \sigma_1, \sigma_2 \in \Sigma \text{ and } \sigma_1 = \sigma_2\}.$$

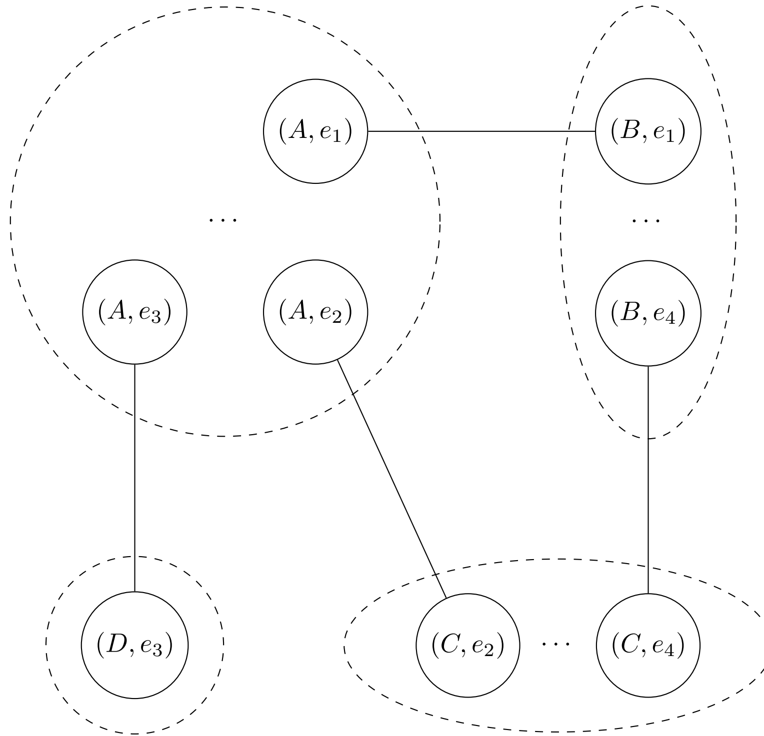


Figure 5.6: An example of the output graph obtained after the conversion of the graph shown in Figure 5.5 to a regular graph. The dashed ellipses represent the groups and their inner edges have been omitted for now.

It is quite straightforward to see that if the input constraint graph is satisfiable, then the output constraint graph is also satisfiable. Given a satisfying assignment $a : V \to \Sigma \times \Sigma$ for the input graph, the following assignment $a' : V' \to \Sigma \times \Sigma$ will satisfy the output graph:

$$a'((v, e)) = a(v).$$

Because it assigns the same symbol to all nodes in a group, the constraints on edges between nodes in the same group, which are equality constraints, are satisfied. Secondly, because the constraints on nodes of different groups are the same as the constraints on the associated edges, and the symbols assigned to nodes in the output graph are the same as the symbols assigned to the associated nodes in the input graph, the constraints between nodes in different groups are also satisfied.

However, what can we say about the unsat-value of the output graph when the input graph is not satisfiable? Remember that we want the unsat-value of the output to be at most a constant fraction smaller than the unsat-value of the input. So, it should not be possible to create an assignment that a violates 'much' smaller fraction of constraints than the unsat-value of the input graph. Say we have an input constraint graph $A = ((V, E), C)$ and d-regular output constraint graph $B = ((V', E'), C')$. Now, assignments for the output graph that assign the same symbol to all nodes in the same group, will only violate edges between nodes of different groups, but will violate at least a

$$\frac{|E| \cdot \text{UNSAT}(A)}{|E'|}$$

fraction of the constraints. Because we know that $|E'| \leq 2d \cdot |E|$, we have

$$\frac{|E| \cdot \text{UNSAT}(A)}{|E'|} \geq \frac{1}{2d} \cdot \text{UNSAT}(A).$$

So, for these assignments, the unsat-value can only decrease by a constant factor (assuming d is a constant), which is what we want.

Unfortunately, we also have to consider assignments to the output constraint graph that do not assign the same symbol to all nodes in the same group. Note that we have not yet specified how to place edges between nodes of the same group. Consider an assignment to the output constraint graph, this assignment will assign a symbol of the alphabet to each node in the output graph. More intuitively, an assignment to a node (v, e) can be seen as an 'opinion' on the assignment of v, in the original graph. Then, we want to make sure that if there is a lot of disagreement on the assignment of a certain node, than there are also a lot of edges (in the output graph) whose constraints are violated (that is, not satisfied). This is the case when, in some group $g(v)$, every set of nodes (in the output graph) that have the same opinion on the assignment to v, are connected to a lot of nodes that have a different opinion. This intuitive idea is captured by the 'well-connectedness' of expander graphs. Therefore, we use an expander graph to fill in the edges inside a group.

Now that we have described the first step of the preprocessing step, we prove its correctness in the following lemma.

Lemma 5.44 There exists a constant $d \in \mathbb{N}$ with $d > 1$ and a constant $c \in \mathbb{R}$, with $0 < c \leq 1$, such that there exists a polynomial-time algorithm that, on input a constraint graph $A = (G, C)$ over alphabet Σ , outputs a constraint graph $B = (G', C')$ over alphabet Σ, such that:

- the graph G' is d-regular,
- $\text{size}(G') = O(\text{size}(G))$,
- $\text{UNSAT}(B) \geq c \cdot \text{UNSAT}(A)$, and
- if $\text{UNSAT}(A) = 0$, then $\text{UNSAT}(B) = 0$.

□

Proof. Let $A = (G, C)$, with $G = (V, E)$, be the input constraint graph. The algorithm will output the constraint graph $B = (G', C')$, with $G' = (V', E')$, in the following way.

- The nodes are
$$V' = \{g(v) \mid v \in V\},$$
with
$$g(v) = \{(v, e) \mid e \in E \text{ and } e \text{ is incident to } v\}.$$
- The edges are $E' = E_1 \cup E_2$, where
$$E_1 = \{\{(v, e), (u, e)\} \mid e = \{v, u\} \text{ and } e \in E\},$$
and
$$E_2 = \{E(X_v) \mid v \in V\},$$
where $E(X_v)$ are the edges of the graph X_v, which is a d_0-regular expander graph with $h(X_v) \geq h$, for some constants d_0 and h, with X_v having as nodes $g(v)$. Note that such graphs are guaranteed to exist and constructible in polynomial time by Theorem 5.20.
- The constraints are $C' : V' \to \Sigma \times \Sigma$, such that
$$C'(\{(v_1, e_1), (v_2, e_2)\}) = \begin{cases} c_= & \text{if } v_1 = v_2 \\ C(e_1) & \text{if } v_1 \neq v_2 \end{cases}$$
where $c_=$ denotes the equality constraint. The first case specifies the constraints of edges between nodes in the same group, while the second case specifies the constraints of edges between nodes of different groups. Note that in the second case, when $v_1 \neq v_2$, the edges are equal, $e_1 = e_2$, otherwise $\{(v_1, e_1), (v_2, e_2)\}$ would not be an edge.

Now, it is clear that the resulting graph is d-regular, with $d = d_0 + 1$. Also, we have already shown that $\text{size}(G') = O(\text{size}(G)$

When $\text{UNSAT}(A) = 0$, there exists an assignment $a : V \to \Sigma \times \Sigma$ that satisfies A. If we take the assignment $a' : V' \to \Sigma \times \Sigma$ where

$$a'((v, e)) = a(v),$$

then a' is a satisfying assignment for B, thus $\text{UNSAT}(B) = 0$.

Finally, we have to prove that

$$\text{UNSAT}(B) \geq c \cdot \text{UNSAT}(A),$$

for some constant c.

Define $\text{UNSAT}_a(A)$ to be the fraction of constraints violated by assignment a for the constraint graph A. Clearly, $\text{UNSAT}_a(A) \geq \text{UNSAT}(A)$ for all a, and

$$\text{UNSAT}(A) = \min_a \text{UNSAT}_a(A).$$

Now, take an assignment a' for B with minimal unsat-value, that means $\text{UNSAT}_{a'}(B) = \text{UNSAT}(B)$. Using this, consider the assignment a for A, where

$$a(v) = \arg\max_\sigma |\{(u, e) \mid u = v \text{ and } a'((u, e)) = \sigma\}|,$$

in other words, the assignment a takes the 'popular opinion' of a' for every node v, that is, the syumbol assigned the most to nodes in the group $g(v)$. Note that, for a node (u, e), if $u = v$, then $(u, e) \in g(v)$. Let $F \subseteq E$ be the edges of A that are violated by a and let $F' \subseteq E'$ be the edges of B that are violated by a'. Then we have

$$\frac{|F|}{|E|} = \text{UNSAT}_a(A) \geq \text{UNSAT}(A)$$

and

$$\frac{|F'|}{|E'|} = \text{UNSAT}_{a'}(B) = \text{UNSAT}(B).$$

Define the set $S \subseteq V'$ as

$$S = \bigcup_{v \in V} \{(u, e) \mid u = v \text{ and } a'((u, e)) \neq a(v)\},$$

in other words, S is the set of nodes in B that do not agree with the popular opinion. Now, consider an edge $e = \{v, u\}$ in A, and the corresponding edge $e' = \{(v, e), (u, e)\}$ in B. When e is in F, then either $e' \in F'$ or one of the two endpoints is in S. This is true because when e' is not in F' (thus not violated by a'), and both endpoints of e' are in S (that is, they both agree with the popular opinion), then e is not violated by the popular opinion assignment a because the constraint on e and e' are the same. This means that, for every edge $e \in F$, we have at least one element in F' or S, thus

$$|F'| + |S| \geq |F| \geq |E| \cdot \text{UNSAT}(A).$$

Now, we distinguish between two cases.

- When $|F'| \geq \frac{|F|}{2}$, then we have:

$$|F'| \geq \frac{|F|}{2} \geq \frac{|E|}{2} \cdot \text{UNSAT}(A).$$

  If we divide this by $|E'|$ we get:

$$\begin{aligned} \frac{|F'|}{|E'|} &= \text{UNSAT}(B) \\ &\geq \frac{|E|}{2 \cdot |E'|} \cdot \text{UNSAT}(A) \\ &\geq \frac{|E|}{4d \cdot |E|} \cdot \text{UNSAT}(A) \\ &\geq \frac{1}{4d} \cdot \text{UNSAT}(A). \end{aligned}$$

- On the other hand, if $|F'| < \frac{|F|}{2}$, then $|S|$ must be greater than $\frac{|F|}{2}$. Define $S_{v,\sigma}$ as:

  $$S_{v,\sigma} = \{(u,e) \in S \mid u = v \text{ and } a((u,e)) = \sigma\},$$

  in other words, $S_{v,\sigma}$ is the set of nodes in group $g(v)$, with opinion σ where σ is different from the popular opinion. Then $|S_{v,\sigma}| \leq \frac{|g(v)|}{2}$, because otherwise, σ would be the popular opinion. Now, we use the fact that edges connecting nodes of the same group form an expander graph. So, we know that the edge expansion ratio of such graph is greater than some constant h. This means that the edge boundary of $S_{v,\sigma}$ (relative to the group $g(v)$) is at least $h \cdot |S_{v,\sigma}|$, or

  $$E(S_{v,\sigma}, \overline{S_{v,\sigma}}) \geq h \cdot |S_{v,\sigma}|,$$

  where $\overline{S_{v,\sigma}} = g(v) \setminus S_{v,\sigma}$. Every edge that connects a node in $S_{v,\sigma}$ to a node in $g(v)$ but outside $S_{v,\sigma}$, is violated by a' because all the edges connecting nodes in the same group have the equality constraint while the two nodes must have a different assignment. So, in a group $g(v)$, there are at least

  $$\sum_{\sigma} \frac{h}{2} \cdot |S_{v,\sigma}|$$

  constraints violated, and in the whole graph B, there are at least

  $$\begin{aligned} |F'| &\geq \sum_{v} \sum_{\sigma} \frac{h}{2} \cdot |S_{v,\sigma}| \\ &= \frac{h}{2} \cdot \sum_{v} \sum_{\sigma} |S_{v,\sigma}| \\ &= \frac{h}{2} \cdot |S| \\ &\geq \frac{h}{4} \cdot |F|. \end{aligned}$$

  If we divide by $|E'|$ we get:

  $$\begin{aligned} \frac{|F'|}{|E'|} &= \text{UNSAT}(B) \\ &\geq \frac{h \cdot |F|}{4 \cdot |E'|} \\ &\geq \frac{h \cdot |F|}{8d \cdot |E|} \\ &\geq \frac{h}{8d} \cdot \text{UNSAT}(A). \end{aligned}$$

So, if we take

$$c = \min(\frac{1}{4d}, \frac{h}{8d}),$$

then we have $\text{UNSAT}(B) \geq c \cdot \text{UNSAT}(A)$. □

Converting to an expander graph

Here we show a method to convert a constraint graph produced by the previous step into a constraint graph of which the graph is an expander graph. To do this, we require another property of the eigenvalues of a matrix, being that the second largest eigenvalue in absolute value is equal to what is known as the Rayleigh quotient, as shown in the following property.

Property 5.45 Given matrix A, with eigenvalues $\lambda_1 \leq \lambda_2, \cdots \leq \lambda_n$, then:

$$\max(|\lambda_2|, |\lambda_n|) = \max_{\|x\|, x \perp \vec{1}} |\langle x, Ax \rangle|,$$

where $\|x\|$ denotes the norm of the vector x, $x \perp \vec{1}$ means that x is orthogonal to the vector $\vec{1}$ (whose components are all 1), and $\langle x, Ax \rangle$ is the dot product of x and Ax. □

The next property follows directly from this.

Property 5.46 Given a d-regular graph G with adjacency matrix $A = A(G)$, with eigenvalues $\lambda_1 \leq \lambda_2, \cdots \leq \lambda_n$, then:

$$\lambda(G) = \max_{\|x\|, x \perp \vec{1}} |\langle x, Ax \rangle|,$$

□

Lemma 5.47 There exist constants $d_c \in \mathbb{N}$ with $d > 1$, and $\lambda \in \mathbb{R}$ with $0 < \lambda < d_c$, such that there exists a polynomial-time algorithm that, on input a d-regular constraint graph $A = (G, C)$ over alphabet Σ, outputs a constraint graph $B = (G', C')$ over alphabet Σ, such that:

- the graph G' is $(d_c + d + 1)$-regular,
- every node in G' has at least one loop,
- $\lambda(G') \leq \lambda + d + 1 < d_c + d + 1$,
- $\text{size}(G') = O(\text{size}(G))$,
- $\text{UNSAT}(B) \geq \frac{d}{d+d_c+1} \cdot \text{UNSAT}(A)$, and
- if $\text{UNSAT}(A) = 0$, then $\text{UNSAT}(B) = 0$.

□

Proof. Let $A = (G, C)$, with $G = (V, E)$, be the input constraint graph. The algorithm will output the constraint graph $B = (G', C')$, with $G' = (V', E')$, in the following way.

- The nodes stay the same, that means $V' = V$.
- The edges are $E' = E_1 \cup E_2 \cup E_3$. First we take the edges of A, thus $E_1 = E$. Then, we add a loop to every node, thus
$$E_2 = \{(v, v) \mid v \in V\}.$$
Finally, we add the edges of an expander graph, so
$$E_3 = E(X_n),$$
where X_n is a d_c-regular expander graph with n nodes where $\lambda(X_n) \leq \lambda$ for some constant λ, and n equals the number of nodes in G, $n = |V|$. Note that such graphs are guaranteed to exist and constructible in polynomial time by Theorem 5.36.
- The constraints are defined as follows:
$$C'(e) = \begin{cases} C(e) & \text{if } e \in E \\ \Sigma \times \Sigma & \text{if } e \notin E \end{cases}$$
This means that the constraints on the edges taken from the input graph remain the same, while the constraints on the added edges are always satisfied.

Note that, when we say $E' = E_1 \cup E_2 \cup E_3$, we are taking the union of multisets, which means all edges are simply added together. To clarify this, we can look at the adjacency matrix of the output graph, $A(G')$, which then equals $A(G) + I + A(X_n)$, where I denotes the identity matrix, which has 1 on its diagonal elements and 0 everywhere else.

It is easy to see that the resulting graph is $(d_c + d + 1)$-regular, and that the resulting graph has at least one loop on every node. We can write $\lambda(G')$ as:

$$\begin{aligned}
\lambda(G') &= \max_{\|x\|, x \perp \vec{1}} |\langle x, A(G')x \rangle| \\
&\leq \max_{\|x\|, x \perp \vec{1}} |\langle x, A(G)x \rangle| + \max_{\|x\|, x \perp \vec{1}} |\langle x, Ix \rangle| + \max_{\|x\|, x \perp \vec{1}} |\langle x, A(X_n)x \rangle| \\
&\leq \lambda(G) + 1 + \lambda(X_n) \\
&\leq d + 1 + \lambda.
\end{aligned}$$

For the size of the resulting graph, we have:

$$\begin{aligned}\text{size}(G') &= |V'| + |E'| \\ &= |V| + (|E| + |V| + d_c \cdot |V|) \\ &\leq |E| + (d_c + 2) \cdot |V| \\ &\leq (d_c + 2) \cdot (|V| + |E|)\end{aligned}$$

Thus $\text{size}(G') = O(\text{size}(G))$.

Now, let us look the unsat-value of the output constraint graph. If $\text{UNSAT}(A) = 0$, then there exists a satisfying assignment for A, and because we only added edges with constraints that are always satisfied, the same assignment will also satisfy B, thus $\text{UNSAT}(B) = 0$. On the other hand, because we added $d_c + 1$ edges which are always satisfied, we have:

$$\text{UNSAT}(B) = \frac{d}{d + d_c + 1} \cdot \text{UNSAT}(A).$$

□

Combining the steps

Now that we have proven the correctness of both steps, we can prove the preprocessing lemma, restated below.

Lemma 5.40 (The preprocessing lemma) There exist constants $d \in \mathbb{N}$, $\lambda \in \mathbb{R}$ and $\beta_1 \in \mathbb{R}$ with $\lambda < d$ and $0 < \beta_1 < 1$, such that there exists a polynomial-time algorithm that, on input a constraint graph (G, C) over an alphabet Σ, outputs a constraint graph (G', C') over Σ, such that:

- the graph G' is d-regular with at least one loop on every node;
- $\lambda(G') \leq \lambda$;
- $\text{size}(G') = O(\text{size}(G))$;
- $\text{UNSAT}(G') \geq \beta_1 \cdot \text{UNSAT}(G')$; and
- if $\text{UNSAT}(G) = 0$, then $\text{UNSAT}(G') = 0$ (in other words, when the input graph is satisfiable, so is the output graph).

□

Proof. The algorithm works as follows. Given a constraint graph $A = (G, C)$ over alphabet Σ as input, we apply Lemma 5.44 to obtain the constraint graph $B = (G_1, C_1)$ over Σ. Then, we apply Lemma 5.47 to obtain the constraint graph $C = (G_2, C_2)$, this is the output constraint graph. The properties listed below follow immediately from Lemma 5.44 and Lemma 5.47.

- The graph G_2 is d-regular, for some constant d, and has at least one loop on every node;
- $\lambda(G_2) \leq \lambda$, for some constant $\lambda < d$;
- $\text{size}(G_2) = O(\text{size}(G)$;
- if $\text{UNSAT}(A) = 0$, then $\text{UNSAT}(C) = 0$.

The only thing left to prove is that $\text{UNSAT}(C) \geq \beta_1 \cdot \text{UNSAT}(A)$, for some constant β_1 with $0 < \beta_1 < 1$. By Lemma 5.44, we have:

$$\text{UNSAT}(B) \geq c \cdot \text{UNSAT}(A),$$

where c is a constant with $0 < c \leq 1$. By Lemma 5.47, we have:

$$\text{UNSAT}(C) \geq \frac{d}{d + d_c + 1} \cdot \text{UNSAT}(B),$$

where d_c is a constant. Combining this, we get:

$$\text{UNSAT}(C) \geq \frac{d \cdot c}{d + d_c + 1} \text{UNSAT}(A).$$

So if we take $\beta_1 = \frac{d \cdot c}{d+d_c+1}$, we get the desired result. Note that β_1 is constant because d, c, and d_c are all constant. □

5.3.5 Proof of the gap amplification lemma

To perform gap amplification on an input constraint graph, we perform a technique known as *graph powering* to the input graph.

Definition 5.48 (Graph powering) Given as input a d-regular constraint graph $A = ((V, E), C)$ over alphabet Σ and a number $t \in \mathbb{N}$ with $t > 1$, we define the constraint graph $A^t = ((V', E'), C')$ over $\Sigma^{d^{\lceil t/2 \rceil}}$ as follows.

- The nodes stay the same, $V' = V$.
- The edges of the output graph correspond to t-step walks in the input graph, that is, the number of edges between two nodes u and v in the output graph equals the number of t-step walks between u and v in the input graph.
- For the constraints let us first look at the alphabet, which is $\Sigma^{d^{\lceil t/2 \rceil}}$. An assignment to a node u in the output constraint graph, can then be seen as an opinion of u on the symbols assigned to the nodes in its neighbourhood. We define the neighbourhood of a node u as the set of nodes that are reachable with a $\lceil t/2 \rceil$-step walk, denoted by $\Gamma(u)$. Then, a symbol in $\Sigma^{d^{\lceil t/2 \rceil}}$ can be seen as assignment itself, assigning a symbol (in Σ) to each node in $\Gamma(u)$, assuming we use some ordering on the nodes. When a node u is assigned a symbol $\sigma \in \Sigma^{d^{\lceil t/2 \rceil}}$, we denote the opinion of u on a node v in its neighbourhood, by σ_v. Note that there are exactly $d^{\lceil t/2 \rceil}$ walks of length $\lceil t/2 \rceil$ starting from the same node. However, since some walks may end in the same node, we have $\Gamma(u) \leq d^{\lceil t/2 \rceil}$, for every node u.

  Now, we want the constraint of an edge $e = \{u, v\}$ in the output graph to be satisfied if the symbols that u and v assign to their neighbours satisfy the edges in the original graph. Formally, when $e = \{u, v\} \in E'$, we have $(\sigma_1, \sigma_2) \in C'(e)$, for $\sigma_1, \sigma_2 \in \Sigma^{d^{\lceil t/2 \rceil}}$, if and only if for all neighbours $u' \in \Gamma(u)$ and $v' \in \Gamma(v)$ such that $\{u', v'\} \in E$, we have $(\sigma_{1,u'}, \sigma_{2,v'}) \in C(\{u', v'\})$. In other words, an edge $\{u, v\}$ in the output graph is satisfied if every edge (in the input graph) between a neighbour of u and a neighbour of v is satisfied by the symbols that u and v assign to these neighbours (that is, their opinion on them).

□

Some notes are in order here:

- When computing the output constraint graph $A^t = (G', C')$ where $A = (G, C)$, we can obtain the adjacency matrix of G' as the adjacency matrix of G to the power t, that is $A(G') = A(G)^t$.
- We defined the neighbourhood of a node u as the set of nodes that can be reached by a walk of exactly $\lceil t/2 \rceil$ steps, starting in u. This means that a node adjacent to u, that is, they are connected by an edge, is not guaranteed to be a member of the neighbourhood of u. This, of course, seems counter-intuitive. However, in the gap amplification lemma, we only apply the graph powering technique to graphs with at least one loop on every node. In that case, the neighbourhood of a node is the set of nodes that can be reached by a walk of at most $\lceil t/2 \rceil$ steps (instead of exactly). This is true because any walk with less than $\lceil t/2 \rceil$ steps can be converted to one with exactly $\lceil t/2 \rceil$ steps with the same end node, by repeatedly following the loop on the end node.
- When a node u is in the neighbourhood of a node v, then v is in the neighbourhood of u.

Now, we prove the following lemma, which immediately implies the gap amplification lemma, Lemma 5.41.

Lemma 5.49 Given $\lambda \in \mathbb{R}$, $d \in \mathbb{N}$ and an alphabet Σ, with $\lambda < d$, there exists a $\beta_2 > 0$, such that for every number $t \in \mathbb{N}$, with $t > 1$, and for every constraint graph $A = (G, C)$ over Σ, where G is a d-regular graph with at least one loop on every node and where $\lambda(G) \leq \lambda$, the following properties hold for the constraint graph $B = A^t = (G', C')$:

- $\mathrm{size}(G') = O(d^t \cdot \mathrm{size}(G))$;
- if $\mathrm{UNSAT}(A) = 0$, then $\mathrm{UNSAT}(B) = 0$;
- $\mathrm{UNSAT}(B) \geq \beta_2\sqrt{t} \cdot \min(\mathrm{UNSAT}(A), \frac{1}{t})$.

□

Proof. Let us start by proving the first property:

$$\mathrm{size}(G') = O(d^t \cdot \mathrm{size}(G)).$$

To do this, we need two observations:

1. In the original graph, G, the number of nodes is at most 2 times the number of edges, $|V| \leq 2 \cdot |E|$, this is true for any connected graph, and our graph is connected because we have $\lambda(G) \leq \lambda < d$;
2. The number of t-step walks starting from a single node in the original graph is exactly d^t. Thus the number of edges in the output graph is at most d^t times the number of nodes, $|E'| = d^t \cdot |V'| = d^t \cdot |V|$.

Using these observations, we get:

$$|E'| = d^t \cdot |V| \leq 2d^t \cdot |E|.$$

Filling this into the size of G' gives us:

$$\begin{aligned}
\mathrm{size}(G') &= |V'| + |E'| \\
&= |V| + |E'| \\
&\leq |V| + d^t \cdot |V| \\
&\leq (d^t + 1) \cdot (|V| + |E|) \\
&= O(d^t \cdot \mathrm{size}(G)).
\end{aligned}$$

Next, we prove the second property: if $\mathrm{UNSAT}(A) = 0$, then $\mathrm{UNSAT}(B) = 0$. Assume $\mathrm{UNSAT}(A) = 0$, and take a satisfying assignment $a : V \to \Sigma$. Then, we construct the assignment $a' : V \to \Sigma^{d^{\lceil t/2 \rceil}}$ as follows. For every node u we assign a symbol $a'(u) = \sigma \in \Sigma^{d^{\lceil t/2 \rceil}}$ such that for every v in $\Gamma(u)$, we have $\sigma_v = a(v)$. In other words, we let the opinion of a node u on a neighbour v be the symbol that a assigns to v. Now, a' is clearly a satisfying assignment for the output constraint graph, B.

The final property that we have to prove is:

$$\mathrm{UNSAT}(B) \geq \beta_2\sqrt{t} \cdot \min(\mathrm{UNSAT}(A), \frac{1}{t}).$$

To do this, we make use of Lemma 5.50, shown below. This lemma says that for every assignment a' we take for B, there exists an assignment a for A such that

$$\mathrm{UNSAT}_{a'}(B) \geq \beta_2\sqrt{t} \cdot \min(\mathrm{UNSAT}_a(A), \frac{1}{t}).$$

So, if we take a' as the assignment with minimal unsat-value, we get

$$\mathrm{UNSAT}(B) = \mathrm{UNSAT}_{a'}(B) \geq \beta_2\sqrt{t} \cdot \min(\mathrm{UNSAT}_a(A), \frac{1}{t}) \leq \beta_2\sqrt{t} \cdot \min(\mathrm{UNSAT}(A), \frac{1}{t}),$$

which concludes this proof. □

The following lemma is really the core of the gap amplification lemma, because it shows that the gap in unsat-values is effectively increased by applying the graph powering operation.

Lemma 5.50 Given $\lambda \in \mathbb{R}$, $d \in \mathbb{N}$ and an alphabet Σ, with $\lambda < d$, there exists a $\beta_2 > 0$, such that for every number $t \in \mathbb{N}$, with $t > 1$, and for every constraint graph $A = (G, C)$ over Σ, where G is a d-regular graph with at least one loop on every node and where $\lambda(G) \leq \lambda$, we have that for every assignment $a' : V \to \Sigma^{d^{\lceil t/2 \rceil}}$ for the constraint constraint graph $B = A^t = (G', C')$, there exists an assignment $a : V \to \Sigma$, such that:

$$\mathrm{UNSAT}_{a'}(B) \geq \beta_2 \sqrt{t} \cdot \min(\mathrm{UNSAT}_a(A), \frac{1}{t}).$$

□

Proof. We first give an intuitive explanation of the proof. Similar to what we did in the proof of the first step of the preprocessing lemma, given an assignment for B, we construct a popular opinion assignment for A, which has a lower unsat-value. Say we are given an assignment a' for B, then the popular opinion assignment for A is defined as:

$$a(u) = \arg\max_{\sigma \in \Sigma} |\{v \mid v \in \Gamma(u) \text{ and } a'(v) = \sigma\}|.$$

In other words, to every node u, we assign the symbol which most nodes in the neighbourhood of u have as opinion on u. Now, we can relate the fraction of edges in B that are violated to the fraction of edges in A that are violated. Say we were to chose t edges from the graph of A, uniformly at random, then the probability that we chose one or more edges violated by a is approximately equal to $t \cdot \mathrm{UNSAT}_a(A)$ (this is a very rough estimate, but we do this to keep the intuitive explanation simpler). Now, remember that our input graph is an expander graph, and that random walks on expander graphs converge rapidly to the uniform distribution, as seen in Theorem 5.35. Now, taking a random t-step walk in A is equivalent to taking a random edge in B. And when taking a random edge in B, the probability that we chose an edge violated by a' equals $\mathrm{UNSAT}_{a'}(B)$. On the other hand, that probability will be close to the estimate we gave for uniform sampling, $t \cdot \mathrm{UNSAT}_a(A)$. This is the intuitive idea of how to relate the fraction of violated edges of A by a to the fraction of violated edges of B by a'.

Now, we prove the lemma. Let $A = (G, C)$ be a constraint graph over alphabet Σ where G is a d-regular constraint and $\lambda(G) \leq \lambda$ for some constant $\lambda < d$. Let $B = A^t = (G', C')$ for some number $t > 1$. Let a' be an assignment for B and let a be the popular assignment of a', that is

$$a(u) = \arg\max_{\sigma \in \Sigma} |\{v \mid v \in \Gamma(u) \text{ and } a'(v) = \sigma\}|.$$

Let $F \subseteq E$ be a subset of edges which constraints are violated by a, such that:

- if $\mathrm{UNSAT}_a(A) < \frac{1}{t}$, then we have $\frac{|F|}{|E|} = \mathrm{UNSAT}_a(A)$, this means $|F|$ consists of all violated edges;
- otherwise, take F such that $|F| = \left\lfloor \frac{|E|}{t} \right\rfloor$.

Now, we clearly have

$$\frac{|F|}{|E|} \leq \min(\mathrm{UNSAT}_a(A), \frac{1}{t}).$$

This will be useful later on.

With each edge in E', we can associate a t-step walk in the graph of A, as can be seen from the description of the graph powering operation. We denote the walk associated with an edge $e \in E'$ by $w(e) = (v_0, v_1, \dots, v_t)$. We say a walk $(v_0, v_1, \dots v_t)$ is hit by its i-th edge if

- $\{v_{i-1}, v_i\} \in F$, and
- $a'(v_0)_{v_{i-1}} = a(v_{i-1})$ and $a'(v_t)_{v_i} = a(v_i)$.

Now, if for some edge $e \in E'$, the associated walk $w(e)$ is hit on any edge, then e is violated by a'. This is true because if $\{v_{i-1}, v_i\} \in F$, then

$$(a(v_{i-1}), a(v_i)) \notin C(\{v_{i-1}, v_i\}),$$

and thus

$$(a'(v_0)_{v_{i-1}}, a(v_t)_{v_i}) \notin C(\{v_{i-1}, v_i\}),$$

which implies that $(a'(v_0), a'(v_t)) \notin C'(e)$, by definition of the constraints C', which means that e is violated by a'. Let us focus on the middle part of the walks, that is, where the indices are in

$$I = \{i \mid \frac{t}{2} - \sqrt{t} < i \leq \frac{t}{2} + \sqrt{t}\}.$$

We define $N(e)$ as the number of edges hit in the middle part of the walk associated with e, or

$$N(e) = |\{i \mid i \in I \text{ and } w(e) \text{ is hit by its } i\text{-th edge}\}|.$$

If we randomly chose an edge e from E', then the probability that $N(e)$ is greater than 0 must be less than the fraction of violated edges of B. So, we have

$$\Pr[N(e) > 0] \leq \mathrm{UNSAT}_{a'}(B).$$

If we can prove that

$$\beta_2\sqrt{t} \cdot \frac{|F|}{|E|} \leq \Pr[N(e) > 0],$$

for some constant β_2, then the lemma is proven because

$$\mathrm{UNSAT}_{a'} \geq \Pr[N(e) > 0] \geq \beta_2\sqrt{t}\min(\mathrm{UNSAT}_a(A), \frac{1}{t}).$$

To prove this, let us, for now, assume the following claims are true:

1. $\mathrm{E}[N(e)] \geq c_1\sqrt{t}\frac{|F|}{|E|}$, for some constant c_1; and
2. $\mathrm{E}[N(e)^2] \leq c_2\sqrt{t}\frac{|F|}{|E|}$, for some constant c_2.

Here, $\mathrm{E}[N(e)]$ is the expected value of $N(e)$ when randomly choosing $e \in E'$. Now we can use the following lemma, which is a result in probability theory and we do not give a proof here.

Lemma 5.51 For any non-negative random variable X, with $\Pr[X = 0] \neq 1$, we have

$$\Pr[X > 0] \geq \frac{\mathrm{E}[X]^2}{\mathrm{E}[X^2]}.$$

□

If we apply this to our claims, we get

$$\Pr[N(e) > 0] \geq \frac{\mathrm{E}[N(e)]^2}{\mathrm{E}[N(e)^2]} \geq \frac{c_1^2}{c_2} \cdot \sqrt{t} \cdot \frac{|F|}{|E|}.$$

So, when we take $\beta_2 = \frac{c_1^2}{c_2}$, the lemma is proven.

Let us prove these claims.

Proof of claim 1 Let $N_i(e)$ be a function such that, for $i \in I$, we have $N_i(e) = 1$ if the walk associated with e is hit by its i-th edge, and $N_i(e) = 0$ otherwise. Then, clearly

$$N(e) = \sum_{i \in I} N_i(e).$$

And, by the linearity of the expected value,

$$\mathrm{E}[N(e)] = \sum_{i \in I} \mathrm{E}[N_i(e)].$$

Consider the outcome of $\mathrm{E}[N_i(e)]$, which equals the fraction of edges e in E' of which the associated walk $w(e)$ is hit by its i-th edge. In other words, this is the probability that, when choosing a random edge $e \in E'$, the associated walk $w(e)$ is hit by its i-th edge. Thus,

$$\mathrm{E}[N_i(e)] = \Pr[N_i(e) = 1].$$

As we know, choosing a random edge in E' is equivalent to making a random t-step walk in the original graph, G, starting in a random node. A way to obtain these random t-step walks is the following, given some $i \in I$:

1. Choose an edge $\{u, v\} \in E$ uniformly at random from the original graph.
2. Choose a random walk of $i - 1$ steps starting in node u, $(u = v_{i-1}, v_{i-2}, \dots v_0)$.
3. Choose a random walk of $t - i$ steps starting in node v, $(v = v_i, v_{i+1}, \dots v_t)$.
4. Output the walk $(v_0, v_1, \dots v_t)$.

Clearly, the walks obtained this way are distributed uniformly over all t-step walks. Now, the edge produced in step 1, $\{u, v\}$ is the i-th edge of the walk. So, by definition, the walk is hit by its i-th edge if $\{u, v\} \in F$, $a'(v_0)_u = a(u)$ and $a'(v_t)_v = a(v)$. The probability that $\{u, v\}$ is in F equals $\frac{|F|}{|E|}$. So we can write $\Pr[N_i(e)]$ as

$$\Pr[N_i(e)] = \frac{|F|}{|E|} \cdot p_u \cdot p_v,$$

where p_u is the probability that $a'(v_0)_u = a(u)$, and likewise, p_v is the probability that $a'(v_t)_v = a(v)$. Consider the random variable $X_{u,l}$ such that $\Pr[X_{u,l} = \sigma]$, with $\sigma \in \Sigma$, equals the probability that, when taking a random l-step walk starting in u we reach a node w (that is, the last node in the walk) such that $a'(w)_u = \sigma$. Then we can write

$$p_u = \Pr[X_{u,i-1} = a(u)] \text{ and } p_v = \Pr[X_{v,t-i} = a(v)].$$

It can be proven that there exists a constant c_3 such that for all l, if $|l - \frac{t}{2}| \leq \sqrt{\frac{t}{2}}$, then $\Pr[X_{u,l} = \sigma] \geq c_3 \cdot \Pr[X_{u,t/2} = \sigma]$. We do not give a proof here, but we refer the reader to [9] for a proof. However, the intuitive idea is that random walks of length l are similar to random walks of length $\frac{t}{2}$ as long as l is close to $\frac{t}{2}$ and the graph contains at least one loop on every node, as we assume in this lemma. Note that for every $l \in I$, we have $|l - \frac{t}{2}| \leq \sqrt{\frac{t}{2}}$.

Applying this to p_u and p_v, we get:

$$\begin{aligned} \Pr[N_i(e) = 1] &= \frac{|F|}{|E|} \cdot p_u \cdot p_v \\ &\geq \frac{|F|}{|E|} \cdot c_3 \Pr[X_{u,t/2} = a(u)] \cdot c_3 \Pr[X_{v,t/2} = a(v)] \end{aligned}$$

Remember that a is the popular opinion assignment. This means that, for every node u, $a(u)$ is chosen to maximise the number of neighbours (reachable within $\frac{t}{2}$ steps) of u with the opinion $a(u)$ on u. In other words, $a(u)$ is chosen such that $\Pr[X_{u,t/2} = a(u)$ is maximal. This implies that

$$\Pr[X_{u,t/2} = a(u)] \geq \frac{1}{|\Sigma|}.$$

Using this, we get

$$\begin{aligned}\Pr[N_i(e) = 1] &\geq \frac{|F|}{|E|} \cdot c_3 \Pr[X_{u,t/2} = a(u)] \cdot c_3 \Pr[X_{v,t/2} = a(v)] \\ &\geq \frac{|F|}{|E|} \cdot \left(\frac{c_3}{|\Sigma|}\right)^2\end{aligned}$$

Finally, we get

$$\begin{aligned}\mathrm{E}[N(e)] &= \sum_{i\in I} \mathrm{E}[N_i(e)] \\ &= \sum_{i\in I} \Pr[N_i(e) = 1] \\ &\geq \sum_{i\in I} \frac{|F|}{|E|} \cdot \left(\frac{c_3}{|\Sigma|}\right)^2 \\ &\geq |I| \cdot \frac{|F|}{|E|} \cdot \left(\frac{c_3}{|\Sigma|}\right)^2 \\ &= \sqrt{2} \cdot \sqrt{t} \cdot \left(\frac{c_3}{|\Sigma|}\right)^2 \cdot \frac{|F|}{|E|}\end{aligned}$$

Note that the last equality holds because $|I| = 2\sqrt{\frac{t}{2}} = \sqrt{2}\sqrt{t}$. Thus we have proven claim 1 with

$$c_1 = \sqrt{2} \cdot \left(\frac{c3}{|\Sigma|}\right)^2.$$

Proof of claim 2 Let $Z(e)$ be the number of times that the walk associated with the edge $e \in E'$ intersects with F in the middle part, I. Clearly, $N(e) \leq Z(e)$, for all $e \in E'$, so we have

$$\mathrm{E}[N(e)^2] \leq \mathrm{E}[Z(e)^2].$$

Define $Z_i(e)$ such that, for $i \in I$, $Z_i(e) = 1$ if the i-th edge of the walk associated with e is in F, otherwise $Z_i(e) = 0$. Then,

$$Z(e) = \sum_{i\in I} Z_i(e).$$

By the linearity of the expected value, we get

$$\begin{aligned}\mathrm{E}[Z(e)^2] &= \sum_{i\in I} \mathrm{E}[Z_i(e) \cdot Z(e)] \\ &= \sum_{i\in I} \sum_{j\in I} \mathrm{E}[Z_i(e) \cdot Z_j(e)] \\ &= \sum_{i\in I} \mathrm{E}[Z_i(e)] + 2 \cdot \sum_{\substack{i,j\in I \\ j>i}} \mathrm{E}[Z_i(e) Z_j(e)].\end{aligned}$$

In the last equality, the first sum represents the cases where $i = j$, because $Z_i(e) \cdot Z_i(e) = Z_i(e)$. Note that, because $Z_i(e)$ and $Z_j(e)$ can take only values in $\{0, 1\}$, the product $Z_i(e) \cdot Z_j(e)$ can also only take values in $\{0, 1\}$. So we have, for every $i, j \in I$,

$$\mathrm{E}[Z_i(e) Z_j(e)] = \Pr[Z_i(e) Z_j(e) = 1].$$

Also note that, for every $i \in I$, we have $\Pr[Z_i(e) = 1] = \frac{|F|}{|E|}$, so

$$\sum_{i\in I} \mathrm{E}[Z_i(e)] \leq |I| \cdot \frac{|F|}{|E|}.$$

Using the fact that $Z_i(e)Z_j(e) = 1$ if and only if both $Z_i(e)$ and $Z_j(e)$ are 1, we get

$$\Pr[Z_i(e)Z_j(e) = 1] = \Pr[Z_i(e)] \cdot \Pr[Z_j(e) \mid Z_i(e) = 1] = \frac{|F|}{|E|} \cdot \Pr[Z_j(e) = 1 \mid Z_i(e) = 1].$$

Where $\Pr[Z_j(e) \mid Z_i(e) = 1]$ is the conditional probability, that means the probability that $Z_j(e) = 1$ given that $Z_i(e) = 1$.

Now, assume for a moment that $i = 1$ and $j > i$. Then $\Pr[Z_j(e) = 1 \mid Z_1(e) = 1]$ denotes the probability that, starting with a node incident to an edge in F, the j-th edge is in F. We can see this by thinking of the walk associated with the edge e, $w(e) = (v_0, v_1, \dots v_t)$, because we are given that $Z_1(e) = 1$, we know that the edge $\{v_0, v_1\}$ is in F. So, we are looking for the probability that, when randomly choosing a t-step walk with the first edge in F, the j-th edge of the walk is in F. When thinking of expander graphs, this should sound familiar. In fact, because we have $\lambda(G) \leq \lambda$, we can apply Theorem 5.35. This gives us

$$\Pr[Z_j(e) = 1 \mid Z_1(e) = 1] \leq \frac{|F|}{|E|} + \left(\frac{\lambda}{d}\right)^{j-2}.$$

We note that the random walk starts in v_1, because the first edge is already given, we need $j-2$ steps to get to the j-th edge. Now, if $j > i > 1$, we simply ignore the first $i-1$ steps and we get a similar result, just for walks of length $j-i-1$ instead of $j-2$. Thus:

$$\Pr[Z_j(e) = 1 \mid Z_i(e) = 1] \leq \frac{|F|}{|E|} + \left(\frac{\lambda}{d}\right)^{j-i-1}.$$

So, finally we get

$$\begin{aligned}
\mathrm{E}[Z(e)^2] &= \sum_{i \in I} \mathrm{E}[Z_i(e)] + 2 \cdot \sum_{\substack{i,j \in I \\ j>i}} \mathrm{E}[Z_i(e)Z_j(e)] \\
&\leq |I| \cdot \frac{|F|}{|E|} + 2 \cdot \frac{|F|}{|E|} \cdot \sum_{\substack{i,j \in I \\ j>i}} \left(\frac{|F|}{|E|} + \left(\frac{\lambda}{d}\right)^{j-i-1} \right) \\
&\leq \sqrt{2} \cdot \frac{|F|}{|E|} + 2 \cdot \left(\frac{|F|}{|E|}\right)^2 \cdot |I|^2 + \frac{|F|}{|E|} \cdot \sum_{\substack{i,j \in I \\ j>i}} \left(\frac{\lambda}{d}\right)^{j-i-1} \\
&\leq \sqrt{2} \cdot \frac{|F|}{|E|} + 2 \left(|I| \cdot \frac{|F|}{|E|} \right)^2 + \frac{|F|}{|E|} \cdot \sum_{\substack{i,j \in I \\ j>i}} \left(\frac{\lambda}{d}\right)^{j-i-1} \\
&\leq \sqrt{2} \cdot \frac{|F|}{|E|} + 2 \left(|I| \cdot \frac{|F|}{|E|} \right)^2 + \frac{|F|}{|E|} \cdot |I| \cdot \sum_{i=1}^{\sqrt{2t}} \left(\frac{\lambda}{d}\right)^{i}.
\end{aligned}$$

We note that

$$|I| \cdot \frac{|F|}{|E|} \leq \sqrt{2t} \cdot \frac{1}{t} = \sqrt{2} \cdot \frac{\sqrt{t}}{t} = \frac{\sqrt{2}}{\sqrt{t}} \leq \sqrt{2}.$$

Looking at the summation in the last inequality:

$$\sum_{i=1}^{\sqrt{2t}} \left(\frac{\lambda}{d}\right)^{i} \leq \sum_{i=1}^{\infty} \left(\frac{\lambda}{d}\right)^{i} = \frac{1}{1 - \frac{\lambda}{d}}.$$

Filling this in gives us

$$
\begin{aligned}
\mathrm{E}[Z(e)^2] &\leq \sqrt{2} \cdot \frac{|F|}{|E|} + 2\sqrt{2} \cdot |I| \cdot \frac{|F|}{|E|} + \frac{|F|}{|E|} \cdot |I| \cdot \frac{1}{1 - \frac{\lambda}{d}} \\
&= \left(\sqrt{2} + 2\sqrt{2} \cdot |I| + |I| \cdot \frac{1}{1 - \frac{\lambda}{d}} \right) \cdot \frac{|F|}{|E|} \\
&= \left(\sqrt{2} + 4\sqrt{t} + \sqrt{2}\sqrt{t} \cdot \frac{1}{1 - \frac{\lambda}{d}} \right) \cdot \frac{|F|}{|E|} \\
&\leq \left(\sqrt{2} + 4 + \frac{\sqrt{2}}{1 - \frac{\lambda}{d}} \right) \cdot \sqrt{t} \cdot \frac{|F|}{|E|}
\end{aligned}
$$

This proves claim 2 with

$$c_2 = \sqrt{2} + 4 + \frac{\sqrt{2}}{1 - \frac{\lambda}{d}}.$$

□

Conclusion

During the writing of this thesis, I have learnt a lot and gained many insights into these more advanced topics in the field of computational complexity. Originally, the goal was to mainly focus on the PCP theorem, its proof and its consequences. However, in the process I realized that treating the full proof of the PCP theorem was a bit too ambitious. To be specific, the *alphabet reduction* step of the proof was emitted because it requires results in Fourier analysis. Therefore, the scope of the thesis changed, becoming broader, to include topics like randomized computation and interactive proof systems, while still covering a significant part of the proof of the PCP theorem. I believe it was very insightful to study some of the alternative (non-trivial) characterizations of existing complexity classes, like PSPACE = IP, NP = PCP($\log n, 1$), ZPP = RP $\cap$ coRP, $\ldots$, especially when writing out the proof myself, for some of these. I also learned about some topics that I was not yet familiar with, like number theory, where I used Fermat's little theorem to explain probabilistic primality tests. Or expander graphs, which are used in the proof of the PCP theorem, and have many applications in theoretical computer science and other fields (see [14] for a survey of the applications of expander graphs).

Appendix A

Nederlandstalige samenvatting

A.1 Inleiding

Al van voor het bestaan van computers zoals we ze nu kennen, kende men problemen die niet opgelost kunnen worden door een computer. Zo bedacht Alan Turing in 1936 een formeel model om een algoritme te beschrijven, vandaag gekend als de Turing machine, en bewees daarbij dat het halting probleem niet opgelost kan worden door een Turing machine. Er werd niet alleen onderzocht of problemen wel of niet oplosbaar zijn door algoritmes, maar ook hoe efficiënt een algoritme een probleem kan oplossen, dat wil zeggen, hoeveel tijd en/of geheugen er nodig is. Zo kwam de complexiteitstheorie tot stand, waarin men de *computationele complexiteit* van problemen onderzoekt. Een van de belangrijkste vraagstukken binnen de complexiteitstheorie, en misschien wel binnen de informatica in zijn geheel, is het fameuze P vs. NP probleem. P en NP zijn beide *complexiteitsklassen*, die een manier bieden om problemen te classificeren volgens hun complexiteit.

In deze thesis bestuderen we enkele geavanceerde onderwerpen binnen de complexiteitstheorie. Het eerste onderwerp dat we bekijken is willekeurigheid in algoritmes. Alhoewel computers in principe deterministische machines zijn, is het toch interessant om een model te bekijken waarin algoritmes toegang hebben tot willekeurigheid. Tegenwoordig weten we trouwens niet eens of we, door algoritmes toegang te geven tot willekeurigheid, problemen efficiënter kunnen oplossen of niet, wederom een open vraag binnen de complexiteitstheorie.

Als tweede onderwerp bekijken we interactieve bewijssystemen. Hier vertrekken we vanuit het feit dat de klasse NP kan gedefinieerd worden gebruikmakende van een *bewijssysteem*. Een bewijssysteem bestaat uit twee computers (of algoritmes, Turing machines, ...), een prover en een verifier. Deze twee kunnen met elkaar communiceren door het verzenden van berichten. De prover probeert dan de verifier te overtuigen van een bepaalde bewering, door het verzenden van een bewijs van die bewering. We kunnen dit concept uitbreiden door interactie toe te laten, dat wil zeggen dat de prover en de verifier meerdere berichten naar elkaar mogen verzenden, en tegelijkertijd door de verifier toegang te geven tot willekeurigheid. Zo verkrijgen we een zeer krachtig type van bewijssystemen. Er werd namelijk bewezen, in 1992, dat er zo een bewijssysteem bestaat voor elke taal in PSPACE, dat is de klasse van problemen die opgelost kunnen worden door een algoritme dat hoogstens een polynomiale hoeveelheid geheugen gebruikt. Daarnaast bekijken we nog een bepaald type van bewijssystemen, genaamd *zero-knowledge proof systems*. Dat zijn bewijssystemen waarbij de verifier niets bijleert, buiten het feit dat de gemaakte bewering al dan niet waar is.

Ten slotte bekijken *probabilistically checkable proofs*, afgekort PCP. Dat kan men ook zien als een soort van bewijssysteem, echter is deze niet interactief, de prover zou dan maar een enkel bericht verzenden, namelijk het bewijs zelf. Daarom is het handiger om het te vergelijken met een klassiek bewijs, zoals we dat kennen uit de wiskunde, het verschil daarbij is dan weer het probabilistische aspect van een PCP. Het is zo dat een PCP verifier maar een klein aantal, willekeurig gekozen, symbolen uit het bewijs leest en aan de hand daarvan zal het zijn oordeel maken. Daarbij laten we

toe dat de verifier af en toe een fout oordeel maakt, dat wil zeggen, een bewijs goedkeurt terwijl het foutief is, echter mag dat maar gebeuren met een kleine kans die onafhankelijk is van de invoer. De PCP stelling zegt dat er zo een PCP verifier bestaat voor elk probleem in NP. Om het belang van de PCP stelling te benadrukken, vermelden we dat in 2001, de Gödel prijs werkt uitgereikt voor het origineel bewijs van de PCP stelling, alsook in 2019 voor een eenvoudiger bewijs. We bekijken ook belangrijk gevolg van de PCP stelling, namelijk dat er voor bepaalde optimalisatie problemen er geen poylnomiale-tijd benaderingsalgoritme bestaat, tenzij P = NP.

A.2 Achtergrond

Hier geven we een korte, informele introductie tot de complexiteitstheorie. De formele definities zijn te vinden in Hoofdstuk 1. In de complexiteitstheorie willen we formaliseren hoe moeilijk (dat wil zeggen, hoeveel tijd het kost) het is om een bepaald probleem op te lossen voor een algoritme. Dan hebben we natuurlijk een formeel model nodig om algoritmes te beschrijven, hiervoor maken we gebruik van *Turing machines*. Conceptueel kan zo een Turing machine gezien worden als een apparaat dat als invoer een *string* krijgt, daar een aantal berekeningen op doet, en uiteindelijk kan kiezen om deze invoer te accepteren of af te wijzen. Een string is een reeks van symbolen uit een bepaald alfabet. We zeggen dat de *taal* die een Turing machine herkend gelijk is aan de verzameling van alle strings die de Turing machine accepteert. De manier waarop Turing machines gedefinieerd zijn sluit niet uit dat een Turing machine oneindig lang zou blijven rekenen, en dus nooit de invoer zal accepteren of afwijzen. Echter zullen we in de complexiteitstheorie enkel werken met Turing machines die altijd stoppen na een eindig altijd stappen, ongeacht de invoer. Zo een Turing machine word ook wel een *beslisser* genoemd.

Het aantal stappen die een Turing machine nodig heeft in zijn berekening gegeven een bepaalde invoer, noemen we de uitvoertijd van die machine voor die bepaalde invoer. In de complexiteitstheorie zijn we vooral geïnteresseerd in het asymptotisch gedrag van de uitvoertijd in functie van de lengte van de invoer. We zeggen dat een Turing machine M als tijdscomplexiteit een functie $f : \mathbb{N} \to \mathbb{R}$ heeft als voor elke invoer x, met $|x|$ de lengte van x, de uitvoertijd van M op x hoogstens $f(|x|)$ bedraagt. Om het asymptotisch gedrag te bekijken, maken we gebruik van de grote-O notatie, beschreven in Hoofdstuk 1. Dan zeggen we dat een Turing machine M als tijdscomplexiteit $O(f)$ heeft, met $f : \mathbb{N} \to \mathbb{R}$ een functie, als er een functie $g : \mathbb{N} \to \mathbb{R}$ bestaat zodanig dat $g = O(f)$ en M tijdscomplexiteit g heeft. Als volgende zeggen we dat een een Turing machine M een polynomiale tijdscomplexiteit heeft als er een veelterm (of polynoom) p bestaat zodanig dat M tijdscomplexiteit $O(p)$ heeft.

Nu zijn we klaar om de complexiteitsklasse P te definiëren. Een complexiteitsklasse is eenvoudigweg een verzameling van talen. De complexiteitsklasse P bestaat uit alle talen die beslist worden door een Turing machine met polynomiale tijdscomplexiteit. De klasse P wordt wel eens aanzien als de klasse van talen die efficiënt te beslissen zijn.

Een andere klasse die we definiëren is NP. Een taal L behoort tot NP als lidmaatschap van L in polynomiale tijd geverifieerd kan worden. Dat betekent dat er een Turing machine M met polynomiale tijdscomplexiteit bestaat zodanig dat een invoer x tot de taal L behoort als en slechts als er een string c bestaat zodanig dat M de string x, c accepteert. De string c wordt het certificaat genoemd en kan gezien worden als een bewijs voor het lidmaatschap van x in L. Er bestaat een alternatieve, maar equivalente, definitie voor de klasse NP, waar gebruik gemaakt wordt van niet-deterministische Turing machines. Vandaar komt ook de naam van de klasse NP, dat is een afkorting voor *non-deterministic polynomial time*.

Het al dan niet gelijk zijn van de klassen P en NP is een bekende open vraag. Er zijn bepaalde talen in NP met een opmerkelijke eigenschap, namelijk dat als die taal beslist kan worden in polynomiale tijd, dan kunnen alle talen in NP in polynomiale tijd beslist worden, en dat zou betekenen dat P gelijk is aan NP. Zo een taal wordt NP-compleet genoemd, voor een formele definitie van NP-compleetheid verwijzen we opnieuw naar Hoofdstuk 1.

A.3 Willekeurigheid in algoritmes

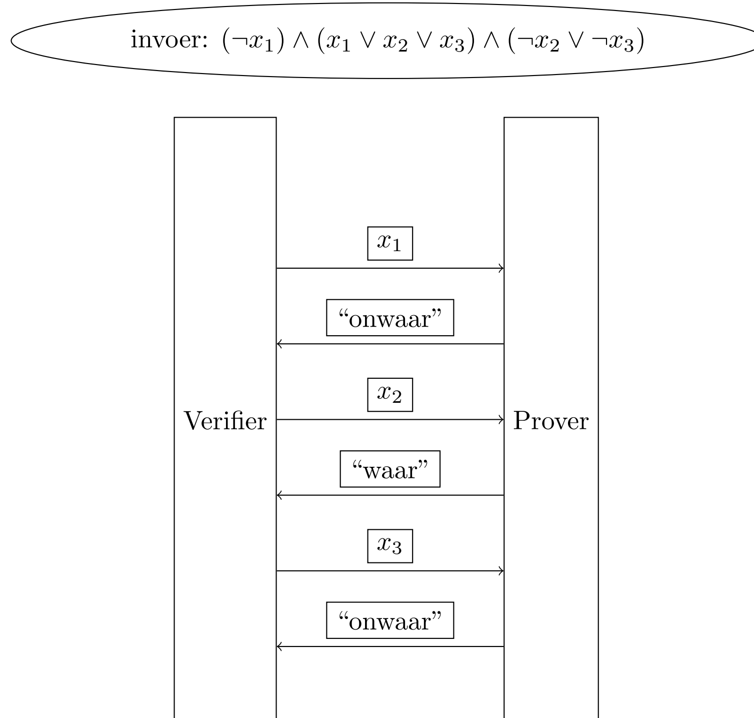
Deze sectie is een samenvatting van Hoofdstuk 2. Op het eerste zicht lijkt het niet erg nuttig om algoritmes willekeurige beslissingen te laten maken, aangezien, in de realiteit, computers enkel deterministische berekeningen kunnen uitvoeren. Toch worden veel computerprogramma's geschreven alsof ze toegang hebben tot willekeurigheid, dan wordt er gebruik gemaakt van zogenaamde *pseudo-random generators*. Daarom is het interessant om te kijken wat er gebeurt wanneer we in ons formeel model algoritmes toegang geven tot willekeurigheid. Om dit te doen, gebruiken we een aangepaste versie van de Turing machine, genaamd de probabilistische Turing machine. Een probabilistische Turing machine kan, op eender welk punt in de berekening, een conceptuele munt opgooien, en aan de hand van de uitkomst daarvan, zijn volgende stap bepalen. Een alternatieve, maar equivalente, formulering is dat een probabilistische Turing machine een willekeurige bit (0 of 1) kan genereren, en aan de hand daarvan zijn volgende stap bepaalt. Nog verschillend van een Turing machine, zullen we bij een probabilistische Turing machine een kans toekennen aan de gebeurtenis dat de machine een gegeven invoer accepteert. Zo kan bijvoorbeeld een probabilistische Turing machine een bepaalde invoer met 75% kans accepteren en met 25% kans afwijzen.

Gebruikmakende van probabilistische Turing machines, kunnen we nieuwe klassen van talen definiëren. De eerste die we hiervan bekijken, is de klasse RP, dat staat voor *randomized polynomial time*. In de definitie van de klasse RP laten we toe dat de probabilistische Turing machine af en toe een *eenzijdige fout* maakt, dat wil zeggen dat de machine een invoer die in de taal behoort toch, met kleine kans, mag afwijzen. Langs de andere kant moet de machine een invoer die niet tot de taal behoort altijd afwijzen. Daarnaast definiëren we ook de klasse coRP, waarbij een probabilistische Turing machine, opnieuw met kleine kans, een fout mag maken op een invoer die niet tot de taal behoort.

Een klassiek voorbeeld van een probabilistisch algoritme is het testen van priemgetallen. Een natuurlijk getal is een priemgetal als het geen product is van twee kleinere natuurlijke getallen. In de context van complexiteitstheorie, kijken we naar de taal PRIMES, die bestaat uit de voorstellingen van alle priemgetallen. In Sectie 2.3 beschrijven we het Miller-Rabin algoritme, een probabilistische test voor priemgetallen. De Miller-Rabin test zal een priemgetal altijd accepteren, en zal samengestelde getallen met grote kans afwijzen. Daardoor weten dat de taal PRIMES tot de klasse coRP behoort. Een belangrijk detail hierbij is dat, alhoewel de Miller-Rabin test een samengesteld getal foutief zal accepteren met kans hoogstens 25%, de kans op een foute uitkomst zo klein gemaakt kan worden als men maar wil (maar niet 0), door de test herhaaldelijk uit te voeren. Het is zelfs zo dat dit voor elke taal in RP en coRP mogelijk is. Nog een merkwaardigheid is dat het lang een open vraag was of priemgetallen al dan niet getest kunnen worden in (deterministische) polynomiale tijd (en dus of PRIMES deel uitmaakt van P). Deze vraag werd in 2002 positief beantwoord door Agrawal, Kayal en Saxena door het uitvinden van de, naar hen vernoemde, AKS test. Nu kan men zich afvragen of er dan wel talen bestaan in RP of coRP die niet in de klasse P behoren. Het antwoord daarop is nog niet gekend, dit is dan ook één van de vele open vragen binnen de complexiteitstheorie.

Er zijn nog complexiteitsklassen die men kan definiëren met probabilistische Turing machines. Zo is er de klasse BPP, een afkorting voor *bounded-error probabilistic polynomial-time*. Het verschil met RP en coRP is dat we bij deze een *tweezijdige fout* toelaten. Dat wil zeggen dat een probabilistische Turing machine een invoer die tot de taal behoort met kleine kans mag afwijzen, alsook een invoer die niet tot de taal behoort met kleine kans mag accepteren. Het is makkelijk om in te zien dat beide RP en coRP een deelverzameling zijn van BPP. Ook bij BPP kan de kans op een fout zo klein gemaakt worden als men maar wil.

Ten slotte hebben we het ook nog over de klasse ZPP, een afkorting voor *zerro-error probabilistic polynomial time*. Zoals uit de naam valt af te leiden, laten we het hier inderdaad niet toe om een fout te maken. Wat we wel toelaten, is dat de uitvoertijd, afhankelijk van uitkomst van de opgegooide munten, meer dan polynomial mag zijn. Langs de andere kant moet, gegeven een invoer, de gemiddelde uitvoertijd wel hoogstens een veelterm zijn in functie van de lengte van de invoer. Opmerkelijk aan de klasse ZPP is dat ze gelijk is aan de intersectie tussen RP en coRP, een bewijs hiervan is te vinden in Sectie 2.5.



Figuur A.1: Interactief bewijssysteem voor de taal 3SAT.

A.4 Interactieve bewijssystemen

Deze sectie is een samenvatting van Hoofdstuk 3 en Hoofdstuk 4. Hier kijken we naar de kracht van *bewijssystemen*. Een bewijssysteem bestaat uit twee Turing machines, een *verifier* en een *prover*. Deze twee kunnen berichten naar elkaar toe sturen. Op die manier zal de prover proberen de verifier te overtuigen van een bepaalde bewering. Die beweringen zullen altijd van dezelfde soort zijn, namelijk dat een bepaalde string een element is van een bepaalde taal. Een voorbeeld van zo een bewering is: een string x is element van de taal PRIMES, dan is de bewering eigenlijk equivalent met de bewering dat a een priemgetal is, waarbij a het getal is met voorstelling x. Een ander voorbeeld is: een string x is element van de taal 3SAT en dan is de bewering equivalent met de bewering dat ϕ satisfiable is, waarbij ϕ de 3CNF formule met voorstelling x is. Zoals we kunnen zien is het op deze manier mogelijk om allerlei soorten beweringen uit te drukken.

We hebben de klasse NP gedefinieerd gebruikmakende van een certificaat. We kunnen dit echter ook bekijken als een bewijssysteem waarbij de prover het certificaat als bericht verzend naar de verifier. Dus we stellen vast dat talen met een bewijssysteem waarbij enkel de prover een bericht stuurt overeenkomen met talen in NP. De vraag is nu wat er gebeurt als we interactieve bewijssystemen toelaten, dat wil zeggen dat de prover en de verifier meerdere berichten naar elkaar mogen verzenden. Laten we als voorbeeld een interactief bewijssysteem voor de taal 3SAT tonen. Stel we hebben de 3CNF formule

$$\phi = (\neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3).$$

Het bewijssysteem werkt als volgt. De verifer verzend telkens een bericht met de naam van een variabele in ϕ naar de prover. Daarop zal de prover antwoorden met een waarheidstoekenning (waar of onwaar) voor die variabele. Eenmaal de verifier een waarheidstoekenning heeft voor elke variabele in ϕ, zal de verifier controleren of die waarheidstoekenning ϕ waar maakt. Als dat zo is, kan de verifier ϕ accepteren, anders wordt ϕ afgewezen. Een voorbeeld van zo een interactie is geïllustreerd in Figuur A.1. Merk op dat in het voorbeeld de waarheidstoekenning, gegeven door de prover, de formule ϕ waar maakt.

Men kan zich vragen stellen over het nut van een interactief bewijssysteem voor de taal 3SAT,

omdat 3SAT tot NP behoort weten we namelijk dat er een bewijssysteem is voor 3SAT waarbij enkel de prover een bericht naar de verifier verzend. Dat bericht kan bijvoorbeeld bestaan uit een waarheidstoekenning voor alle variabelen.

Het is zo dat het toevoegen van interactie op zich weinig verandert aan de kracht van een bewijs-systeem. Om dit in te zien, moet men weten dat we geen limiet leggen op de tijdscomplexiteit van de prover (we doen dit overigens wel voor de verifier, die moet polynomiale tijdscomplexiteit hebben). Dat betekent dat de prover alle mogelijke berichten van de verifier op voorhand kan berekenen en dus in principe alle antwoorden daarop al verzenden naar de verifier in het eerste bericht.

Het is pas als we de verifier probabilistisch maken, dat het interessanter wordt. Dan kan de prover natuurlijk nog altijd op voorhand alle mogelijke berichten van de verifier berekenen, maar dat zouden er nu exponentieel veel kunnen zijn in functie van de lengte van de input, en dan zou de verifier niet alle mogelijke antwoorden kunnen lezen in polynomiale tijd. Nu kunnen we de klasse IP definiëren, die bestaat uit alle talen met een interactief bewijssysteem met probabilistische verifier. In 1992 toonde Shamir aan dat IP = PSPACE [17], een bewijs hiervan is gegeven in Sectie 3.3. Dit resultaat toont de kracht van interactieve bewijssystemen aan. Hierbij moeten we wel opmerken dat het niet geweten of NP al dan niet gelijk is aan PSPACE. We kunnen echter nog een uitbreiding van interactieve bewijssystemen, namelijk het toelaten van meerdere provers (die niet met elkaar mogen communiceren). De klasse die daaruit voortkomt is MIP en daarvan is bewezen dat ze gelijk is aan NEXP, een klasse waarvan NP een strikte deelverzameling is. Dus interactieve bewijssystemen met meerdere provers zijn bewijsbaar krachtiger dan het type bewijssysteem gebruikt in de definitie van NP.

Een speciaal type van bewijssystemen zijn de zogenaamde *zero-knowledge proof systems*. Dat zijn bewijssystemen waarbij de verifier, intuïtief gezien, niets geen informatie verkrijgt over de invoer buiten het feit dat de bewering al dan niet waar is. Dit wordt geformaliseerd door ervoor te zorgen dat alle berichten die de verifier ontvangt, ook door de verifier zelf gegenereerd kunnen worden. Een belangrijke stelling hieromtrent werd in 1991 bewezen door Goldreich, Micali en Widgerson [11], die zegt dat alle talen in NP een zero-knowledge proof system hebben, onder de veronderstelling dat *one-way functies* bestaan.

A.5 De PCP stelling

Laten we terugkijken naar de definitie van NP. Daar zeiden we dat een taal een element is van NP als deze in polynomiale tijd geverifieerd kan worden, met behulp van een certificaat. In deze sectie maken we een aantal aanpassingen aan die definitie om zo een nieuwe complexiteitsklasse te verkrijgen. Om te beginnen laten we de verifier probabilistisch zijn, hierbij laten we ook een eenzijdige fout toe, waardoor een invoer die niet tot de taal behoort, met een kleine kans toch geaccepteerd mag worden, net zoals bij coRP. Ten tweede zetten we een limiet op het aantal munten dat de verifier mag opgooien en op het aantal symbolen dat hij mag lezen uit het certificaat. De resulterende verifier noemen we een PCP verifier, waarbij PCP staat voor *probabilistically checkable proof.* Er is aangetoond dat elke taal in NP een PCP verifier heeft die een logaritmisch aantal munten opgooit en een constant aantal symbolen bekijkt van het certificaat. Dit resultaat staat bekend als de PCP stelling, we geven hiervan een deel van het bewijs, in Sectie 5.3.

Een belangrijk gevolg van de PCP stelling is dat de benadering van bepaalde optimalisatie proble-men een eigenschap hebben die vergelijkbaar is met het concept van NP-compleetheid. Dit betekent dat als er een algoritme met polynomiale tijdscomplexiteit bestaat voor één van die problemen, dan is P gelijk aan NP. Als voorbeeld van een optimalisatie probleem nemen we max-3SAT, waarbij het de bedoeling is om een waarheidstoekenning te vinden die het aantal satisfied clauses in een gegeven 3CNF formule maximaliseert. We zeggen dat een benaderingsalgoritme voor max-3SAT c-optimaal is als het altijd een waarheidstoekenning vind die minstens c maal het maximum aantal clauses satsfied. Zo bestaat er een eenvoudig benaderingsalgoritme voor max-3SAT dat $\frac{1}{2}$-optimaal is. Een gevolg van de PCP stelling is echter dat er een constante c bestaat zodanig dat er geen c-optimaal benaderingsalgoritme bestaat voor max-3SAT met polynomiale tijdscomplexiteit, tenzij

P gelijk is aan NP.

Bibliography

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160, 09 2002.

[2] W. R. Alford, Andrew Granville, and Carl Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703–722, 1994.

[3] Noga Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[4] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach.* Cambridge Univ. Press, 2010.

[5] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, May 1998.

[6] L Babai. Trading group theory for randomness. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 421–429, New York, NY, USA, 1985. Association for Computing Machinery.

[7] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Comput. Complex.*, 2(4):374, December 1992.

[8] Stephen A. Cook. A hierarchy for nondeterministic time complexity. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, page 187–192, New York, NY, USA, 1972. Association for Computing Machinery.

[9] Irit Dinur. The PCP theorem by gap amplification. *J. ACM*, 54(3):12–es, June 2007.

[10] Oded Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, 2008.

[11] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[12] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[13] S Goldwasser and M Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 59–68, New York, NY, USA, 1986. Association for Computing Machinery.

[14] Shlomo Hoory, Nathan Linial, and Avi Widgerson. Expander graphs and their application. *Bulletin (New Series) of the American Mathematical Society*, 43, 10 2006.

[15] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[16] O. Reingold, S. Vadhan, and A. Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 3–13, 2000.

[17] Adi Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, October 1992.

[18] A. Shen. IP = SPACE: Simplified proof. *J. ACM*, 39(4):878–880, October 1992.

[19] Michael Sipser. *Introduction to the theory of computation.* Course Technology Cengage Learning, 3rd edition, 2013.