UNIVERSITY OF LJUBLJANA
FACULTY OF MATHEMATICS AND PHYSICS
DEPARTMENT OF MATHEMATICS

David Gajser

Verifying Time Complexity of Turing Machines

Doctoral dissertation

Advisor: izred. prof. dr. Sergio Cabello Justo
Co-advisor: prof. dr. Bojan Mohar

Ljubljana, 2015

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
ODDELEK ZA MATEMATIKO

David Gajser

Preverjanje časovne zahtevnosti Turingovih strojev

Doktorska disertacija

Mentor: izred. prof. dr. Sergio Cabello Justo
Somentor: prof. dr. Bojan Mohar

Ljubljana, 2015

Izjava

Podpisani David Gajser izjavljam:

- da sem doktorsko disertacijo z naslovom *Preverjanje časovne zahtevnosti Turingovih strojev* izdelal samostojno pod mentorstvom izred. prof. dr. Sergia Cabella Justa in somentorstvom prof. dr. Bojana Moharja.
- da Fakulteti za matematiko in fiziko Univerze v Ljubljani dovoljujem objavo elektronske oblike svojega dela na spletnih straneh.

Ljubljana, 13. 10. 2015 Podpis

Acknowledgements

I met my advisor Sergio Cabello just before I started writing my bachelor thesis. Since then, he guided me carefully, offering numerous advices. I chose not to do research in his main area of expertise which is computational geometry. Instead, I analyzed Turing machines in detail. However, we had many enlightening consultations about topics of my, his and common interest and about the mathematical research in general. Sergio, I would like to thank you for all the guidance and support.

I would like to thank my co-advisor Bojan Mohar and Valentine Kabanets for co-hosting me for a semester on Simon Fraser University, Canada. I met a big research group of Bojan there and I saw how productive such a group can be. With Valentine, I was able to discuss the state of art in the area that interests me most, that is computational complexity theory. I would also like to thank him for aikido lessons.

I would like to thank numerous people for reading and commenting my work. First on the list is my advisor, then Valentine Kabanets, Bojan Mohar, the anonymous reviewer of the paper [10], Andrej Bauer, Marko Petkovšek, Matjaž Konvalinka and Jurij Mihelič.

I would like to thank Jana, Gašper, my family, and all of my friends for taking and making me as I am.

Abstract

The central problem in the dissertation is the following.

For a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, how hard is it to verify whether a given Turing machine runs in time at most $T(n)$? Is it even possible?

Our first main contibution is that, for all reasonable functions $T(n) = \mathrm{o}(n \log n)$, it is possible to verify with an algorithm whether a given one-tape Turing machine runs in time at most $T(n)$. This is a tight bound on the order of growth for the function T because we prove that, for $T(n) = \Omega(n \log n)$ and $T(n) \geq n+1$, there exists no algorithm that would verify whether a given one-tape Turing machine runs in time at most $T(n)$. As opposed to one-tape Turing machines, we show that we can verify with an algorithm whether a given multi-tape Turing machine runs in time at most $T(n)$ if and only if $T(n_0) < n_0 + 1$ for some $n_0 \in \mathbb{N}$.

Linear time bounds are the most natural algorithmically verifiable time bounds for one-tape Turing machines, because a one-tape Turing machine that runs in time $\mathrm{o}(n \log n)$ actually runs in linear time. This motivates our second main contibution which is the analysis of complexity of the following family of problems, parameterized by integers $C \geq 2$ and $D \geq 1$:

Does a given one-tape q-state Turing machine run in time $Cn + D$?

Assuming a fixed tape and input alphabet, we show that these problems are co-NP-complete and we provide good lower bounds. Specifically, these problems cannot be solved in $\mathrm{o}(q^{(C-1)/4})$ non-deterministic time by multi-tape Turing machines. We also show that the complements of these problems can be solved in $\mathrm{O}(q^{C+2})$ non-deterministic time and not in $\mathrm{o}(q^{(C-1)/4})$ non-deterministic time by multi-tape Turing machines.

To prove the upper bound $\mathrm{O}(q^{C+2})$, we use the so-called compactness theorem which is our third main contribution. We need more notation to state it in full generality, but a simple corollary tells the following: To verify whether an input one-tape Turing machine runs in time $Cn + D$, it is enough to verify this on a finite number of inputs.

We argue that our main results are proved with techniques that relativize and that using only such techniques we cannot solve the P versus NP problem.

Math. Subj. Class. (2010): 68Q05, 68Q10, 68Q15, 68Q17

Keywords: Turing machine, relativization, NP-completeness, crossing sequence, decidability, lower bound, time complexity, running time, linear time

Povzetek

Osrednji problem v disertaciji je sledeč.

Naj bo $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ poljubna funkcija. Kako težko je preveriti, ali je časovna zahtevnost danega Turingovega stroja $T(n)$? Je to sploh mogoče preveriti?

Naš prvi večji prispevek pove, da je za vse "normalne" funkcije $T(n) = \mathrm{o}(n \log n)$ možno z algoritmom preveriti, ali je časovna zahtevnost danega enotračnega Turingovega stroja $T(n)$. Meja $\mathrm{o}(n \log n)$ je tesna, saj za $T(n) = \Omega(n \log n)$ in $T(n) \geq n + 1$ ni mogoče z algoritmom preveriti, ali je časovna zahtevnost danega enotračnega Turingovega stroja $T(n)$. Pri večtračnih Turingovih strojih je rezultat enostavnejši. Zanje namreč velja, da je časovno zahtevnost $T(n)$ moč z algoritmom preveriti natanko tedaj, ko velja $T(n_0) < n_0 + 1$ za neki $n_0 \in \mathbb{N}$.

Znano je, da je vsak enotračni Turingov stroj časovne zahtevnosti $\mathrm{o}(n \log n)$ tudi linearne časovne zahtevnosti. Posledično je linearna časovna zahtevnost najbolj naravna časovna zahtevnost, ki jo lahko z algoritmom preverimo pri enotračnih Turingovih strojih. V disertaciji se zato ukvarjamo tudi z naslednjimi problemi, ki so parametrizirani z naravnima številoma $C \geq 2$ in $D \geq 1$:

Ali je dani enotračni Turingov stroj s q stanji časovne zahtevnosti $Cn + D$?

Pri analizi teh problemov, kar je naš drugi večji prispevek, predpostavljamo fiksno vhodno in tračno abecedo. Ti problemi so co-NP-polni in zanje lahko dokažemo dobre spodnje meje računske zahtevnosti. Ni jih namreč mogoče rešiti v času $\mathrm{o}(q^{(C-1)/4})$ z nedeterminističnimi večtračnimi Turingovimi stroji. Še več, komplementi teh problemov so rešljivi z večtračnimi nedeterminističnimi Turingovimi stroji v času $\mathrm{O}(q^{C+2})$, ne pa v času $\mathrm{o}(q^{(C-1)/4})$.

Pri dokazu zgornje meje $\mathrm{O}(q^{C+2})$ uporabimo tako imenovani izrek o kompaktnosti, naš tretji večji prispevek. Potrebovali bi več notacije, da bi ga na tem mestu navedli, zato povejmo le njegovo posledico: Da bi preverili, ali dani enotračni Turingov stroj teče v času $Cn + D$, je dovolj preveriti čas izvajanja Turingovega stroja le na končno mnogo vhodih.

Glavni prispevki te disertacije so dokazani s tehnikami, ki relativizirajo. Dokažemo tudi znano dejstvo, da s takimi tehnikami ni mogoče rešiti slavnega problema $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

Daljši povzetek v slovenskem jeziku najdemo na koncu disertacije.

Ključne besede: Turingov stroj, relativizacija, NP-polnost, prekrižno zaporedje, odločljivost, spodnja meja, časovna zahtevnost, čas izvajanja, linearni čas

Contents

Chapter 1

Introduction

The introduction is split into three parts. The first part is for the general public; it gives the motivation to study the problems presented in the dissertation and it explains how the presented concepts reflect (in) the real world. The second part is for readers that are familiar with undergraduate computational complexity theory; we skim through the chapters in this part. The third part is for experts that are interested also in related work.

1.1 Motivation

Since the invention of modern day computers, the following definition of complexity has been very natural: A problem is hard if a computer cannot solve it fast. This empirical definition can be put on solid ground with a well defined model of computation and the history shows that the Turing machines are a very reasonable one. Thus we can say that a problem is hard if no Turing machine can solve it fast. This reasoning is supported by two theses (see Section 3.5). The first one is the *Church-Turing thesis* which states that *intuitively computable functions* are exactly those computable by Turing machines. The thesis is not a mathematical statement, so it cannot be formally proven, despite some attempts [5]. A good argument in favor of the thesis is the fact that many *realizable* models of computation, also the models of personal computers, can be simulated by Turing machines. What is more, the simulations are efficient in the sense that the simulating Turing machine does not make essentially more steps than the simulated model of computation. This ascertainment forms the basis for the *strong version of the Church-Turing thesis* which states that all reasonable models of computation are polynomially equivalent to the Turing machines, i.e., they are comparably fast. The strong version is not so generally accepted as the Church-Turing thesis and quantum computers presumably violate it. However, the current technology still does not allow us to build reasonably big quantum computers.

While Turing machines can compute everything that our computers can compute, and they can compute it (theoretically) comparably fast, the biggest advantage of Turing machines over today's computers is their simplicity. To present how simple they are, let us sketchily describe a one-tape deterministic Turing machine M (abbreviated as one-tape DTM); the formal definition can be found in Section 3.1.6. M physically consists of three things: a two-way infinite *tape*, a *head* and a *state control*. The tape is divided into tape *cells*, where each cell contains exactly one *symbol* and the head of M is always above some tape cell. In the state control, which is connected to the head, there are finitely many *states* in which M can be, one being the *starting state* and some of them being the *halting states*. The computation begins with the input written on the tape (each symbol

of the input in its own tape cell), the rest of the tape cells are filled with so-called *blank symbols* $\sqcup$ (see Figure 1.1), the head is over the first symbol of the input and the machine is in the starting state. Then, in each step, the machine reads the symbol below its head which together with the current state completely determines an action of the following type: rewrite the symbol below the head with a prescribed symbol, change the current state and move the head for one cell to the left or to the right. Hence, M computes very locally since the action in each step is determined only by the current state and the symbol below its head. M finishes its computation when it enters a halting state, although this may never happen in which case M runs forever. The result of the computation can be either the content of the tape after the computation or the halting state in which M finished its computation. As we discussed above, despite the simplicity of the model, one-tape DTMs can efficiently simulate computations carried out by modern day computers.

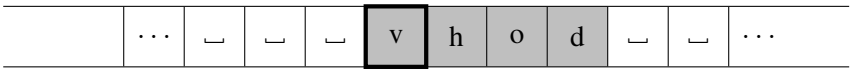


Figure 1.1: The tape of a one-tape DTM with input *vhod* written on it. Before the first step of the computation, the DTM is in the starting state and its head is above the symbol *v*.

In this dissertation we mostly consider decision problems, i.e., the problems which have a *yes* or *no* answer. While the notion of a decision problem is more formally defined in Section 2.1.4, we present here only a few examples.

COMPARE LENGTH	...	Given a string of symbols w, is w of the form $00\ldots011\ldots1$ where the number of 0s equals the number of 1s?
HAMILTONIAN CYCLE	...	Given a simple undirected graph G, is G Hamiltonian[1]?
D-HALT$^1_\varepsilon$	...	Given a one-tape DTM M, does M halt on the empty input, i.e., does M halt on the input with no symbols?

The hardest decision problems are those that cannot be solved by Turing machines and it is well known that the problem D-HALT$^1_\varepsilon$ is an example of such a problem (see Section 4.1.1 for a proof). While the fact that such a simple problem cannot be solved by Turing machines is interesting by itself, it also has a real-world application. It tells us that there is no computer program that could solve the problem

> Given a code of a program in Java that does not need an input, would the program ever terminate if we would run it, or would it run forever?

Hence, verifying correctness of the code is a job that cannot be completely automated.

It is natural to classify decision problems with respect to how fast they can be solved by Turing machines. If we do so, we get a whole hierarchy of distinct complexity classes (see Section 4.2.2). The most well known class is the class P of decision problems that can be solved in polynomial time by one-tape DTMs. In other words, a problem is in P if and only if there exists a polynomial p and a one-tape DTM that solves the problem and, for all n, makes at most $p(n)$ steps on inputs of length n.

Another well known class of decision problems is NP. It is defined as the class of decision problems whose *yes*-answers can be verified in polynomial time by a one-tape DTM that is always

[1]For the definition of a simple and a Hamiltonian graph see Section 2.1.4.

given a so-called certificate (a short hint) together with the input. The class is formally defined in Section 3.4; here we give just an example. The problem HAMILTONIAN CYCLE is in NP because, for each graph that has a Hamiltonian cycle, we can give a sequence of vertices that form the cycle as a certificate. Given such a certificate, we can verify in polynomial time whether the certificate is indeed a Hamiltonian cycle. However, no short certificates are known for the complement of the problem HAMILTONIAN CYCLE:

Given a simple undirected graph G, is it true that G is not Hamiltonian?

This problem is in the class co-NP which includes exactly the complements of the decision problems from NP (the *yes* and *no* answers are switched). There are several natural, real-world problems that are in NP or co-NP but not known to be in P, one of them being HAMILTONIAN CYCLE (see also [12]). While clearly $\mathsf{P} \subseteq \mathsf{NP} \cap \text{co-}\mathsf{NP}$, the question whether $\mathsf{P} = \mathsf{NP}$ is central in computational complexity theory and has spurred the field. It is one of the still unsolved Millennium Prize Problems and its solution is worth a million USD [23]. The question appeared also in the title of a book by Richard J. Lipton [22] and surveys have been written about what theorists think of the P versus NP problem [13]. Two more open problems are whether $\mathsf{NP} = \text{co-}\mathsf{NP}$ and whether $\mathsf{P} = \mathsf{NP} \cap \text{co-}\mathsf{NP}$ and there are many other natural classes of decision problems for which it is not known how they relate to P, to NP or among themselves.

Motivated by such questions and having in mind that many natural classes of decision problems can be rigorously defined by Turing machines, it is of great benefit for a researcher to know and understand very well the model of Turing machines. One of the basic properties of a Turing machine is its running time. The main results of the author during his graduate years talk about how to verify and whether it is even possible to algorithmically verify whether a Turing machine runs in a specified time [10, 11]. These results are presented in Chapter 6.

1.2 Outline of the Dissertation

The objective of the dissertation is to present results from Chapters 5 and 6 together with their background. Results in Chapter 6 talk about verifying time complexity of a given Turing machine. While the results for multi-tape Turing machines are simple, the results for one-tape Turing machines are more involved. A main tool used to analyze one-tape Turing machines are crossing sequences, studied in Chapter 5. Most results from Chapters 5 and 6 are by the author [10, 11] and we present them in more detail below.

Chapters 2, 3 and 4 contain quite standard undergraduate and graduate topics that are preliminary or supplementary to the topics in Chapters 5 and 6. In Chapter 2, basic notation is introduced and regular languages are studied. In Chapter 3, several models of Turing machines are introduced together with time-related complexity classes. A major and very technical section in this chapter, Section 3.3, explains how different attributes of Turing machines influence the time complexity of deciding a language, where the attributes are size of the tape alphabet, number of tapes and the use of non-determinism. In Chapter 4 we prove undecidability of the halting problem, time hierarchy theorems and the famous limitation of relativizing results: the solution of the P versus NP problem does not relativize. The author uses this fact to show that using only the methods from Chapters 5 and 6, we cannot solve famous problems such as P versus NP.

Chapter 6

This is the last chapter and it holds the same title as the dissertation: Verifying Time Complexity of Turing Machines. For this introduction, if not specified otherwise, all results hold for non-deterministic Turing machines (abbreviated as NTMs) as well as deterministic Turing machines (abbreviated as DTMs).

While it is tempting to argue about a Turing machine's time complexity, we cannot algorithmically tell even whether a given Turing machine halts on the empty input (see Section 4.1.1). Can we perhaps check whether it is of a specified time complexity? While the answer is *no* in most cases, there is an interesting case where the answer is *yes*: verifying a time bound $T(n) = Cn + D$, $C, D \in \mathbb{Z}$, for a given one-tape Turing machine.

There are at least two natural types of questions about whether a Turing machine obeys a given time bound:

- For a function $T : \mathbb{N} \to \mathbb{R}_{>0}$, does a given Turing machine run in time $\mathrm{O}(T(n))$?
- For a function $T : \mathbb{N} \to \mathbb{R}_{>0}$, does a given Turing machine run in time $T(n)$, i.e., does it make at most $T(n)$ steps on all computations on inputs of length n for all n?

It is a folklore that it is undecidable whether a Turing machine runs in time $\mathrm{O}(1)$, thus the first question is undecidable for all practical functions T. We state a generalization of this well known fact in Theorem 6.1.6 and prove it using standard techniques. However, for the second question, it is not hard to see that it is decidable whether a given Turing machine runs in time C for some constant $C \in \mathbb{N}$: we just need to simulate a given Turing machine on all the inputs up to the length C (for details, see Lemma 6.1.1). It would be interesting if the second question were decidable also for linear functions T. However, we prove in Theorem 6.1.3 that it is decidable whether a multi-tape Turing machine runs in time $T(n)$ if and only if we have the "eccentric" case $T(n_0) < n_0 + 1$ for some $n_0 \in \mathbb{N}$. The time bound $n + 1$ is special because it minimally enables a multi-tape Turing machine to mark time while simulating another Turing machine. The timekeeping can be done on the input tape by just moving the head to the right until the blank symbol at the end marks $n + 1$ steps, while the other tapes are used for the simulation. But what if the simulation has to be performed on the same tape as the timekeeping, i.e., how much time do we need for a one-tape Turing machine to count steps and simulate another Turing machine? We show in Theorem 6.1.5 that $\Omega(n \log n)$ time is enough:

> *Let $T : \mathbb{N} \to \mathbb{R}_{>0}$ be a function such that $T(n) = \Omega(n \log n)$ and, for all $n \in \mathbb{N}$, it holds $T(n) \geq n + 1$. Then it is undecidable whether a given one-tape Turing machine runs in time $T(n)$.*

Theorem 6.1.10 gives a nice contrast:

> *For any "nice" function $T : \mathbb{N} \to \mathbb{R}_{>0}$, $T(n) = \mathrm{o}(n \log n)$, it is decidable whether a given one-tape Turing machine runs in time $T(n)$.*

Hence, a one-tape Turing machine that runs in time $T(n) = \mathrm{o}(n \log n)$ cannot count steps while simulating another Turing machine. There is another well known fact about one-tape Turing machines that makes the time bounds $\Theta(n \log n)$ special: these bounds are the tightest that allow a one-tape Turing machine to recognize a non-regular language (see Propositions 5.1.7 and 5.1.9).

Corollary 5.1.6 from Chapter 5 tells that one-tape Turing machines that run in time $\mathrm{o}(n \log n)$ actually run in linear time. Thus, we can conclude that the most natural algorithmically verifiable

time bound for one-tape Turing machines is the linear one. This is a motivation for the second half of the last chapter, where we analyze the computational complexity of the following problems parameterized by integers $C, D \in \mathbb{N}$. The problem HALT^1_{Cn+D} is defined as

Given a one-tape NTM, does it run in time $Cn + D$?

and the problem D-HALT^1_{Cn+D} is defined as

Given a one-tape DTM, does it run in time $Cn + D$?

For the analyses of the problems HALT^1_{Cn+D} and D-HALT^1_{Cn+D}, we fix an input alphabet Σ, $|\Sigma| \geq 2$, and a tape alphabet $\Gamma \supset \Sigma$. It follows that the length of most standard encodings of q-state one-tape Turing machines is $\mathrm{O}(q^2)$. To make it simple, we assume that each code of a q-state one-tape Turing machines has length $\Theta(q^2)$ and when we will talk about the complexity of the problems HALT^1_{Cn+D}, we will always use q as the parameter to measure the length of the input (see Section 6.2.1). We prove the following.

Theorem 1.2.1. *For all integers $C \geq 2$ and $D \geq 1$, all of the following holds.*

(i) The problems HALT^1_{Cn+D} *and* D-HALT^1_{Cn+D} *are* co-NP-complete.

(ii) The problems HALT^1_{Cn+D} *and* D-HALT^1_{Cn+D} cannot *be solved in time* $\mathrm{o}(q^{(C-1)/4})$ *by multi-tape NTMs.*

(iii) The complements of the problems HALT^1_{Cn+D} *and* D-HALT^1_{Cn+D} can *be solved in time* $\mathrm{O}(q^{C+2})$ *by multi-tape NTMs.*

(iv) The complement of the problem HALT^1_{Cn+D} cannot *be solved in time* $\mathrm{o}(q^{(C-1)/2})$ *by multi-tape NTMs.*

(v) The complement of the problem D-HALT^1_{Cn+D} cannot *be solved in time* $\mathrm{o}(q^{(C-1)/4})$ *by multi-tape NTMs.*

To put the theorem in short, the problems HALT^1_{Cn+D} and D-HALT^1_{Cn+D} are co-NP-complete with a non-deterministic and a co-non-deterministic time complexity lower bound $\Omega(q^{0.25C-1})$ and a co-non-deterministic time complexity upper bound $\mathrm{O}(q^{C+2})$.

Chapter 5

This chapter contains the definition and the main results about crossing sequences. They are defined only for one-tape Turing machines. Intuitively, a *crossing sequence generated by a one-tape Turing machine M after t steps of a computation ζ on an input w at a boundary i* (see Figure 1.2) is a sequence of states of M in which M crosses the ith boundary of its tape when considering the first t steps of the computation ζ on the input w. We assume that, in each step, M first changes the state and then moves the head. Note that this sequence contains all information that the machine carries across the ith boundary of the tape in the first t steps of the computation ζ.

The main technique to deal with crossing sequences is called the *cut-and-paste* technique. We describe it in Section 5.1.1 and use it to prove the main result in this chapter, the compactness theorem (Theorem 5.2.1). We need more notation to state it in full generality, but a simple corollary is the following.

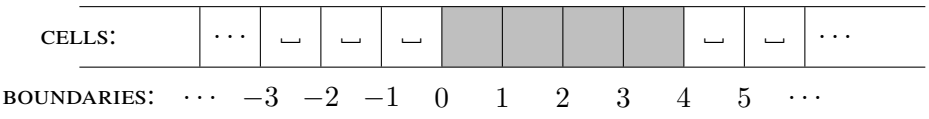


Figure 1.2: Numbering of tape boundaries of a one-tape Turing machine. The shaded part is a potential input of length 4.

Corollary 1.2.2. *For all positive integers C and D, a one-tape q-state Turing machine runs in time $Cn + D$ if and only if, for each $n \leq \mathrm{O}(q^{2C})$, it makes at most $Cn + D$ steps on each computation on inputs of length n.*

To rephrase the corollary, we can solve the problem $\textsc{Halt}^1_{Cn+D}$ for an input Turing machine M by just verifying the running time of M on the inputs of length at most $\mathrm{O}(q^{2C})$. Behind the big O is hidden a polynomial in C and D (see Corollary 5.2.5). The result is interesting not only because it allows us to algorithmically solve the problem $\textsc{Halt}^1_{Cn+D}$, but also because it gives a good insight into one-tape linear-time computations. However, we need the more powerful compactness theorem to prove the upper bound in Theorem 1.2.1.

In Section 5.1.2 we prove a standard result about one-tape Turing machines that run in time $\mathrm{o}(n \log n)$: such Turing machines generate only crossing sequences of size $\mathrm{O}(1)$ and they accept only regular languages. Additionally, we show that they actually run in linear time. In Section 5.2.3 we give an algorithm that takes integers $C, D \in \mathbb{N}$ and a one-tape NTM M as inputs and if M runs in time $Cn + D$, returns an equivalent finite automaton.

Historically, crossing sequences were also used to prove complexity lower bounds for solving problems on one-tape Turing machines (see e.g. [17]) and we present two such lower bounds in Section 5.1.3.

1.3 Literature and Related Work

Chapters 2, 3 and 4 are primarily covered in books by Arora and Barak [2], and Sipser [28]. For most results in these chapters we give our own proofs and reshape the statements so that they fit in the given setting. The results are standard and the additional literature that was used is marked on appropriate places.

Chapters 5 and 6 are based on the papers [10, 11] of the author. While there is quite some other literature about crossing sequences (Chapter 5), the literature for Chapter 6 is harder to find. However, Hájek [19] in the late 1970s proved that it is undecidable whether a given multi-tape DTM runs in time $n + 1$. Roughly at the same time Hartmanis published a monograph [16], where in Chapter 6 he argues about what can and cannot be proven about computational complexity. There, for a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, he compares the class of languages of Turing machines that run in time $T(n)$ to the class of languages of Turing machines that provably run in time $T(n)$. There is also a result of Adachi, Iwata and Kasai [1] from 1984 where they proved good deterministic lower bounds for some problems that are complete in P, a result that has a structure comparable to the structure of Theorem 1.2.1.

Crossing sequences were first studied in the 1960s by Hartmanis [15], Hennie [17], and Trakhtenbrot [30]. In 1968 Hartmanis [15] proved that any one-tape DTM that runs in time $\mathrm{o}(n \log n)$ recognizes a regular language. He acknowledges that Trakhtenbrot [30, in Russian] came to the same result independently. In the proof Hartmanis showed that a one-tape DTM which runs in time

$o(n \log n)$ produces only crossing sequences of bounded length and then he used Hennie's [17] result which tells that such Turing machines recognize only regular languages. Later in 1980s Kobayashi [20] gave another proof of the same result but, in contrast to Hartmanis' approach, his proof gives a way to compute a constant upper bound on the length of the crossing sequences. Recently Tadaki, Yamakami and Lin [29] generalized his proof to show that one-tape NTMs which run in time $o(n \log n)$ accept only regular languages. Their proof also gives a way to compute a constant upper bound on the length of the crossing sequences that such machines can produce. This is essential for the proof of Theorem 6.1.10 which states that we can verify whether a given one-tape NTM obeys a (nice) time bound of order $o(n \log n)$. In [26] Pighizzini showed that one-tape NTMs that run in time $o(n \log n)$ actually run in linear time. A summary of results about one-tape linear-time Turing machines of different types can be found in [29].

Other Work by the Author. Let us only mention two results of the author from his graduate years that are off the topic of the dissertation and will not be discussed further. In [4] the author and Cabello show that very simple algorithms based on local search are polynomial-time approximation schemes for the problems Maximum Independent Set, Minimum Vertex Cover and Minimum Dominating Set, when the input graphs have a fixed forbidden minor. In [9], the author compares convergence properties of some sequences.

Chapter 2

Preliminaries

In this chapter we first introduce basic notation, languages and decision problems. Then we define finite automata and regular expressions and we show that they both describe the same set of languages called regular languages. All the material in this chapter could be taught in an undergraduate theoretical computer science course, so those familiar with the topic might want to skip it.

2.1 Notation, Languages and Problems

2.1.1 Basic Notation

$\mathbb{N}$ ... the set of non-negative integers.

$\mathbb{Z}$ ... the set of integers.

$\mathbb{Q}$ ... the set of rational numbers.

$\mathbb{R}$ ... the set of real numbers.

$\mathbb{R}_{\geq 0}$ ... the set of non-negative real numbers.

$\mathbb{R}_{>0}$ ... the set of positive real numbers.

$\mathcal{P}(A)$ ... the power set of a set A.

For $r \in \mathbb{R}_{\geq 0}$,

$\lfloor r \rfloor$ ... the integer part of r, i.e., the largest integer smaller than or equal to r.

$\lceil r \rceil$ ... the smallest integer greater than or equal to r.

For a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$,

$\lfloor f \rfloor$ ... $\lfloor f \rfloor : \mathbb{N} \to \mathbb{N}$, defined by $\lfloor f \rfloor(n) = \lfloor f(n) \rfloor$.

For functions $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we say that

$f(n) = \mathrm{O}(g(n))$ ... if there exist $k > 0$ and $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, it holds $f(n) \leq k \cdot g(n)$.

$f(n) = \Omega(g(n))$ ... if there exist $k > 0$ and $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, it holds $f(n) \geq k \cdot g(n)$.

$f(n) = \Theta(g(n))$ ... if $f(n) = \Omega(g(n))$ and $f(n) = \mathrm{O}(g(n))$.

$f(n) = \mathrm{o}(g(n))$ ... if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

All logarithms have base 2.

2.1.2 Languages over Alphabets

An *alphabet* is any non-empty finite set of symbols. Two examples are the Slovenian alphabet {a, b, c, č ... ž} and the *binary alphabet* $\{0, 1\}$. For an alphabet Σ and for an integer $i \in \mathbb{N}$, we denote by

Σ^i ... the set of all possible finite sequences of symbols from Σ of length i.

ε ... the empty sequence of symbols. Note that $\Sigma^0 = \{\varepsilon\}$.

Σ_ε ... $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

Σ^* ... the set of all finite sequences of symbols from Σ. Note that

$$\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma^i.$$

Each element of Σ^* is called a *string* over the alphabet Σ. For a string $w \in \Sigma^*$, we denote by

$|w|$ ... the length of w. Clearly, $|\varepsilon| = 0$.

For a string w and for integers $0 \leq i \leq j \leq |w|$, we denote by

$w(i, j)$ ... the string of symbols of w from ith to jth, including the ith symbol and excluding the jth symbol (we start counting with 0). If $i = j$ then $w(i, i) = \varepsilon$. Note that $|w(i, j)| = j - i$ and $w(0, |w|) = w$.

For each i, j, we call $w(i, j)$ a substring of w. For example, the word *ananas* over the Slovenian alphabet is a string of length 6 with a substring *ananas*$(0, 2) =$ *an*. For strings w_1 and w_2 and for $n \in \mathbb{N}$, we denote by

$w_1 w_2$ ... the concatenation of strings w_1 and w_2 and by

w_1^n ... the concatenation of n copies of w_1. If $n = 0$, then $w_1^0 = \varepsilon$.

For example, for strings $w_1 =$ *ananas* and $w_2 =$ *banana*, we have $w_1 w_2 =$ *ananasbanana* and $w_1^2 =$ *ananasananas*.

Any subset of Σ^* is called a *language over an alphabet* Σ. We can imagine the language as the set of strings that mean something. For example, a language over the Slovenian alphabet could be the set of all words found in SSKJ (the dictionary of standard Slovenian language) that are composed exclusively of Slovenian letters.

For languages $L_1, L_2 \subseteq \Sigma^*$ and for an integer $i \in \mathbb{N}$, we denote by

$\overline{L_1}$ ... the complement $\Sigma^* \setminus L_1$ of L_1,

$L_1 L_2$ ... $= \{w_1 w_2;\ w_1 \in L_1, w_2 \in L_2\}$ the concatenation of languages L_1 and L_2,

L_1^i ... $= \{w_1 w_2 \cdots w_i;\ w_1, w_2 \ldots w_i \in L_1\}$. Note that this definition is consistent with the definition of Σ^i where Σ is an alphabet. (We can view Σ as a language over Σ.)

L_1^* ... $= \bigcup_{i \in \mathbb{N}} L_1^i$. This definition is consistent with the definition of Σ^*.

2.1.3 Encodings

Let S be a set of some objects (like graphs, boolean formulas ...) and let Σ be an alphabet. An *encoding* of S over an alphabet Σ is a function f which maps elements of S to pairwise disjoint non-empty subsets of Σ^*. For a fixed encoding f and for an element $s \in S$, we say that each string in $f(s)$ is a code of s.

Example. For example, if S is the *set of all matrices with entries 0 or 1*, then the following $f : S \to \{\text{subsets of } \{0,1,\#\}^*\}$ is an encoding over the alphabet $\{0,1,\#\}$:

$$f\left(\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}\right) = \{a_{11}a_{12}\ldots a_{1n}\#a_{21}a_{22}\ldots a_{2n}\#\ldots\#a_{m1}a_{m2}\ldots a_{mn}\#\}.$$

We see that each matrix A from S has a unique code. However, we could add any number of zeros at the end of each code so that our encoding would be

$$f\left(\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}\right) = \{a_{11}\ldots a_{1n}\#a_{21}\ldots a_{2n}\#\ldots\#a_{m1}\ldots a_{mn}\#0^k;\ k \in \mathbb{N}\}.$$

Note that f maps different matrices to disjoint subsets of $\{0,1,\#\}^*$ and is hence an encoding. The technique we used to get infinitely many codes for each element of S, by just adding redundant symbols to the code, is called *padding*. It will be used several times in the dissertation. ■

When describing an encoding over an alphabet Σ, we usually do not mention Σ if it is clear from the context what alphabet is used, as we can see in the next example.

Example. Let us give an *encoding of weighted directed graphs*[1] whose weights are strings over an alphabet $\tilde{\Sigma}$. We may assume that $\tilde{\Sigma}$ does not contain the symbols $\%$, $\&$ and $\#$ (else we would choose other three special symbols). In our encoding, each graph will have a unique code, hence it is enough to describe the code of some weighted directed graph G. Suppose G has n vertices, hence it can be represented by an $n \times n$ matrix A_G whose entry $A_G[i,j]$ equals the weight on an edge

[1] We suppose that the reader is familiar with basic graph notions. More information about graphs can be found, for example, in [6].

between the ith and the jth vertex, if the edge between these two vertices exists, else $A_G[i,j] = \%$. Then the code of G is:

$$A_G[1,1]\&A_G[1,2]\&\ldots\&A_G[1,n]\#A_G[2,1]\&A_G[2,2]\&\ldots\&A_G[2,n]\#\ldots \\ \ldots\#A_G[n,1]\&A_G[n,2]\&\ldots\&A_G[n,n]\#.$$

Note that the number of vertices n can be deduced from the code. Our encoding is over the alphabet $\tilde{\Sigma} \cup \{\%, \&, \#\}$. ■

Encoding in Binary

It might seem that the choice of an alphabet Σ that is used in an encoding matters a lot, but this is actually not the case. Here we describe one of several possible ways to transform an encoding over an arbitrary alphabet Σ to a related encoding over $\{0,1\}$.

First, we label each symbol from Σ with non-negative integers. Then a string w over Σ can be written over the binary alphabet as follows: we swap each symbol in w with the corresponding binary number with all digits doubled and between each two symbols of w we write 01. It is easy to see that this way all of the codes of an encoding can be transformed so that we get an encoding over $\{0,1\}$.

Example. For example, the word *ananas* viewed as a string over the Slovenian alphabet can be transformed into a binary string

$$110111111111011101111111101110111000011111,$$

using

$$a \mapsto 1$$
$$n \mapsto 1111$$
$$s \mapsto 10011.$$

■

Fixing an Alphabet Σ

Because for all encodings there exists a naturally related binary encoding, we can fix an alphabet Σ that will be used to encode things. We assume also that $0, 1 \in \Sigma$, hence Σ has at least two elements. This is because unary codes, which come from encodings with just one symbol, are too long for our purposes.

Example. One needs unary codes of length at least n to encode numbers $0, 1, 2 \ldots n$. With binary encoding we can encode the same numbers using codes of length $\mathrm{O}(\log(n))$ by just writing the numbers in binary. ■

Natural Encodings

Clearly, every finite or countably infinite set of objects has several encodings over some alphabet, but we are usually interested in encodings that are natural in the sense that:

- Given a code of an object s, one can quickly figure out what s is.
- Given an object s, one can quickly construct one of its codes.

By "quickly", we mean quickly relative to the current technology and developed algorithms. We will only be interested in such (vaguely defined) natural encodings.

What Encodings are Used in this Dissertation

Although the title suggests differently, we will not fix any encodings in this section. This is because there are several good ways of how to encode well studied objects and we do not want to limit ourselves to just one particular encoding. However, for objects that will matter to us, we will describe at least one relevant encoding.

Example. First, we give an *encoding of tuples of strings* that are over the alphabet Σ. We may suppose that the symbols '(' and ')' for brackets and the symbol ',' for comma are not in Σ. Then the unique code of a tuple $(s_1, s_2 \dots s_k)$ is the string $(s_1, s_2 \dots s_k)$. To change the encoding to be over the binary alphabet, we can use the trick described above. ■

Example. As a final example we give an *encoding of matrices with string entries*. We can view a matrix A with string entries as a tuple of rows, where each rows is a tuple of strings. We already described how we can encode tuples, thus if we first encode the rows of A and then encode the tuple of the codes of the rows, then we get the code of the matrix A. Note that we implicitly used the same idea when describing an encoding of weighted graphs. ■

2.1.4 Decision Problems

For a decision problem, we need one (usually infinite) set of elements U called *instances* and a subset $Y \subseteq U$, and the problem is given as:

"Given an instance $x \in U$, is x an element of Y?"

Example. A graph is simple if it does not contain loops and parallel edges. We say that a simple undirected graph G is Hamiltonian, if it admits a cycle (called a Hamiltonian cycle) that contains all of its vertices. The problem HAMILTONIAN CYCLE is the following:

Given a simple undirected graph G, is G Hamiltonian?

In this case U is the set of all graphs and Y is the set of all Hamiltonian graphs. ■

To *solve* a decision problem means the following: For a fixed (natural) encoding of the set U over the alphabet Σ, *find* an algorithm that takes a string $w \in \Sigma^*$ as input and returns the appropriate value "YES" or "NO" depending on whether w is a code of some element $x \in Y$. At this point we can think of an algorithm as a computer program. Later, the word algorithm will be replaced by the phrase Turing machine, which will be well defined in Chapter 3.

If we fix some natural encoding f of U over Σ, then

$$L = \bigcup_{y \in Y} f(y)$$

is a language over the alphabet Σ. Note that L contains exactly the codes of elements of Y, hence to solve our decision problem it is enough to find an algorithm that solves the problem:

"Given $x \in \Sigma^*$, is x an element of L?"

This problem is a special form of a decision problem, given by a language over some alphabet. Solving such a problem is called *deciding* a language $L \subseteq \Sigma^*$. More generally, if we can solve a decision problem, we say that the problem is *decidable*.

2.2 Finite Automata, Regular Languages and Regular Expressions

In this section we define finite automata, regular languages and regular expressions and we present how these notions are related. We follow the book by Sipser [28] where these topics are covered in detail.

2.2.1 Deterministic Finite Automata and Regular Languages

A *deterministic finite automaton* (abbreviated as *DFA*) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

Q ... a finite set whose elements are called *states*,

Σ ... the *input alphabet*, fixed in Section 2.1.3,

δ ... a function $\delta : Q \times \Sigma \to Q$ called the *transition function*,

$q_0 \in Q$ ... the *starting state* and

$F \subseteq Q$ ... the set of *accepting states*.

Computation of a DFA

A DFA $M = (Q, \Sigma, \delta, q_0, F)$ *computes* as follows on an input $w \in \Sigma^*$ of length n. It begins in state q_0. Then it reads the first symbol a_1 of w and moves to the state $q_1 = \delta(q_0, a_1)$. Then it reads the second symbol a_2 of w and moves to the state $\delta(q_1, a_2)$. Then it reads the third symbol ... Then it reads the last symbol a_n of w and moves to the state $q_n = \delta(q_{n-1}, a_n)$. Then it halts. If $q_n \in F$, then we say that M *accepts* w, else it *rejects* w. The set of all strings accepted by M is called the *language* of M and is denoted by $L(M)$. We also say that M *recognizes* the language $L(M)$.

Regular Languages

A language $L \subset \Sigma^*$ is *regular* if it is recognized by some DFA. Because DFAs are a very simple model of computation, they can solve only a small amount of (decision) problems (see Section 4.2.2), hence the set of regular languages is a small subset of all decidable languages.

Presenting a DFA with a Graph

We can present a DFA $M = (Q, \Sigma, \delta, q_0, F)$ as a weighted directed multigraph[2] with the vertex set Q, one special vertex $q_0 \in Q$ and the set of special vertices $F \subseteq Q$. There is an edge from $q_1 \in Q$ to $q_2 \in Q$ with weight $a \in \Sigma$ if and only if $\delta(q_1, a) = q_2$.

The computation of M starts in the vertex q_0 and then it continues along the appropriate edges depending on the symbols of the input which are read one by one from left to right. The computation terminates when all the symbols of the input have been considered. If the last transition was to a state from F, then the computation is accepting, else it is rejecting.

Example. Let us consider a DFA $M_1 = (\{q_0, q_1, q_2, r_1, r_2\}, \{0, 1\}, \delta, q_0, \{q_1, r_1\})$, where δ is given by Table 2.1. The corresponding weighted multigraph is drawn in Figure 2.1.

We leave to the reader to verify that M_1 accepts exactly such binary inputs that begin and end with the same symbol. Hence, M_1 recognizes the language L of all binary strings that begin and end with the same symbol. ■

[2] A multigraph is a graph that can have parallel edges and loops.

δ	0	1
q_0	q_1	r_1
q_1	q_1	q_2
q_2	q_1	q_2
r_1	r_2	r_1
r_2	r_2	r_1

Table 2.1: The transition function of DFA M_1.

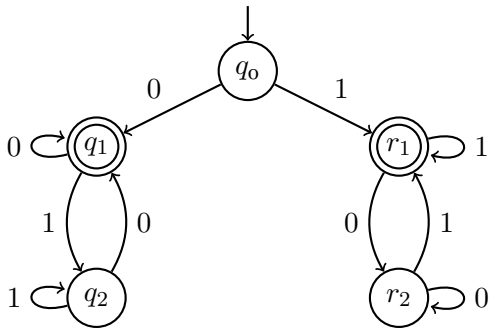


Figure 2.1: DFA M_1 as a weighted multigraph. The starting state is labeled by an incoming arrow and the accepting states are marked by two circles.

The simplest languages are the finite ones. Let us show that they are all regular.

Proposition 2.2.1. *Every finite language $L \subseteq \Sigma^*$ is regular.*

Proof. We will prove the proposition by constructing a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that will recognize L. Let w be the longest string form L and let $n_0 = |w|$ be the length of w. Let Q be the set of all strings from Σ^* of length at most n_0, together with an additional state q_{REJ} that we call the rejecting state. Let $q_0 = \varepsilon \in Q$ be the starting state. For a state q that is a string of length strictly less than n_0 and for $a \in \Sigma$, define $\delta(q, a) = qa$, where qa is the concatenation of the string q and the symbol a. For each string q of length n_0 and for each $a \in \Sigma$, define $\delta(q, a) = q_{\text{REJ}}$. For each $a \in \Sigma$, define $\delta(q_{\text{REJ}}, a) = q_{\text{REJ}}$.

Now it is clear that if we define $F = L$, we get $L(M) = L$, hence L is regular. □

Now that we have some regular languages, let us also give an example of a simple non-regular language.

Example. A palindrome over an alphabet Σ is any string from Σ^* that reads the same from left to right. Examples of palindromes over the Slovenian alphabet are ε, *a*, *aa*, *anana* and *banab*, but not *ananas* or *banana*. Consider the decision problem PALINDROME that is the following:

Given $w \in \Sigma^*$, is w a palindrome?

If L is the language of palindromes over Σ, we will show that L it is not regular. If it would be, then there would be a finite automaton M with q states that would recognize L. Let us suppose that this is true and derive a contradiction. Because $|\Sigma|^q > q$, there exist at least two inputs w_1 and w_2

for M of length q such that M is in the same state after it finishes computation on w_1 or w_2. If w_1^R denotes the reversed string of w_1, i.e., the string that reads as w_1 if we read from right to left, then M will return the same on the inputs $w_1 w_1^R$ and $w_2 w_1^R$, which is a contradiction because $w_1 w_1^R$ is a palindrome and $w_2 w_1^R$ is not.

Later in Proposition 5.1.10 we will prove that the language of palindromes cannot be recognized even by a much stronger model of computation than a DFA. ■

2.2.2 Non-Deterministic Finite Automata

A *non-deterministic finite automaton* (abbreviated as *NFA*) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ is the starting state, $F \subseteq Q$ is the set of accepting states, Σ is the input alphabet fixed in Section 2.1.3 and

$$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$$

is the transition function. Recall that $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\mathcal{P}(Q)$ is the power set of Q.

The definitions of DFAs and NFAs are very similar, they also have similar presentations with weighted multigraphs. As by DFAs, we can present an NFA $M = (Q, \Sigma, \delta, q_0, F)$ as a weighted directed multigraph with the vertex set Q, one special vertex $q_0 \in Q$ and a set of special vertices $F \subseteq Q$. There is an edge from $q_1 \in Q$ to $q_2 \in Q$ with a weight $a \in \Sigma_\varepsilon$ if and only if $q_2 \in \delta(q_1, a)$. The only difference with DFAs in the presentation with multigraphs is that for a DFA, each vertex has exactly $|\Sigma|$ edges going out, one for each symbol, while an NFA can have any number of edges with any labels going out from a vertex. An example of an NFA is given in Figure 2.2.

Example. Let us consider an NFA $M_2 = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_0\})$, where δ is given by Table 2.2. The corresponding weighted multigraph is drawn in Figure 2.2. ■

δ	0	1	ε
q_0	$\{q_1\}$	$\{\}$	$\{q_2\}$
q_1	$\{q_2\}$	$\{q_1, q_2\}$	$\{\}$
q_2	$\{\}$	$\{q_0\}$	$\{\}$

Table 2.2: The transition function of NFA M_2.

Computation of an NFA

We will explain how an NFA $M = (Q, \Sigma, \delta, q_0, F)$ computes on an input w by using a graph representation of M. M does not compute deterministically which means that, for a single input, there are several possible computations. A computation begins in the state q_0 and M either stays in state $q_1 = q_0$ or it moves to some new state q_1 that is reachable from q_0 by edges labeled ε. Then M reads the first symbol a_1 of w. If $\delta(q_1, a_1) = \emptyset$, then M halts, else it moves to a state q_2' from $\delta(q_1, a_1)$ and it either stays in state $q_2 = q_2'$ or it moves to a new state q_2 that is reachable from q_2' by edges labeled ε. ... At the end, M reads the last symbol a_n of w. If $\delta(q_n, a_n) = \emptyset$, then M halts, else it moves to a state q_{n+1}' from $\delta(q_n, a_n)$ and it either stays in state $q_{n+1} = q_{n+1}'$ or it moves to a new state q_{n+1} that is reachable form q_{n+1}' by edges labeled ε. If $q_{n+1} \in F$, then we say that the computation is *accepting*. Else, the computation is *rejecting*. We say that M *accepts* the input w if there exists an accepting computation of M on w. If not, then M *rejects* w.

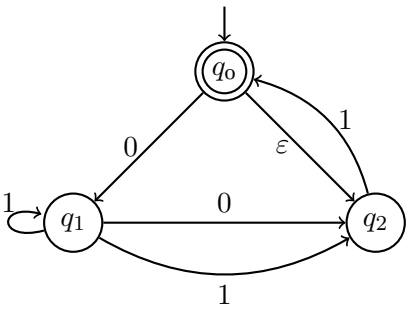


Figure 2.2: NFA M_2 as a weighted multigraph. The starting state is labeled by an incoming arrow and the accepting states are marked by two circles.

To define what strings are accepted by M more formally, we say that M *accepts* $w \in \Sigma^*$ if w can be written as $w = y_1 y_2 \dots y_m$, where each y_i is a member of Σ_ε and if there exists a sequence of states $r_0, r_1 \dots r_m$ of M with three conditions:

- $r_0 = q_0$,
- $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, 1 \dots m-1$, and
- $r_m \in F$.

The first condition says that the machine starts out in the starting state. The second condition says that when in state r_i, the machine can move to the state r_{i+1} if it reads the symbol y_{i+1}. Finally, the third condition says that the machine accepts its input if the last state is an accepting state.

Example. The NFA M_2 from Figure 2.2 accepts exactly such binary inputs that are composed of blocks

- 1^i for $i \in \mathbb{N}$,
- 011 and
- 01^i01 for $i \in \mathbb{N}$.

The proof is left to the reader. ■

Equivalence of DFAs and NFAs

It is clear from the definition that each DFA $M = (Q, \Sigma, \delta, q_0, F)$ is also an NFA, if we imagine the transition function of M mapping $(q, a) \in Q \times \Sigma$ to the set $\{\delta(q, a)\}$ and if we extend the definition of the transition function to the domain $Q \times \Sigma_\varepsilon$ by $(q, \varepsilon) \mapsto \emptyset$. However, despite NFAs seeming somewhat stronger, they accept only regular languages and are thus *equivalent* to DFAs.

Proposition 2.2.2. *For each NFA, there exists a DFA that recognizes the same language.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing some language $L \subseteq \Sigma^*$. To describe a DFA $M' = (Q', \Sigma, \delta', q_0', F')$ that will recognize L, let us first define Q', q_0' and F'.

- $Q' = \mathcal{P}(Q)$ is the power set of Q. This definition reveals the main idea of how M' will simulate M: it will keep track of which states M can be when reading some input symbol.
- $q'_0 = S$, where S is the set of all the states that are either q_0 or are reachable from q_0 by an arrow labeled ε. Note that S is the set of all the possible states in which M can be before it reads the first symbol of an input.
- F' is the set of all the states $X \in Q'$ with $X \cap F$ not empty. Clearly, if M can reach an accepting state after reading the last symbol of an input, M' should also accept that input.

To describe the transition function δ', let us first define a function $\gamma : Q \times \Sigma \to Q'$ as

$\gamma(q, a) =$ the set of all the possible states of M that can be reached by M if it starts in the state q, then follows some edges labeled ε, then it follows an edge labeled a and then again it follows some edges labeled ε.

For $X \in Q'$ and $a \in \Sigma$, define

$$\delta'(X, a) = \bigcup_{q \in X} \gamma(q, a).$$

We see that on an input w before M' scans its ith symbol, its state is the set of all the states in which M can be when scanning the ith symbol. Thus M' accepts the same language as M. □

Corollary 2.2.3. *A language is regular if and only if some NFA recognizes it.*

Proof. The *if* part is proven by Proposition 2.2.2 and the *only if* part is clear because DFAs are a special form of NFAs. □

Operations on Regular Languages

In this section we first give three operations on languages called the *regular operations* and then we prove that the set of regular languages is closed under these operations. We also prove that the set of regular languages is closed under complementation and intersection.

There are three *regular operations*:

- The union $L_1 \cup L_2$ of languages L_1 and L_2,
- the concatenation $L_1 L_2$ of languages L_1 and L_2 and
- the star L^* of a language L.

The concatenation of languages and the star operation were defined in Section 2.1.2. Let us prove that regular languages are closed under regular operations.

Proposition 2.2.4. *A language that is obtained from regular languages by regular operations is regular.*

Proof. Let L_1 and L_2 be regular languages. Then there exists a DFA M_1 that recognizes L_1 and a DFA M_2 that recognizes L_2. We make proofs by picture to show how to construct

- an NFA that recognizes $L_1 \cup L_2$ (see Figure 2.3),
- an NFA that recognizes $L_1 L_2$ (see Figure 2.4),

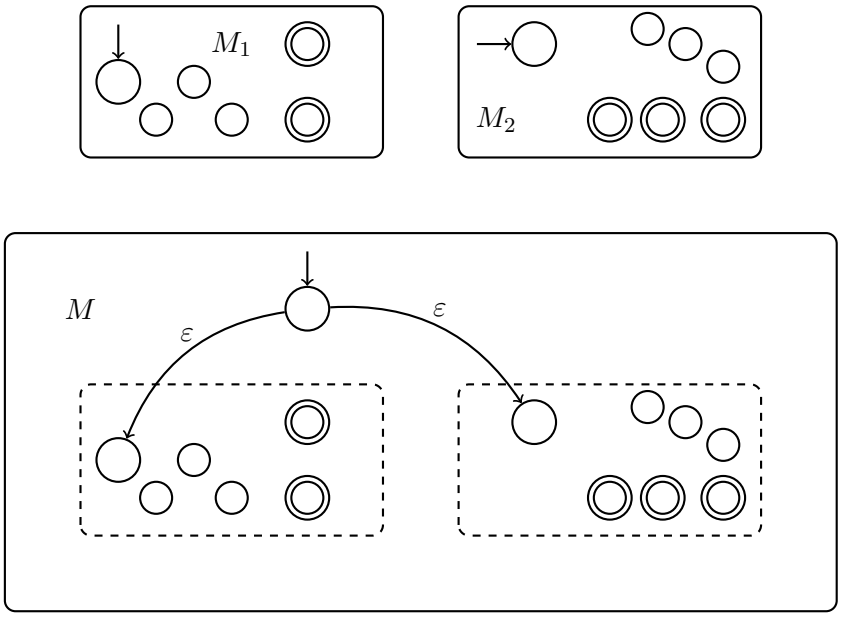


Figure 2.3: Finite automata M_1, M_2 and M. If M_1 recognizes the language L_1 and M_2 recognizes the language L_2, then M recognizes the language $L_1 \cup L_2$. The starting states of the automata are labeled by incoming arrows and their accepting states are marked by two circles.

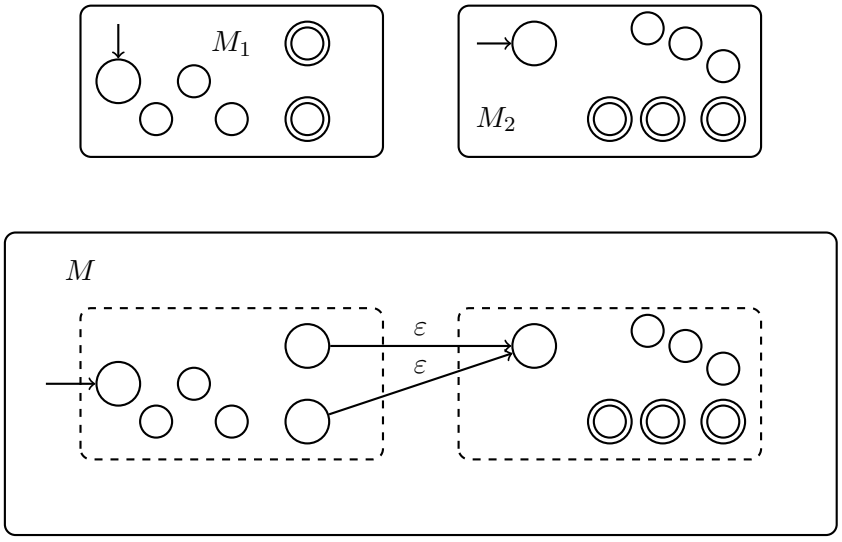


Figure 2.4: Finite automata M_1, M_2 and M. If M_1 recognizes the language L_1 and M_2 recognizes the language L_2, then M recognizes the language $L_1 L_2$. The starting states of the automata are labeled by incoming arrows and their accepting states are marked by two circles.

- an NFA that recognizes L_2^* (see Figure 2.5).

By Corollary 2.2.3 the languages $L_1 \cup L_2$, $L_1 L_2$ and L_2^* are regular. □

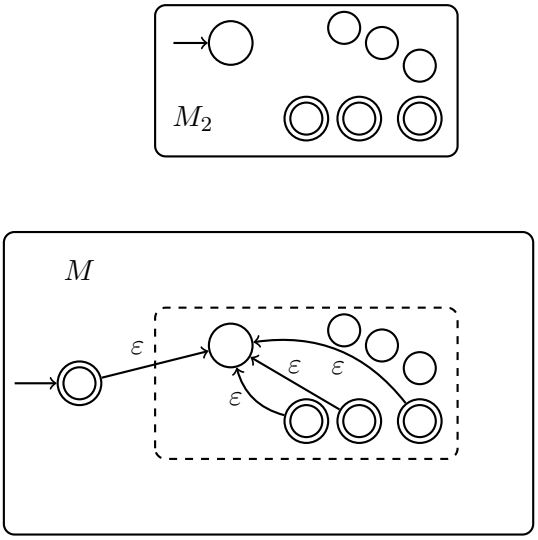


Figure 2.5: Finite automata M_2 and M. If M_2 recognizes the language L_2 then M recognizes the language L_2^*. The starting states of the automata are labeled by incoming arrows and their accepting states are marked by two circles.

Next, we show that the class of regular languages is closed under complementation.

Proposition 2.2.5. *The complement of a regular language is regular.*

Proof. Let L be a regular language. Then there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes L. It is clear that the DFA $(Q, \Sigma, \delta, q_0, Q \backslash F)$ recognizes $\overline{L}$, hence $\overline{L}$ is regular. □

A simple corollary of the above two results tells that regular languages are closed under intersection.

Corollary 2.2.6. *The intersection of two regular languages is a regular language.*

Proof. Let L_1 and L_2 be regular languages. By Proposition 2.2.5, the languages $\overline{L_1}$ and $\overline{L_2}$ are regular, by Proposition 2.2.4 the language $\overline{L_1} \cup \overline{L_2}$ is regular and again by Proposition 2.2.5, the language $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ is regular. □

2.2.3 Regular Expressions

Regular expressions are a way of describing a special type of languages which we will prove are exactly the regular languages. They are defined inductively as follows. We say that R is a *regular expression* (over the alphabet Σ) if R is either

1. a for some $a \in \Sigma$,
2. ε
3. $\emptyset$

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions (here $\cup$ is just a symbol, like $+$ in the arithmetic expression $a + b$),

5. $(R_1 R_2)$, where R_1 and R_2 are regular expressions, or

6. R_1^*, where R_1 is a regular expression.

In items 1 and 2 of the definition, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expression represents the languages obtained by taking the union or concatenation of the languages given by expressions R_1 and R_2, or the star of the language given by R_1, respectively.

Example. Let us consider the languages L_1 and L_2 recognized by the DFA M_1 from Figure 2.1 and by the NFA M_2 form Figure 2.2, respectively. If we recall, L_1 is a binary language of strings that start and end with the same symbol and L_2 is the binary language of strings that are composed of blocks

- 1^i for $i \in \mathbb{N}$,
- 011 and
- 01^i01 for $i \in \mathbb{N}$.

We know that these two languages are regular, however they are also given by the following two regular expressions. L_1 is given by

$$\big(1(0 \cup 1)^*1\big) \cup \big(0(0 \cup 1)^*0\big) \cup 1 \cup 0$$

and L_2 is given by

$$(1 \cup 011 \cup 01^*01)^*.$$ ■

The next theorem tells us that languages given by regular expressions are exactly the regular languages.

Theorem 2.2.7. *A language is regular if and only if some regular expression describes it.*

Proof. To prove the *if* part, we make an induction on the number of regular operations in a regular expression. If a regular expression contains no regular operation, then it represents a regular language by Lemma 2.2.1. Else, let R be a regular expression that has $k \geq 1$ regular operations. Then $R = R_1^*$ for some regular expression R_1 with $k - 1$ regular operations, or $R = R_1 R_2$ for some regular expressions R_1 and R_2 with at most $k - 1$ regular operations each, or $R = R_1 \cup R_2$ for some regular expressions R_1 and R_2 with at most $k - 1$ regular operations each. By induction, R represents a language that is in each of these cases obtained by a regular operation from regular languages. Hence R represents a regular language by Proposition 2.2.4.

To prove the *only if* part, we introduce a new type of automata. A *generalized non-deterministic finite automaton* (abbreviated as *GNFA*) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, q_{\text{ACC}})$ where Q is a finite set of states, $q_0, q_{\text{ACC}} \in Q$ are distinct starting and accepting states, Σ is the input alphabet fixed in Section 2.1.3 and

$$\delta : Q \backslash \{q_{\text{ACC}}\} \times Q \backslash \{q_0\} \to \{\text{regular expressions over } \Sigma\}$$

is a transition function.

We can present M as a weighted directed graph with loops on the vertex set Q, where the weight on the edge q_iq_j is $\delta(q_i, q_j)$. Note this graph is "almost complete", having all the possible edges except for the edges from q_{ACC} and the edges to q_0. It also has $|Q| - 2$ loops. An example of such a graph is given in Figure 2.6.

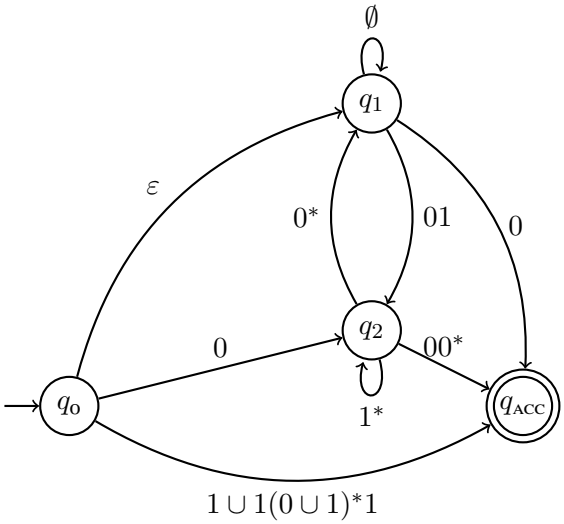


Figure 2.6: A GNFA represented as a weighted graph. The starting state is labeled by an incoming arrow and the accepting state is marked by two circles. It is clear that this GNFA accepts only binary strings that begin and end with the same symbol and we leave to the reader to verify that it accepts all such strings and is thus equivalent to the DFA in Figure 2.1.

As the name GNFA suggests, M will compute non-deterministically, hence there will be several possible computations on a single input. Given an input w, M first chooses an edge going from the starting state and a substring $w(0, j_1)$ of first few (possibly 0) letters of w. If $w(0, j_1)$ is an element of the language given by the regular expression on the chosen edge, then M takes this edge and comes to state q_1. As in the first step, M then chooses an edge going from the state q_1 and a substring $w(j_1, j_2)$ of the first few (possibly 0) letters of $w(j_1, |w|)$. If $w(j_1, j_2)$ is an element of the language given by the regular expression on the chosen edge, then M takes this edge. The computation continues this way for finitely many steps. If M in the last step enters the state q_{ACC} and during the computation it read the whole input (i.e., the parts of the input that were used to travel through the states can be concatenated into w), then we say that the computation is *accepting*. Else, it is *rejecting*. If there exists an accepting computation on an input w, then we say that M *accepts* w. Else, M *rejects* w. Note that a GNFA computes exactly like an NFA, only that it reads blocks of symbols from the input instead of just a single symbol or ε.

Now that we have defined the GNFAs, we can prove the *only if* part of the theorem. Let L be a regular language and let M be a DFA that recognizes it. We first convert M to a GNFA $\tilde{M}$:

- We add a new starting state q_0 and connect it with the old one by an edge with weight ε.
- We add an accepting state q_{ACC} and we connect each of the old accepting states with q_{ACC} by an edge with weight ε.

- For each ordered pair of vertices (q_i, q_j), if there are multiple edges from q_i to q_j, we delete them and we put a single edge from q_i to q_j with the weight that is the union of the weights of the deleted edges. If $q_i \neq q_{\text{ACC}}$ and $q_j \neq q_{\text{o}}$ and there is no edge from q_i to q_j, we add such an edge and label it with $\emptyset$.

It is clear that $\tilde{M}$ is a GNFA that accepts the same language as M. Next, we are going to delete vertices of $\tilde{M}$ one by one leaving q_{o} and q_{ACC} intact and changing the weights on the edges in such a way that, after a deletion of a vertex, the automaton will recognize exactly the same language as before. If we manage to delete all the vertices of $\tilde{M}$ except of q_{o} and q_{ACC}, we will be left with a GNFA with just two vertices and one edge (from q_{o} to q_{ACC}) that accepts the same language as $\tilde{M}$, which is L. Hence, the weight on the remaining edge will be a regular expression that represents L.

The only thing left to show is how to delete a vertex from $\tilde{M}$ while not changing the language it accepts. This is evident in Figure 2.7 where it is shown how the weight on every edge from $\tilde{M}$ should change when a vertex is deleted so that the language that $\tilde{M}$ accepts remains the same. □

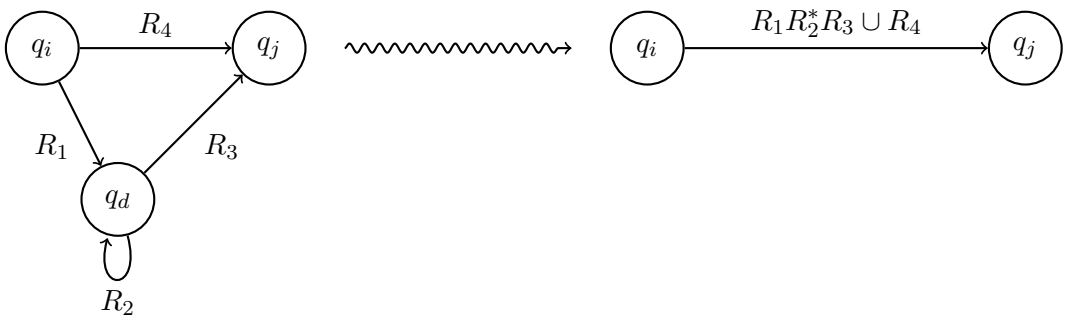


Figure 2.7: Changing the regular expression R_4 on the edge from q_i to q_j when deleting a vertex q_d of a GNFA. The same transformation is applied for every edge of the GNFA that is not incident to q_d.

Chapter 3

Turing Machines

Turing machines are a standard model of computation in theoretical computer science. They are very simple, however powerful enough to simulate our computers. In this chapter we discuss the basics (definition, running time, time related complexity classes, Church-Turing thesis, classes P, NP and NP) and some technical topics about Turing machines (Section 3.3 and Section 3.6). While most of the content in this chapter is on undergraduate level, there are some complex simulations presented in Section 3.3, like fast simulations of a multi-tape Turing machine on a two-tape one. In Section 3.6 we present an encoding of Turing machines that enables a construction of a code of the composition of two Turing machines in linear time.

3.1 One-Tape Turing Machines

A *one-tape non-deterministic Turing machine* (abbreviated as one-tape *NTM*) is an 8-tuple $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$, where

Q	...	a finite set of states,
Σ	...	the input alphabet fixed in Section 2.1.3,
$\Gamma \supset \Sigma$	...	a tape alphabet,
$\sqcup \in \Gamma \backslash \Sigma$	...	a blank symbol,
$\delta : \Gamma \times Q \backslash \{q_{\text{ACC}}, q_{\text{REJ}}\} \to \mathcal{P}(\Gamma \times Q \times \{-1, 1\}) \backslash \{\emptyset\}$		
	...	a transition function and
$q_0, q_{\text{ACC}}, q_{\text{REJ}} \in Q$	...	pairwise distinct starting, accepting and rejecting states. The states q_{ACC} and q_{REJ} are called *halting states*.

The machine M has a both-way infinite tape and we can number its cells as can be seen in Figure 3.1. To give an input w to M means to write symbols of w on tape cells $0, 1 \ldots (|w| - 1)$ and to write the blank symbol $\sqcup$ on all other tape cells. Before M starts the computation, it has the head over the cell 0 and it is in the state q_0. During a computation, M moves its head one cell at a time, overwriting what is written below it with a symbol from Γ and changing the current state. A next step of M depends only on the current state and the symbol below the head. If M is in a state q and below the head is a symbol a, then M can make a steps described by any of the triples from

$\delta(a, q)$. What this means is that if $(b, r, d) \in \delta(a, q)$, then M can rewrite a below the head by b, change the state into r and move in direction d (-1 for one cell left, 1 for one cell right). M halts when it reaches a halting state.

CELLS:	$\cdots$	-3	-2	-1	0	1	2	3	4	5	$\cdots$

Figure 3.1: Numbering of tape cells of a one-tape Turing machine. The shaded part is a potential input of length 4.

3.1.1 The Formal Definition of a Computation of a One-Tape NTM

A computation is more formally defined by a sequence of valid configurations.

- A *valid configuration* of M is a triple $C = (q, w_1, w_2)$, where q is a state of M, and w_1, w_2 are non-empty strings over the alphabet Γ. If $q = q_{\text{REJ}}$, then we call C the *rejecting* configuration and if $q = q_{\text{ACC}}$, then we call C the *accepting* configuration. If $q = q_{\text{o}}$, $w_1 = \sqcup$ and $w_2 = w_{\sqcup}$ where $w \in \Sigma^*$, then we call C the starting configuration for an input w. Intuitively, a valid configuration describes a hypothetical situation that can happen during a computation of M. The state q represents the current state of the machine, the tape has the string w_1w_2 written on it while there are only blank symbols left and right of w_1w_2 and the head is above the first symbol from w_2. Note that a valid configuration does not include the information about where on the tape the string w_1w_2 is written, i.e., where is the cell 0 relative to w_1w_2.
- A valid configuration $C_1 = (q, w_1, w_2)$ that is not accepting nor rejecting *yields* a configuration $C_2 = (r, u_1, u_2)$ if M can legally go from C_1 to C_2 in a single step. If $w_2 = av_2$ for $a \in \Gamma$ and $v_2 \in \Gamma^*$, then C_2 can be any of the valid configurations obtained the following way:
  - Take any $(b, r, d) \in \delta(a, q)$.
  - If $d = 1$ then $u_1 = w_1b$. If $|v_2| = 0$ then $u_2 = \sqcup$, else $u_2 = v_2$.
  - If $d = -1$, let $w_1 = v_1c$ for $v_1 \in \Gamma^*$ and $c \in \Gamma$. Then $u_2 = cbv_2$. If $|v_1| = 0$, then $u_1 = \sqcup$, else $u_1 = v_1$.
- A *computation* of M is any sequence of valid configurations $C_1, C_2, C_3 \ldots$ such that C_1 is the starting configuration, configuration C_{i-1} yields C_i for each i and the sequence is infinite or ends in an accepting or rejecting configuration. If the computation is finite, we say that it is *accepting* or *rejecting* depending on the final configuration. A transition from one configuration of a computation to the next one is called a *step*.

We see that M can have several possible computations on a single input. We say that M *accepts* an input w if there exists an accepting computation on w, otherwise we say that M rejects w.

3.1.2 Giving an Input to a Turing Machine

It is clear from the definition of a computation that *to give an input $w \in \Sigma^*$ to a one-tape NTM* M means to write the symbols of w in the tape cells from 0 to $|w| - 1$ of the input tape of M

and to fill all other cells with blank symbols (see Figure 3.1). However, we will often describe the input for M in the way as "M is given a pair (w, u), $w, u \in \Sigma^*$" or "M is given a (code of a) graph G" without specifying a particular encoding. And even if an encoding would be specified, not all strings from Σ^* necessarily represent a valid input, e.g. a pair of strings or a graph. When we describe an input in this manner, we have some fixed natural encoding of objects in mind and we allow the reader to have its own fixed natural encoding in mind. Additionally, we treat strings that are not codes of any objects as codes of some fixed trivial object, like the pair $(\varepsilon, \varepsilon)$ or the graph with just one vertex.

3.1.3 Running Time

The number of steps that a one-tape NTM M makes on some computation ζ is called *the length of* ζ and is denoted by $|\zeta|$. Note that $|\zeta|$ could also be ∞. For a function $T : \mathbb{N} \to \mathbb{R}_{>0}$, we say that M *runs in time* $T(n)$ if M makes **at most** $T(n)$ steps on all computations on all inputs of length n, for all $n \in \mathbb{N}$. Note that if a Turing machine runs in time $T(n)$, then all computations are finite. We say that M *runs in time* $\mathrm{O}(T(n))$ if it runs in time $\tilde{T}(n) = \mathrm{O}(T(n))$ for some function $\tilde{T} : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

- If M runs in time $\mathrm{O}(1)$, then we say that it runs in *constant time*.
- If M runs in time $\mathrm{O}(n)$, then we say that it runs in *linear time*.
- If M runs in time $\mathrm{O}(p(n))$ for some polynomial p, then we say that it runs in *polynomial time*.
- If M runs in time $\mathrm{O}(2^{p(n)})$ for some polynomial p, then we say that it runs in *exponential time*.

Our definition of running time is the same as in Arora and Barak [2, Chapter 2.1.2].

3.1.4 Language of a Turing Machine

A *language* L *of* M is the set of all inputs from Σ^* that are accepted by M. We say that M *recognizes* L and we write $L = L(M)$. If additionally M halts on all computations on all inputs, we say that M *decides* L. If M runs in time $T(n)$ and decides L, then we say that M *decides* L *in time* $T(n)$.

3.1.5 About Our Definition of a One-Tape NTM

As can be seen from the definition, the head of M must move in each step of a computation. The same property is assumed also by e.g. Sipser [28]. We will use this property when discussing crossing sequences in Chapter 5 because it will hold that the sum of the lengths of all crossing sequences equals the number of steps. If we allowed the head to stay in place, we would have to change the definition of the length of a computation on a part (Section 5.2.1).

We also assume that at the end of each finite computation the head of M is in a halting state (q_{ACC} or q_{REJ}), as can be seen from the definition of the transition function (it cannot map to the empty set). This is a minor assumption and we can use it without loss of generality. It helps us in a way that we know in which state the computation ends (rejecting or accepting or it runs for ever). Additionally it implies that an NTM makes at least one step on each input, which is naturally true

for deterministic Turing machines as we shall see in the next subsection. This further implies that if a Turing machine runs in time $T(n)$, then $T(n) \geq 1$ for all n.

Finally, because we fixed the alphabet Σ used for encodings (see Section 2.1.3), our Turing machines have a fixed input alphabet.

3.1.6 One-Tape Deterministic Turing Machines

A *one-tape deterministic Turing machine* (abbreviated as one-tape *DTM*) is an 8-tuple $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$, where Q is a finite set of states, Σ an input alphabet, $\Gamma \supset \Sigma$ a tape alphabet, $\sqcup \in \Gamma \backslash \Sigma$ a blank symbol,

$$\delta : \Gamma \times Q \backslash \{q_{\text{ACC}}, q_{\text{REJ}}\} \to \Gamma \times Q \times \{-1, 1\}$$

a transition function and $q_0, q_{\text{ACC}}, q_{\text{REJ}} \in Q$ pairwise distinct starting, accepting and rejecting states.

A one-tape DTM is actually a special form of a one-tape NTM, where the transition function maps each $(a, q) \in \Gamma \times Q \backslash \{q_{\text{ACC}}, q_{\text{REJ}}\}$ to a set with exactly one element. Hence, a DTM has exactly one computation on each input.

3.2 Multi-Tape Turing Machines

For an integer $k \geq 2$, a *k-tape non-deterministic Turing machine* is an 8-tuple $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$, where Q is a finite set of states, $q_0, q_{\text{ACC}}, q_{\text{REJ}} \in Q$ pairwise distinct starting, accepting and rejecting states, Σ the input alphabet fixed in Section 2.1.3, $\Gamma \supset \Sigma$ a tape alphabet, $\sqcup \in \Gamma \backslash \Sigma$ a blank symbol and

$$\delta : \Gamma^k \times Q \backslash \{q_{\text{ACC}}, q_{\text{REJ}}\} \to \mathcal{P}(\Gamma^k \times Q \times \{-1, 0, 1\}^k) \backslash \{\emptyset\}$$

a transition function.

The machine M has k both-way infinite tapes, where one of them is special and is called the input tape. To give an input w to M means the same as to give an input to a one-tape NTM, if we consider only the input tape and all other tapes are filled only with blank symbols. Before M starts the computation, it has the head on the input tape over the cell 0 (on other tapes the head is over any cell because they are all equivalent) and it is in the state q_0. During a computation, M moves its heads one cell at a time, possibly overwriting what is written below them with symbols from Γ and changing the current state. A next step of M depends only on the current state and the symbols below all the k heads. If M is in a state q and below the heads are symbols $(a_1, a_2 \dots a_k)$, then M can make any of the steps from $\delta(a_1, a_2 \dots a_k, q)$.

We will not describe a formal definition of computation as we did by one-tape NTMs. It is quite cumbersome but very intuitive and analogous to the formal definition of a computation of a one-tape NTM. The definition of running time and languages of k-tape Turing machines is the same as by one-tape Turing machines.

A *multi-tape non-deterministic Turing machine* is a one-tape NTM or a k-tape NTM for $k \geq 2$.

3.2.1 About Our Definition of a Multi-Tape NTM

For $k \geq 2$ and for a k-tape Turing machine M, the heads of M do not need to move on each step of a computation. This is because it is easier to construct a Turing machine if you do not need to care about moving each head in each step. Recall that for one-tape Turing machines we wanted their

head to move in each step so that the analyses of crossing sequences would be easier. For multi-tape Turing machines, the author is not aware of any notion analogous to crossing sequences.

3.2.2 Multi-Tape Deterministic Turing Machines

A *k-tape deterministic Turing machine* is a special form of a k-tape NTM where the transition function maps each element to a set with exactly one element (analogously to one-tape DTMs).

3.3 How Different Attributes of Turing Machines Influence the Time Complexity

We defined Turing machines in such a way that we have the following:

$$\begin{array}{lllll} \{\text{one-tape DTMs}\} & \subseteq & \{\text{one-tape NTMs}\} & \subseteq & \{\text{multi-tape NTMs}\} \quad \text{and} \\ \{\text{one-tape DTMs}\} & \subseteq & \{\text{multi-tape DTMs}\} & \subseteq & \{\text{multi-tape NTMs}\}. \end{array}$$

Hence, all Turing machines defined so far are multi-tape NTMs and one-tape DTMs are of all types.

In this section we try to answer the following question: If a language L is decided in time $T(n)$ by one type of a Turing machine, how fast can it be decided on other types of Turing machines? In addition to the above types we also consider a 2-tape Turing machine and a Turing machine that can make at most 2 non-deterministic choices in each step. This will tell us how strong our models are when considering running time. In Section 3.3.2 we prove a so-called linear speedup theorem (Corollary 3.3.7) which tells that, for each language L that is decided in time $T(n)$ by some NTM and for each constant $\ell \in \mathbb{N}$, there exists a multi-tape NTM that decides L in time

$$2^{-\ell}T(n) + (1 - 2^{-\ell})n + 1,$$

which is asymptotically $2^{-\ell}T(n)$ if $n = \mathrm{o}(T(n))$.

We start by a simple lemma that holds for all types of Turing machines defined so far in this dissertation.

Lemma 3.3.1. *If a multi-tape NTM M runs in time $T(n)$ and there exists $n_0 \in \mathbb{N}$ such that $T(n_0) < n_0 + 1$, then*

- *M never reads the $(n_0 + 1)$st symbol of any input,*
- *M runs in constant time,*
- *there exist finite languages $L_1 \subseteq \bigcup_{i=0}^{n_0-1} \Sigma^i$ and $L_2 \subseteq \Sigma^{n_0}$ such that $L(M) = L_1 \cup (L_2\Sigma^*)$,*
- *M accepts a regular language.*

Proof. Suppose an input w of length at least $n_0 + 1$ is given to M and let $w_0 = w(0, n_0)$ be the starting substring of w of length n_0. Because M on any computation on input w_0 makes at most n_0 steps, it never reads the blank symbol to the right of w_0. This means that M on input w never reads the $(n_0 + 1)$st symbol of w and hence it computes the same as on the input w_0. This implies that M on any input of length more than n_0 makes exactly as many steps as on some input of length n_0.

Because M makes at most $T(n)$ steps on any computation on inputs of length n, there are only finitely many computations on inputs of length at most n_0. If ζ is a longest such a computation, then M runs in time $|\zeta|$.

Let $L_1 \subseteq L(M)$ be the set of all strings from $L(M)$ of length strictly less that n_0 and let $L_2 \subseteq L(M)$ be the set of all strings from $L(M)$ of length exactly n_0. We see that $L(M) = L_1 \cup (L_2\Sigma^*)$.

Because L_1 and L_2 are finite, then by Proposition 2.2.1 they are regular. Because Σ^* is also a regular language, $L(M)$ is regular by Proposition 2.2.4. □

3.3.1 Reducing the Tape Alphabet

In this section we will show that a larger tape alphabet does not help much in reducing time complexity for deciding a language. First, we consider a special case, analyzed also in Lemma 3.3.1.

Lemma 3.3.2. *Let $T : \mathbb{N} \to \mathbb{R}_{>0}$ be such that $T(n_0) < n_0 + 1$ for some integer $n_0 \in \mathbb{N}$. If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM, then there exists a one-tape DTM that decides L in time $\mathrm{O}(1)$ using the tape alphabet $\Sigma \cup \{\llcorner\}$.*

Proof. By Lemma 3.3.1 there exist finite languages

$$L_1 \subseteq \bigcup_{i=0}^{n_0-1} \Sigma^i \qquad \text{and} \qquad L_2 \subseteq \Sigma^{n_0}$$

such that $L = L_1 \cup (L_2\Sigma^*)$. Let us describe a one-tape DTM M that decides L in time $\mathrm{O}(1)$. On an input w, the machine M first verifies whether $w \in L_1$ or whether w begins with some string from L_2. If so, then M accepts, else it rejects. This can be done in $\mathrm{O}(1)$ deterministic steps without changing the content of the tape, because M never needs to visit the $(n_0 + 1)$st cell of its tape. □

Next, we prove that the size of Γ does not matter a lot when considering multi-tape Turing machines with more than one tape.

Proposition 3.3.3. *For an integer $k > 1$, if a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a k-tape Turing machine M, then it is also decided by a k-tape Turing machine of the same type (NTM or DTM) in time $\mathrm{O}(T(n))$ using the tape alphabet $\Sigma \cup \{\llcorner\}$.*

Proof. If there is an integer $n_0 \in \mathbb{N}$ such that $T(n_0) < n_0 + 1$, then by Lemma 3.3.2 we can define a one-tape DTM $\tilde{M}$ that decides L in time $\mathrm{O}(1)$ using the tape alphabet $\Sigma \cup \{\llcorner\}$. Because $T(n)$ is a running time of some Turing machine it follows that $T(n) = \Omega(1)$, hence $\tilde{M}$ runs in time $\mathrm{O}(T(n))$. We can add $k - 1$ redundant tapes to $\tilde{M}$ so that the proposition holds.

We are only left with the case where $T(n) \geq n + 1$ for all $n \in \mathbb{N}$. Let Γ be the tape alphabet of M and let each symbol of $\Gamma \setminus \{\llcorner\}$ be represented by a unique binary sequence of length exactly $\lceil \log(|\Gamma| - 1) \rceil$. Let us describe a k-tape Turing machine $\tilde{M}$ of the same type as M that uses the tape alphabet $\Sigma \cup \{\llcorner\}$ and decides $L(M)$ in time $\mathrm{O}(T(n))$.

On an input $w \in \Sigma^*$, $\tilde{M}$ first rewrites the input in such a way that each input symbol is replaced by its binary sequence. Using the second tape, this can be done in time $\mathrm{O}(n)$. Note that the length of the input was increased by a factor of $\lceil \log(|\Gamma| - 1) \rceil$. Next, $\tilde{M}$ simulates M step by step, using only the binary codes of symbols from $\Gamma \setminus \{\llcorner\}$ and using $\lceil \log(|\Gamma| - 1) \rceil$ consecutive blank symbols as a one blank symbol of M. For each step of M, $\tilde{M}$ makes $\mathrm{O}(\lceil \log(|\Gamma| - 1) \rceil) = \mathrm{O}(1)$ steps, hence $\tilde{M}$ runs in time $\mathrm{O}(n + T(n)) = \mathrm{O}(T(n))$. □

The next proposition tells us what is different with one-tape Turing machines.

Proposition 3.3.4. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a one-tape Turing machine M, then it is also decided by a one-tape Turing machine of the same type (NTM or DTM) in time $\mathrm{O}(T(n) + n^2)$ using the tape alphabet $\Sigma \cup \{\llcorner\}$.*

Proof. The proof is the same as that of Proposition 3.3.3, only that when encoding the input of M in binary, we use $\mathrm{O}(n^2)$ steps because we have only one tape. □

3.3.2 Linear Speedup

In the preceding section we saw in Proposition 3.3.3 that we can reduce the tape alphabet of a multi-tape Turing machines to the minimal one if we allow a Turing machine to run for a multiplicative constant factor longer. In this section we will show the converse: We can speed up a Turing machine if we allow it to use more tape symbols or additional tapes.

Proposition 3.3.5. *For $k > 1$, if a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a k-tape NTM M, then it is also decided by some k-tape NTM $\tilde{M}$ in time $\lceil \frac{1}{2}T(n) \rceil + \lceil \frac{3}{2}n \rceil + 1$. If M is deterministic, then $\tilde{M}$ can also be deterministic.*

Proof. The idea for the construction of $\tilde{M}$ is very simple. We can define tape symbols $\tilde{\Gamma}$ of $\tilde{M}$ so that they will represent two symbols from adjacent tape cells of M. Then we simulate two steps of M in just one step of $\tilde{M}$ by ensuring that all information needed to perform two steps of M is below the heads of $\tilde{M}$ and stored in the states of $\tilde{M}$.

Define the tape alphabet $\tilde{\Gamma}$ of $\tilde{M}$ as

$$\tilde{\Gamma} = \Sigma \cup \{\llcorner\} \cup \Gamma^2,$$

where Γ is the tape alphabet of M. Let us now explain how $\tilde{M}$ computes on an input $w = w_1 w_2 \dots w_n$, where $w_i \in \Sigma$ for all i. First, $\tilde{M}$ writes the following on the second tape:

$$(\llcorner, \llcorner)(w_3, w_4)(w_5, w_6) \dots (w_{n-1}, w_n)$$

or

$$(\llcorner, \llcorner)(w_3, w_4)(w_5, w_6) \dots (w_n, \llcorner),$$

depending on whether n is odd or even, and it deletes the content of the first tape. Note that the first two symbols of the input are nowhere on the tape. However, $\tilde{M}$ stores them using states. Then $\tilde{M}$ copies the content of the second tape to the first tape leaving only blank symbols on the second tape, which enables the head of the first tape to be over the "first" symbol $(\llcorner, \llcorner)$ of the new input. This all can be done in at most $\lceil \frac{3}{2}n \rceil + 1$ steps. In the simulation of M that follows, $\tilde{M}$ will use only symbols from Γ^2 on all tapes and it will treat each blank symbol as $(\llcorner, \llcorner) \in \Gamma^2$.

$\tilde{M}$ simulates steps of M two by two. Let us first present the *main invariant* of the simulation. If the content of the ith tape of $\tilde{M}$ before simulating a jth and $(j+1)$st step of M is

$$\dots \llcorner \llcorner (a_1, a_2)(a_3, a_4) \dots (a_{2l+1}, a_{2l+2}) \llcorner \llcorner \dots$$

and the ith head of $\tilde{M}$ is above the symbol (a_{2m+1}, a_{2m+2}), then the content of the ith tape of M before the jth step is either

$$\ldots \sqcup \sqcup a_1 a_2 a_3 \ldots a_{2m-1} a_{2m} b_1 b_2 a_{2m+1} a_{2m+2} a_{2m+3} \ldots a_{2l+1} a_{2l+2} \sqcup \sqcup \ldots \tag{3.1}$$

or

$$\ldots \sqcup \sqcup a_1 a_2 a_3 \ldots a_{2m} a_{2m+1} a_{2m+2} b_1 b_2 a_{2m+3} a_{2m+4} \ldots a_{2l+1} a_{2l+2} \sqcup \sqcup \ldots \tag{3.2}$$

where the symbols b_1 and b_2 are stored using the states of $\tilde{M}$. The ith head of M before a jth step is over one of the symbols b_1, b_2, a_{2m+1} or a_{2m+2}. Even more is true, if the case (3.1) is the right one, then the head of M is above one of the symbols b_2 or a_{2m+1}, and if the case (3.2) is the right one, then the head of M is above one of the symbols b_1 or a_{2m+2}. Which case is the right one and where the ith head of M lies is stored using the states of $\tilde{M}$.

It is clear that the information that $\tilde{M}$ has before simulating the jth and the $(j+1)$st step of M is enough to perform two steps of M in a single step of $\tilde{M}$, because for each head, $\tilde{M}$ knows the state, the symbols below the heads, at least one symbol to the left of each head and at least one symbol to the right of each head.

Let us show how $\tilde{M}$ can maintain the main invariant. We have to consider four basic cases: (3.1) and (3.2) and for each of them there are 2 possibilities where the ith head of M is. For each of these cases, the ith head of M after two steps can stay in place or move one or two cells to the left or to the right. Because of the symmetry, we will only deal with the case (3.1).

- Suppose M has its head over the symbol b_2.
  - If the position of the ith head of M changes for two cells to the left after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ moves one cell to the left, not changing the content of its ith tape, but remembering in its state that we now have case (3.2) with the ith head of M above the right symbol below the head of $\tilde{M}$. Also the symbols b_1 and b_2 get updated.
  - If the position of the ith head of M changes for one cell to the left in the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ moves one cell to the left, not changing the content of its ith tape, but remembering in its state that we now have case (3.2) with the ith head of M above the (updated) symbol b_1. Also the symbol b_2 gets updated.
  - If the position of the ith head of M remains the same after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ stays in place and the symbols b_1, b_2 and a_{2m+1} get updated.
  - If the position of the ith head of M changes for one cell to the right after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ stays in place and the symbols b_2 stored in the state and a_{2m+1} below the ith head get updated. Again we have case (3.1), but with the ith head of M above the (updated) symbol a_{2m+1}.
  - If the position of the ith head of M changes for two cells to the right after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ moves one cell right and the symbols b_1 and the updated symbol b_2 are written on the tape where the pair (a_{2m+1}, a_{2m+2}) was before. In the state, $\tilde{M}$ remembers the updated symbol a_{2m+1} and the symbol a_{2m+2}. We now have case (3.1) again with the ith head of M above the symbol a_{2m+2} stored in the state.

- Suppose M has its head over the symbol a_{2m+1}.
  - If the position of the ith head of M changes for two cells to the left after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ moves one cell left, remembering in its state that we now have case (3.2) with the ith head of M above the symbol b_1. Also the symbols b_2 and a_{2m+1} get updated.
  - If the position of the ith head of M changes for one cell to the left after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ stays in place and the symbols b_2 stored in the state and a_{2m+1} below the ith head get updated. Again we have case (3.1), but with the ith head of M above the updated symbol b_2.
  - If the position of the ith head of M remains the same after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ stays in place and the symbols b_2, a_{2m+1} and a_{2m+2} get updated.
  - If the position of the ith head of M changes for one cell to the right after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ moves one cell to the right, writing the symbol (b_1, b_2) instead of (a_{2m+1}, a_{2m+2}) and remembering the updated symbols a_{2m+1} and a_{2m+2} in the state. We again have case (3.1) with the head over the updated symbol a_{2m+2} stored in the states.
  - If the position of the ith head of M changes for two cells to the right after the jth and the $(j+1)$st step, then the ith head of $\tilde{M}$ moves one cell right and the symbols b_1 and the symbol b_2 are written on the tape where the pair (a_{2m+1}, a_{2m+2}) was before. In the state, $\tilde{M}$ remembers the updated symbols a_{2m+1} and a_{2m+2}. We now have case (3.1) again with the ith head of M above the symbol a_{2m+3}.

It is clear that $\tilde{M}$ runs in time $\lceil \frac{1}{2}T(n) \rceil + \lceil \frac{3}{2}n \rceil + 1$. □

It might seem that the increase of the alphabet or the number of states is necessary to reduce the running time. However, another way to reduce the running time is to increase the number of tapes of a Turing machine.

Lemma 3.3.6. *For $k \geq 1$, if a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a k-tape NTM M, then it is also decided by some $(4k+3)$-tape NTM $\tilde{M}$ in time $\lceil \frac{1}{2}(T(n) - n) \rceil + n$ where $\tilde{M}$ has the same tape alphabet Γ and the same set of states Q as M. If M is deterministic, then $\tilde{M}$ can also be deterministic.*

Proof. As in the proof of Proposition 3.3.5, the idea is very simple. $\tilde{M}$ will simulate M in such a way, that it will use four tapes to handle one tape of M. The four tapes that will handle the ith tape of M will be called the ith block. The input tape and two additional tapes called the *counting* tape and the *parity* tape of $\tilde{M}$ will be special and will not be in any block. We can number the cells of each of M's tapes with integers as in Figure 3.1 (for non-input tapes, the cell 0 coincides with the position of the corresponding head before the beginning of a computation). Let the first of the four tapes of the ith block contain exactly the cells of the ith tape of M that are labeled by integers divisible by 3, let the second tape contain exactly the cells labeled by integers congruent to 1 modulo 3 and let the third tape contain exactly the cells labeled by integers congruent to 2 modulo 3. The fourth tape of the ith block will only be supplementary and the head above this tape will never move. If there will be a symbol $\sqcup$ below this head, this will mean that the ith head of M is on a cell labeled by an integer divisible by 3. If there will be a symbol 0 below this head, this

will mean that the ith head of M is on a cell labeled by an integer congruent to 1 modulo 3 and if there will be a symbol 1 below this head, this will mean that the ith head of M is on a cell labeled by an integer congruent to 2 modulo 3.

$\tilde{M}$ will interleave the copying of the input to the first block and the simulation of a computation of M. In each step, $\tilde{M}$ will simulate exactly one step of M or exactly two steps of M, depending on whether M would read a new input symbol or not. If M would read a new input symbol, then $\tilde{M}$ will simulate only this (one) step of M, else it will simulate exactly two steps of M. Let us explain the invariants during the computation of $\tilde{M}$.

For each block i, except for the first block, the following will hold: before the simulation of the jth (and possibly the $(j+1)$st) step of M, the content of each of the first three tapes of the ith block will be as described in the first paragraph of this proof (each tape contains every third symbol of the ith tape of M before the jth step). The fourth tape of the ith block tells us on which of the three tapes the head of M should be and the head on this tape is exactly on the cell that corresponds to the cell of M with the ith head on it. The heads on the other two tapes are on the cells that correspond to the left and the right cell of where the head of M is. For the first block (representing the input tape of M), the same holds except that instead of the input symbols that have not been read yet, there are blank symbols.

To support the first block which partially describes the input tape of M, we have three more tapes: the input tape, the counting tape and the parity tape. On the input tape, the head will always be on the leftmost unread input symbol, if there exists such. If all the symbols of the input have been read, the head on the input tape will be on some blank symbol and will not move for the rest of the simulation of M. Also the counting tape and the parity tape will become redundant when the input is read. Before this happens, the counting tape will measure how far left of the leftmost unread input symbol the first head of M is. There will be only one non-blank symbol on the counting tape, say 0, which is written in the first step of $\tilde{M}$. If the head on the counting tape will be x cells left of the symbol 0, this means that the first head of M is either $2x$ or $2x+1$ cells left from the leftmost unread input symbol. The function of the parity tape is only to store the information whether the first head of M is $2x$ or $2x+1$ cells left from the leftmost unread input symbol. Hence, the head of the parity tape does not need to move at all and it needs only to change two symbols, say 0 and 1. If there is a blank symbol below the head of the parity tape, we know that the computation has not yet begun and that $\tilde{M}$ has to write 0 on the counting tape in the first (i.e., next) step and one of the symbols 0 or 1 on the parity tape, depending on whether the input head of $\tilde{M}$ moves or not.

We now know that $\tilde{M}$ will always simulate exactly two steps of M except for the following three exceptions when it will simulate exactly one step of M.

1. The head on the parity tape reads the blank symbol $\sqcup$ ($\tilde{M}$ is starting its computation) and the head on the input tape of $\tilde{M}$ reads a symbol from Σ (not the blank symbol).

2. The head on the counting tape reads the symbol 0, the head on the parity tape reads the symbol 0 and the head on the input tape of $\tilde{M}$ reads a symbol from Σ (not the blank symbol).

3. M goes to a halting state in the next step (q_{ACC} or q_{REJ}).

This altogether implies that $\tilde{M}$ runs in time $\lceil \frac{1}{2}(T(n)-n) \rceil + n$.

$\tilde{M}$ only uses the states of M to maintain all the invariants. If $\tilde{M}$ has to simulate the jth and possibly the $(j+1)$st step of M, then $\tilde{M}$ is in the same state as M before these two steps. If the input head of M before the jth step is not above a new symbol of the input, then $\tilde{M}$ has enough information below its heads to simulate two steps of M, else it has enough information to simulate one step of M. It is clear that all the invariants can be maintained. □

If we use Lemma 3.3.6 several times, we can, for any positive constant c, decrease any superlinear[1] running time $T(n)$ of a multi-tape Turing machine down to $n+1$ for small inputs and down to $\frac{1}{c}T(n)$ for long enough inputs, as it can be deduced from the following corollary. With such a transformation, the Turing machine gets several additional tapes.

Proposition 3.3.7. *For every constant $\ell \in \mathbb{N}$, if a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM M, then it is also decided by some multi-tape NTM $\tilde{M}$ in time*

$$2^{-\ell}T(n) + (1 - 2^{-\ell})n + 1$$

where $\tilde{M}$ has the same tape alphabet Γ and the same set of states Q as M. If M is deterministic, then $\tilde{M}$ can also be deterministic.

Proof. Using Lemma 3.3.6 ℓ times we get a multi-tape Turing machine $\tilde{M}$ that has the same tape alphabet Γ and the same set of states Q as M. It runs in time

$$\begin{aligned}\left\lceil \frac{1}{2}\left(\cdots\left\lceil \frac{1}{2}\left(\left\lceil \frac{1}{2}(T(n)-n)\right\rceil + n - n\right)\right\rceil \cdots + n - n\right)\right\rceil + n \\ = \left\lceil \frac{1}{2}\cdots\left\lceil \frac{1}{2}\left\lceil \frac{1}{2}(T(n)-n)\right\rceil\right\rceil\cdots\right\rceil + n,\end{aligned}$$

where there are ℓ divisions by 2 on each side of the equality. Using that

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil$$

holds for all $x \in \mathbb{R}$ and positive integers a, b, we get that $\tilde{M}$ runs in time

$$\left\lceil 2^{-\ell}(T(n)-n)\right\rceil + n \leq 2^{-\ell}T(n) + (1-2^{-\ell})n + 1,$$

which proves the corollary. □

3.3.3 Reducing the Number of Tapes

In this section we show how we can reduce the number of tapes of a Turing machine so that the new machine would recognize the same language and it would run only slightly slower than the original Turing machine. We give three results: in the first result we reduce the number of tapes to 1 and in the second (non-deterministic case) and the third (deterministic case) result we reduce the number of tapes to 2.

In the following proposition we show how to reduce a multi-tape Turing machine to a one-tape Turing machine by only squaring the running time. Later in Proposition 5.1.10 we show that this is optimal by proving that the problem PALINDROME cannot be decided by a one-tape NTM in time $o(n^2)$ while it can clearly be decided in linear time by a 2-tape DTM.

Proposition 3.3.8. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM M, then it is also decided by some one-tape NTM $\tilde{M}$ in time $O(T(n)^2)$. If M is deterministic, then $\tilde{M}$ can also be deterministic.*

[1] A function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is superlinear if $\lim f(n)/n = \infty$.

Proof. Suppose that the multi-tape NTM $M = (Q, \Sigma, \Gamma, \llcorner, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$ has $k > 1$ tapes. We will define a one-tape NTM $\tilde{M} = (\tilde{Q}, \Sigma, \tilde{\Gamma}, \llcorner, \tilde{\delta}, \tilde{q_0}, q_{\text{ACC}}, q_{\text{REJ}})$ that will simulate M on just one tape and will run in time $\mathrm{O}(T(n)^2)$. This will be done by encoding configurations of a computation of M on just one tape, using more symbols. Note that for each configuration of M, we need to know what symbols are on each tape, where the heads are on each tape and what is the current state of M. The content of the tapes and the positions of the heads of M will be encoded on the tape of $\tilde{M}$, while the current state of M will be remembered by the states of $\tilde{M}$.

To define the tape alphabet $\tilde{\Gamma}$, let Γ' be the alphabet that has exactly the same symbols as Γ, except that symbols in Γ' have an additional apostrophe. In other words, for each $a \in \Gamma$ we have a' in Γ'. Without the loss of generality we may assume that $\Gamma \cap \Gamma' = \emptyset$. We define

$$\tilde{\Gamma} = \Sigma \cup \{\llcorner\} \cup \left(\Gamma \cup \Gamma'\right)^k$$

where we assume that the sets Σ, $\{\llcorner\}$ and $(\Gamma \cup \Gamma')^k$ are pairwise disjoint.

Let us describe the meaning of the symbols in $\tilde{\Gamma}$ and how they help us in encoding a configuration of M. Symbols from $\Sigma \cup \{\llcorner\}$, while being initially on the tape, will be replaced by symbols from $(\Gamma \cup \Gamma')^k$ as soon as the head of $\tilde{M}$ comes across them. When this happens, a symbol $a \in \Sigma \cup \{\llcorner\}$ will be treated the same as if there was the symbol $(a, \llcorner, \llcorner \ldots \llcorner)$ instead of a. The only exception is at the start of a computation when the first symbol a of the input is treated as $(a', \llcorner', \llcorner' \ldots \llcorner')$. (If the input is ε, then $a = \llcorner$.) The meaning of a symbol $(x_1, x_2 \ldots x_k) \in (\Gamma \cup \Gamma')^k$ is the following: if $x_i \in \Gamma'$, then the head of the ith tape of M is above this symbol, else x_i is a symbol in a cell of the ith tape of M and the ith head of M is not above this cell. This way we can maintain on just one tape the content of the tapes and head positions of M during the simulation.

During the simulation of M, there will always be blank symbols on the left part of the tape of $\tilde{M}$, followed by some cells with symbols from $(\Gamma \cup \Gamma')^k$, followed only by symbols from $\Sigma \cup \{\llcorner\}$. We call the part of the tape of $\tilde{M}$ with symbols from $(\Gamma \cup \Gamma')^k$ written on it the *visited part* of the tape. To simulate one step of M, the Turing machine $\tilde{M}$ will pass through the visited part of the tape left to rigth or vice versa three times, each time adding a new cell to the visited part. If $\tilde{M}$ is passing to the left, then one symbol is added to the visited part on its left side, else one symbol is added to the visited part on its right side. In the first of the three passes, $\tilde{M}$ reads and remembers (using states) all the k symbols below the heads of M in the current configuration of M. Then it uses non-determinism to decide which non-deterministic choice to simulate for the next step of M and in the next two passes, it changes the symbols on the $\tilde{M}$'s tape so that they represent the next configuration of M. When passing to the left it simulates the heads of M that move to the left in the simulated step and when passing to the right, it simulates the heads of M that move to the right in the simulated step.

Suppose an input of length n is given to $\tilde{M}$. Note that for each simulated step of M, the visited part of the tape of $\tilde{M}$ gets increased by 3 cells. Because we simulate at most $T(n)$ steps, the visited part contains at most $3T(n)$ cells at the end of the simulation. Hence, for each simulated step of M, $\tilde{M}$ makes at most $9T(n)$ steps which sums up to $9T(n)^2 = \mathrm{O}(T(n)^2)$ steps altogether. □

In the simulation of multi-tape Turing machines by one-tape ones in Proposition 3.3.8 we used many new symbols in the tape alphabets of one-tape Turing machines. The following corollary tells us that this was in fact not need.

Corollary 3.3.9. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM M, then it is also decided by some one-tape NTM $\tilde{M}$ in time $\mathrm{O}(T(n)^2)$ using the tape alphabet $\Sigma \cup \{\llcorner\}$. If M is deterministic, then $\tilde{M}$ can also be deterministic.*

Proof. If there is some integer $n_0 \in \mathbb{N}$ such that $T(n_0) < n_0 + 1$, then by Lemma 3.3.2 there exists a one-tape DTM $\tilde{M}$ that decides L in time $\mathrm{O}(1)$ using the tape alphabet $\Sigma \cup \{\sqcup\}$. Because $T(n)$ is a running time of some Turing machine, it follows that $T(n) = \Omega(1)$, hence $\tilde{M}$ runs in time $\mathrm{O}(T(n)^2)$.

Now suppose that $T(n) \geq n + 1$ for all n. By Proposition 3.3.8 we get a one-tape NTM $\bar{M}$ that decides L in time $\mathrm{O}(T(n)^2)$ and by Lemma 3.3.4 we can find a one-tape Turing machine $\tilde{M}$ that decides L in time $\mathrm{O}(T(n)^2 + n^2)$ and uses the tape alphabet $\Sigma \cup \{\sqcup\}$. Because $T(n) \geq n+1$ for all n, $\tilde{M}$ runs in time $\mathrm{O}(T(n)^2)$. Lemma 3.3.4 and Proposition 3.3.8 tell also that if M is deterministic, then $\tilde{M}$ can also be deterministic. □

In the next proposition we show how the time complexity changes when reducing a multi-tape NTM to a 2-tape machine. The idea of the proof is from Seiferas, Fischer and Meyer [27].

Proposition 3.3.10. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM M, then it is also decided by some 2-tape NTM $\tilde{M}$ in time $\mathrm{O}\big(T(n)\big)$.*

Proof. The NTM $\tilde{M}$ will compute in two main phases. In the first phase, $\tilde{M}$ will non-deterministically guess "snapshots" of a computation of M and in the second phase it will verify whether the snapshots could arise in some accepting computation of M. Then $\tilde{M}$ will alternate between these two phases a few times to simulate enough steps of M. $\tilde{M}$ will accept an input if and only if M would.

Let $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$ be a k-tape NTM. A *snapshot* of a configuration of M is a $(k+1)$-tuple consisting of the current state of M and the current symbols below the heads of M. The snapshot determines which actions are legal as the next move and whether the configuration is an accepting one or a rejecting one. Given a snapshot $(q, a_1, a_2 \ldots a_k) \in Q \times \Gamma \times \cdots \times \Gamma$, the set of legal moves is exactly $\delta(q, a_1, a_2 \ldots a_k)$.

Suppose an input w is given to $\tilde{M}$. In the first phase, for some $\ell \in \mathbb{N}$ determined later, $\tilde{M}$ on the second tape non-deterministically guesses an alternating sequence of length ℓ of snapshots and legal moves of M. Note that to guess a legal move, $\tilde{M}$ needs only the information about the preceding snapshot. However, the sequence of snapshots and legal moves may not correspond to a valid computation, but we will deal with this in the second phase. Note that if $\tilde{M}$ uses a big enough alphabet so that it can encode any legal move or a snapshot in a single symbol, then $\tilde{M}$ needs only ℓ cells on the second tape to write down the sequence. If some snapshot is from a halting configuration, i.e., it contains a halting state, then $\tilde{M}$ does not write down the rest of the sequence of snapshots and legal moves.

In the second phase, $\tilde{M}$ deterministically verifies whether the sequence of snapshots and legal moves on the second tape corresponds to a legal computation of M on the input w. It does so one tape of M at a time, using its first tape and head to perform exactly as the M's tape and head that are being simulated and the second tape to tell the next move. Not to forget the input w, $\tilde{M}$ stores it using more symbols which encode two symbols at a time: the original one from the input tape of $\tilde{M}$ and another one from the tape of M currently being simulated. Hence, the input tape of $\tilde{M}$ actually "contains" two tapes: the unchanged input tape of $\tilde{M}$ and the tape of M currently being simulated. Note that in the second phase $\tilde{M}$ can verify in time $\mathrm{O}(k\ell) = \mathrm{O}(\ell)$ whether the snapshots and legal moves from the first phase correspond to a legal computation of M on the input w. There are three possible outcomes of the second phase.

- If the sequence from the first phase does not correspond to a legal computation of M on the input w, then $\tilde{M}$ rejects.

- If the sequence from the first phase corresponds to a legal computation of M on the input w that finishes in a halting configuration, then $\tilde{M}$ returns the same as M would, i.e., if the halting configuration is accepting, then $\tilde{M}$ accepts, else it rejects.
- If the sequence from the first phase corresponds to a legal computation of M on the input w and it does not finish in a halting configuration, then $\tilde{M}$ restores the input and runs the first phase again, this time guessing a sequence of length 2ℓ. Note that, using the first tape to measure ℓ, $\tilde{M}$ can do the first phase in time $\mathrm{O}(\ell)$.

Suppose that initially $\ell = 1$. Because M runs in time $T(n)$, ℓ will get at most $4T(n)$ on the inputs of length n. Hence, $\tilde{M}$ runs in time

$$\mathrm{O}(1) + \mathrm{O}(2) + \mathrm{O}(4) + \mathrm{O}(8) + \cdots + \mathrm{O}(2^{\lceil \log 4T(n) \rceil}) = \mathrm{O}(T(n)).$$

It is clear that $\tilde{M}$ accepts an input w if and only if there exists an accepting computation of M on the input w, hence $\tilde{M}$ decides the same language as M. □

The next corollary tells us that when reducing a multi-tape NTM to a 2-tape NTM, we do not need to increase the tape alphabet to get the same performance.

Corollary 3.3.11. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM M, then it is also decided by some 2-tape NTM $\tilde{M}$ with the tape alphabet $\Sigma \cup \{\llcorner\}$ in time $\mathrm{O}\big(T(n)\big)$.*

Proof. Proposition 3.3.10 gives us a two-tape Turing machine that decides L and runs in time $\mathrm{O}\big(T(n)\big)$ and Proposition 3.3.3 assures that this Turing machine can have the tape alphabet $\Sigma \cup \{\llcorner\}$. □

Finally, we show how the time complexity changes when reducing a multi-tape DTM to a 2-tape DTM. Because we have to compute deterministically this time, we cannot "guess" the computation, hence we need another clever method. While the original idea of the proof is from Hennie and Stearns [18], we will follow the proof from [2, Chapter 1].

Proposition 3.3.12. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape DTM M, then it is also decided by some 2-tape DTM $\tilde{M}$ in time $\mathrm{O}\big(T(n) \log T(n)\big)$.*

Proof. If there is some integer $n_0 \in \mathbb{N}$ such that $T(n_0) < n_0 + 1$, then by Lemma 3.3.2 there exists a one-tape DTM $\tilde{M}$ that decides L in time $\mathrm{O}(1)$ using the tape alphabet $\Sigma \cup \{\llcorner\}$. It follows that there exists a 2-tape DTM that decides L and runs in time $\mathrm{O}\big(T(n) \log T(n)\big)$. Hence, we may suppose that $T(n) \geq n + 1$ for all n.

The idea of the proof is very similar to the idea of the proof of Proposition 3.3.8. If we recall, we increased the tape alphabet of M in such a way that we could encode all of its tapes on just one tape. We also marked the positions of all the k heads. Then we needed to pass between the leftmost cell that marked the position of some head and the rightmost one, which resulted in the need of $\mathrm{O}(T(n))$ steps of $\tilde{M}$ to simulate one step of M. Hence, to simulate M more efficiently, we need to keep all of the k heads together. This will be done in such a way that, instead of moving the heads, we will actually be moving the content of the tapes of M left and right.

If M has the tape alphabet Γ, then $\tilde{M}$ will have the tape alphabet

$$\tilde{\Gamma} = \Sigma \cup \{\llcorner\} \cup \big((\Gamma \cup \{\#\})^k \times \{0,1\}\big),$$

where $\# \notin \Gamma$. Let us explain the meaning of a symbol $(a_1, a_2, a_3 \dots a_k, b) \in (\Gamma \cup \{\#\})^k \times \{0,1\}$. The symbol $b \in \{0,1\}$ will mark the beginnings of zones of the tape of $\tilde{M}$ defined in the next paragraph. For all i, if $a_i \in \Gamma$ then a_i will correspond to some symbol on the ith tape of M, else $a_i = \#$ which means that this symbol should be ignored when considering the ith tape of M. After each simulated step of M, if we consider only the ith element of the symbols from $(\Gamma \cup \{\#\})^k \times \{0,1\}$ on the input tape of $\tilde{M}$ and we discard the symbols $\#$, we get exactly the content of the ith tape of M.

The input tape of $\tilde{M}$ will be the main simulating tape and, before simulating a step of M, it will encode the exact configuration of M before this step. The second tape, however, will serve only as a tool for fast shifting of content of the first tape from one place to another. So let us explain how the first tape will be organized. Its cells are marked with integers as in Figure 3.1 and we divide them into zones $R_0, L_0, R_1, L_1 \dots$ as follows. The cell at the location 0 is not in any zone, the cells 1 and 2 belong to the zone R_0 while the cells -1 and -2 belong to the zone L_0. The cells $3, 4, 5$ and 6 belong to the zone R_1 while the cells $-3, -4, -5$ and -6 belong to the zone L_1. Generally, for every $i \geq 1$, the zone R_i contains the first 2^{i+1} cells that are right of the zone R_{i-1} and the zone L_i contains the first 2^{i+1} cells that are left of the zone L_{i-1} (see Figure 3.2). When simulating M, $\tilde{M}$ will try to keep the positions of all the heads of M over the cell 0, shifting some of the content of each tape through zones.

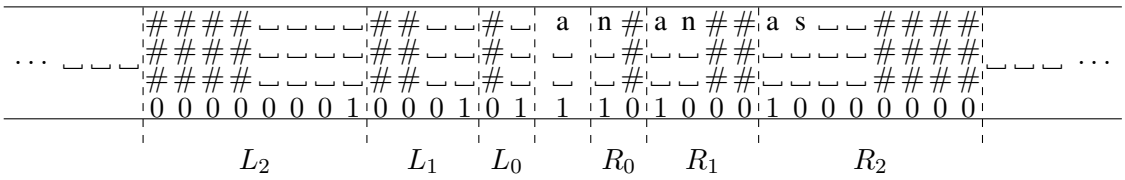


Figure 3.2: The input tape of $\tilde{M}$ just before simulating a first step of a 3-tape Turing machine M on the input *ananas*, where each column represents one symbol of $\tilde{\Gamma}$. The initialized zones are marked below the tape and they are all i-half-full for $i = 1, 2, 3$.

During the simulation of M, $\tilde{M}$ will maintain the following invariants:

- For each tape i of M and for each zone R_j, the zone will be *i-empty*, *i-full* or *i-half-full* with non-$\#$ symbols, which means that the number of non-$\#$ symbols on the ith coordinate of the symbols in the zone R_j will be either 0, 2^{j+1} or 2^j. The same will hold for the zone L_j.

  We assume that initially, i.e., before the zone R_j is initialized, it is i-half-full for every i. The zone R_j will be initialized as soon as the head of $\tilde{M}$ steps into R_j or L_j. The same holds for the zone L_j, hence the zones R_j and L_j are always initialized together. When a zone is initialized, it is filled to the half with symbols $(\sqcup, \sqcup \dots \sqcup, 0)$ and in the other half with symbols $(\#, \# \dots \#, 0)$. The exceptions are the leftmost symbol in the zone R_j and the rightmost symbol in the zone L_j that are initially $(\sqcup, \sqcup \dots \sqcup, 1)$, with a 1 marking the beginning of a new zone.

- For each tape i of M and for each zone R_j, the total number of non-$\#$ symbols on the ith coordinate of symbols in R_j and L_j will be 2^{j+1}. That is, either R_j is i-empty and L_j is i-full, or R_j is i-full and L_j is i-empty or they are both i-half-full.

- Cell 0 of the tape of $\tilde{M}$ does not contain $\#$ in any coordinate at any time.

Let us explain how $\tilde{M}$ prepares for the simulation of M given some input. First, it initializes as many zones as needed to cover all of the input and it remembers the input in the first coordinates of the new symbols on the tape (see Figure 3.2). Clearly, this can be done in time $\mathrm{O}(n)$.

Next, we explain how $\tilde{M}$ simulates one step of M. First, it reads the symbol in the cell 0, which contains the content below the heads of M and replaces this symbol with a new one according to the M's transition function. Then for each head i of M separately, $\tilde{M}$ shifts to the left or to the right (depending on the direction in which the ith head of M moves) the ith coordinates of the symbols in a neighborhood of the cell 0. Because of the symmetry we will only explain how this is done if the ith head of M moves to the right, which means that we have to move the content of the ith tape to the left so that the ith head of M will again be above cell 0.

- $\tilde{M}$ finds the smallest j_0 such that R_{j_0} is not i-empty. Note that this is also the smallest j_0 such that L_{i_0} is not i-full. We call the number j_0 the index of this particular shift.
- If we consider only the ith coordinate of the symbols on the tape of $\tilde{M}$, $\tilde{M}$ writes the leftmost non-# symbol of the zone R_{j_0} in the cell 0 and shifts the next leftmost $2^{j_0}-1$ non-# symbols from R_{j_0} into the zones $R_0, R_1 \dots R_{j_0-1}$ filling up exactly half the symbols in each zone. Note that there is exactly room to perform this since all the zones $R_0, R_1 \dots R_{j_0-1}$ were i-empty and indeed $2^{j_0} - 1 = \sum_{\ell=0}^{i_0-1} 2^\ell$.
- $\tilde{M}$ performs the symmetric operation to the left of the cell 0. That is, for ℓ starting from $j_0 - 1$ down to 0, $\tilde{M}$ iteratively moves the $2^{\ell+1}$ ith coordinates of symbols from L_ℓ to fill half the ith coordinates of the symbols in $L_{\ell+1}$. Finally, $\tilde{M}$ writes the ith coordinate of the symbol that was in position 0 before the shift to the ith coordinate of the rightmost symbol in L_0. An example is given in Figure 3.3.
- At the end of the shift, all of the zones $R_0, L_0 \dots R_{j_0-1}, L_{j_0-1}$ are i-half-full, R_{j_0} has 2^{j_0} fewer non-#-symbols and L_{j_0} has 2^{j_0} additional non-# symbols. Thus our invariants are maintained.
- The total cost of performing the shift is proportional to the total size of all the zones involved $(R_0, L_0 \dots R_{j_0}, L_{j_0})$. That is,

$$\mathrm{O}\left(\sum_{\ell=0}^{j_0} 2^{\ell+1}\right) = \mathrm{O}(2^{j_0})$$

  operations.

We are going to bound the total time of the simulation, as it is done in amortized analyses. After performing a shift of the ith tape with index j, the zones $R_0, L_0 \dots R_{j-1}, L_{j-1}$ are i-half-full which means that it will take at least 2^{j-1} right shifts before the zones $R_0, R_1 \dots R_{j-1}$ become i-full and it will take at least 2^{j-1} left shifts before the zones $R_0, R_1 \dots R_{j-1}$ become i-empty. In any case, once $\tilde{M}$ performs a shift of the ith tape with index j, the next 2^{j-1} shifts of the ith tape will have index less than j. This means that for the tape i of M, at most a $1/2^i$ fraction of the total number of shifts have index i. Because $\tilde{M}$ on an input of length n performs at most $T(n)$ shifts for each tape and because the highest possible index of a shift is at most $\lceil \log T(n) \rceil$, the total number of steps of $\tilde{M}$ on an input of length n is

$$\mathrm{O}\left(n + k \sum_{j=1}^{\lceil \log T(n) \rceil} \frac{T(n)}{2^{j-1}} 2^j\right) = \mathrm{O}\left(T(n) \log T(n)\right),$$

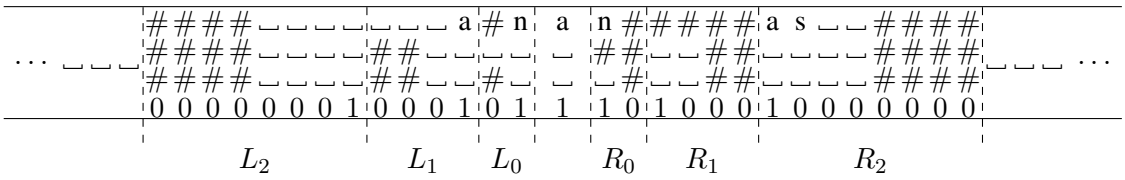


Figure 3.3: If M on input *ananas* would not change the content of its tapes in the first two steps and it would only move the second head one cell to the right and the first head two cells to the right, $\tilde{M}$ would have to make two left shifts of the content of the first tape and one left shift of the content of the second tape to simulate M. In this figure we can see $\tilde{M}$'s input tape after simulating the two steps of M.

which completes the proof. □

Finally, the next corollary tells us that when reducing a multi-tape DTM to a 2-tape DTM, we do not need to increase the tape alphabet to get the same performance.

Corollary 3.3.13. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape DTM M, then it is also decided by some 2-tape DTM $\tilde{M}$ with the tape alphabet $\Sigma \cup \{\llcorner\!\lrcorner\}$ in time $\mathrm{O}\left(T(n) \log T(n)\right)$.*

Proof. Proposition 3.3.12 gives us a two-tape Turing machine that decides L and runs in time $\mathrm{O}\left(T(n) \log T(n)\right)$ and Proposition 3.3.3 assures that this Turing machine can have the tape alphabet $\Sigma \cup \{\llcorner\!\lrcorner\}$. □

3.3.4 Non-Determinism and Determinism

In this short section we compare the non-deterministic Turing machines with the deterministic ones. While the deterministic Turing machines can actually be satisfactory built (we only have to use finite, but very long tapes), we do not know how to implement non-determinism efficiently. What is more, we do not know how to prove (if true) that non-determinism cannot be effectively implemented deterministically. This problem is also the main issue of the P versus NP problem.

However, non-determinism does not help us with deciding new languages, as the following proposition tells.

Proposition 3.3.14. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a one-tape NTM M, then it is also decided by some 3-tape DTM in time $2^{\mathrm{O}(T(n))}$.*

Proof. Let us describe a 3-tape DTM $\tilde{M}$ that decides L in time $2^{\mathrm{O}(T(n))}$. The input tape of $\tilde{M}$ will be reserved for storing the input and $\tilde{M}$ will never change the content of that tape. The second tape of $\tilde{M}$ will be the simulation tape and $\tilde{M}$ will simulate all computations on a particular input of M on it, one by one. The third tape will help $\tilde{M}$ to remember what computations of M were already simulated.

If $M = (Q, \Sigma, \Gamma, \llcorner\!\lrcorner, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$, then the elements of $\Gamma \times Q \times \{-1, 1\}$ represent all potential moves of M in each step. Let us fix some linear ordering of the elements from $\Gamma \times Q \times \{-1, 1\}$. This means that we have a canonical ordering of non-deterministic choices of M. When $\tilde{M}$ will be simulating a step of M, it will first simulate the non-deterministic choice with the lowest number, that still leads to a computation that has not yet been considered.

On an input w of length n, $\tilde{M}$ first simulates the first step of M on w, using the first possible non-deterministic choice and it writes all the possible choices for the first step on the third tape in the canonical ordering. Then it simulates the second step of M using the first possible non-deterministic choice and it writes all the possible choices for this step (step one is fixed) on the third tape. Then it simulates the third step of M using the first possible non-deterministic choice and it writes all the possible choices for this step (steps one and two are fixed) on the third tape. It continues this way until M halts. If M accepts, $\tilde{M}$ also accepts, else M clears the second tape and prepares the third tape the following way. It first locates the last step of M where there was more than one non-deterministic option to choose. Then it deletes all the information about the steps that followed this step and it deletes the first option for this step (the one that was simulated). If such a step does not exist, i.e., if for each step there was only one option on the third tape, $\tilde{M}$ rejects. Then $\tilde{M}$ simulates M on w again, not using the transition function of M but using the first choices for each step as written on the third tape. When all the steps that have some information on the third tape have been simulated, $\tilde{M}$ continues to simulate M using its transition function, always choosing the first choice and writing all the possible non-deterministic choices of M to the third tape. If M accepts, $\tilde{M}$ also accepts, else M clears the second tape and prepares the third tape as before. Then it simulates M on w again using the third tape ...

It is clear that $\tilde{M}$ simulates all the computations of M on w this way and it accepts w if and only if there exists an accepting computation of M on w. Else it rejects w. For the simulation of one computation of M on w, $\tilde{M}$ needs $\mathrm{O}(T(n))$ steps. However, it needs to simulate $2^{\mathrm{O}(T(n))}$ computations, hence $\tilde{M}$ runs in time $\mathrm{O}\left(T(n)\right)2^{\mathrm{O}(T(n))} = 2^{\mathrm{O}(T(n))}$. □

Now we can see that all our models of Turing machines decide the same languages. We state it as a corollary that the weakest model, namely the one-tape DTMs, can decide the same languages as the strongest model, the multi-tape NTMs.

Corollary 3.3.15. *If a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NTM M, then it is also decided by some one-tape DTM in time $2^{\mathrm{O}(T(n)^2)}$.*

Proof. By Proposition 3.3.8, L is decided by some one-tape NTM in time $\mathrm{O}(T(n)^2)$, by Proposition 3.3.14 L is decided by a 3-tape DTM in time $2^{\mathrm{O}(T(n)^2)}$ and again by Proposition 3.3.8 L is decided by a one-tape DTM in time $2^{\mathrm{O}(T(n)^2)}$. □

3.3.5 Reducing the Number of Non-Deterministic Options

We discussed in the previous section that we do not know how to efficiently simulate NTMs by DTMs. The difference between the definition of a DTM and an NTM is only in that the non-deterministic Turing machine can have more than one choice in each step. But what if we limit these options to at most two? We say that an NTM M is a *two-choice* NTM if in each step it has at most two possible non-deterministic choices. In the next proposition we show that a two-choice NTM is just as powerful as an NTM.

Proposition 3.3.16. *For $k \geq 1$, if a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a k-tape NTM M, then it is also decided by some k-tape two-choice NTM in time $\mathrm{O}(T(n))$.*

Proof. Let $\tilde{M}$ be the following k-tape two-choice NTM. It computes exactly as M, only that, for each step of M, it makes a constant number of additional steps during which nothing changes on the tapes. The purpose of these steps is to choose a non-deterministic choice of M. If in one step

M has r non-deterministic choices, it is enough that $\tilde{M}$ uses $\mathrm{O}(\log r)$ non-deterministic steps to consider all these choices. When $\tilde{M}$ selects a choice of M, it goes simulating a next step of M. Because r depends only on M it holds $r = \mathrm{O}(1)$, thus $\tilde{M}$ accepts L in time $\mathrm{O}(T(n))$. □

3.4 Complexity Classes

There are several possible ways to measure how hard it is to decide a language. Because we focus on time complexity in this dissertation, we will divide the languages into time related complexity classes, hence we will measure how fast they can be decided by Turing machines. Our classes will be defined using multi-tape Turing machines because they are easier to construct. Because of the linear speedup discussed in Section 3.3.2 it makes sense to define the classes using the big O notation which hides a constant linear factor.

- For a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we define the complexity class $\mathsf{DTIME}(T(n))$ as the class of all languages $L \subseteq \Sigma^*$ that are decided by some multi-tape DTM in time $\mathrm{O}(T(n))$.
- For a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we define the complexity class $\mathsf{NTIME}(T(n))$ as the class of all languages $L \subseteq \Sigma^*$ that are decided by some multi-tape NTM in time $\mathrm{O}(T(n))$.
- Define the complexity class $\mathsf{P} = \bigcup \mathsf{DTIME}(p(n))$, where the union is over all polynomials $p : \mathbb{N} \to \mathbb{R}_{>0}$. Note that P is the class of languages decidable in polynomial time by a multi-tape DTM. Equivalently, by Proposition 3.3.8 P is the class of languages decided in polynomial time by a one-tape DTM.
- Define the complexity class $\mathsf{NP} = \bigcup \mathsf{NTIME}(p(n))$, where the union is over all polynomials $p : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Note that NP is the class of languages decided in polynomial time by a multi-tape NTM. Equivalently, by Proposition 3.3.8 NP is the class of languages decided in polynomial time by a one-tape NTM.
- Define the complexity class $\mathsf{co\text{-}NP}$ as the class of languages L such that $\Sigma^* \backslash L \in \mathsf{NP}$. That is, $\mathsf{co\text{-}NP}$ is the class of languages such that their complements are decided in polynomial time by an NTM.
- Define the complexity class $\mathsf{EXP} = \bigcup \mathsf{DTIME}(2^{p(n)})$, where the union is over all polynomials p. Note that EXP is the class of languages decided in exponential time by a multi-tape DTM and hence also by a one-tape DTM.
- Define the complexity class $\mathsf{NEXP} = \bigcup \mathsf{NTIME}(2^{p(n)})$, where the union is over all polynomials p. Note that NEXP is the class of languages decided in exponential time by a multi-tape NTM and hence also by a one-tape NTM.
- We say that a language $L \subseteq \Sigma^*$ is *decidable* if there exists a Turing machine that decides it. In other words, the class of decidable languages (also known as the class of recursive languages in the literature) is $\bigcup \mathsf{NTIME}(T(n))$ where the union is over all functions $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

With these classes we set up some hierarchies of decidable languages, like:

$$\mathsf{DTIME}(n) \subseteq \mathsf{DTIME}(n^2) \subseteq \mathsf{DTIME}(n^{10}) \subseteq \mathsf{P} \subseteq \mathsf{DTIME}(2^n) \subseteq \mathsf{EXP}$$

and

$$\mathsf{NTIME}(n) \subseteq \mathsf{NTIME}(n^2) \subseteq \mathsf{NTIME}(n^{10}) \subseteq \mathsf{NP} \subseteq \mathsf{NTIME}(2^n) \subseteq \mathsf{NEXP}.$$

The inclusions are clearly true and we will prove in Theorem 4.2.1 and in Theorem 4.2.3 that they are all strict. However, while we can say

$$\mathsf{DTIME}(n) \subseteq \mathsf{NTIME}(n), \qquad \mathsf{DTIME}(n^2) \subseteq \mathsf{NTIME}(n^2) \qquad \ldots \qquad \mathsf{EXP} \subseteq \mathsf{NEXP},$$

we do not know whether these inclusions are strict or not, except for the first one which was proven strict by Paul, Pippenger, Szemerédi and Trotter [25]. As already discussed in the introduction, the question whether P equals NP is a major unsolved problem in computational complexity theory and a prize of one million US dollars is offered for the solution [23]. By Corollary 3.3.15 it holds $\mathsf{NP} \subseteq \mathsf{EXP}$, however this inclusion is also not known to be strict.

3.4.1 Complexity Classes of Decision Problems

In Section 2.1.4 we made a distinction between languages and decision problems. While a language is a fixed subset $L \subseteq \Sigma^*$, a decision problem is a more general notion, defined by a set of instances together with a subset of YES instances (no encoding is needed).

While we defined complexity classes only for languages, the same complexity classes are used also for decision problems. When we say that a decision problem is in some complexity class, say $\mathsf{DTIME}(n^2)$, we have some natural encoding for the problem in mind and if L is the corresponding language, we are actually claiming $L \in \mathsf{DTIME}(n^2)$. For several complexity classes like P, NP or co-NP all natural encodings are equivalent because we can change between natural encodings in polynomial time.

3.4.2 The Complexity of Regular Languages

In the following proposition we show how hard, relative to the just defined complexity classes, are regular languages. Note that we did not define regular languages as a time-related complexity class. However, in Section 5.1.2 we show that we could do so, namely regular languages are exactly the languages decided by linear-time one-tape NTMs.

Proposition 3.4.1. *The class of regular languages strictly contains the class* $\mathsf{NTIME}(1)$ *and is strictly contained in the class* $\mathsf{DTIME}(n)$.

Proof. We showed in Lemma 3.3.1 that if a language L is in $\mathsf{NTIME}(1)$, then L is regular. Now if we take the language L_0 of binary strings that end with a 0, L_0 is regular because it is given by the regular expression $\{0,1\}^*\{0\}$ (see Theorem 2.2.7). However, any Turing machine that decides L_0 has to read the whole input up to the last symbol, thus it does not run in constant time. Hence, $L_0 \notin \mathsf{NTIME}(1)$, which implies that the class of regular languages strictly contains the class $\mathsf{NTIME}(1)$.

The class of regular languages is contained in the class $\mathsf{DTIME}(n)$ because finite automata are just a restricted version of one-tape DTMs. To show that the inclusion is strict, let us consider the problem Palindrome. In Section 2.2.1 we showed that this problem is not regular, however it can easily be solved in linear time by a multi-tape DTM. □

3.4.3 Complexity of Computing Functions

We defined Turing machines like if their only purpose would be to solve decision problems. However, while solving a problem, one often has to compute some functions during the computations and here we define the corresponding notions.

We say that a function $f : \Sigma^* \to \Sigma^*$ is *computable*, if there exists a DTM M that on any input $w \in \Sigma^*$ halts in an accepting state with $f(w)$ written on the input tape. This means that when M halts, the input tape is of the form

$$\ldots \sqcup \sqcup \sqcup \sqcup \sqcup \boxed{\text{f(w)}} \sqcup \sqcup \sqcup \sqcup \sqcup \ldots.$$

If M runs in time $T(n)$, then we say that M *computes the function* f *in time* $T(n)$.

Example. Let us show that we can convert the binary representation of a positive integer x to unary and vice versa in time $\mathrm{O}(x)$. This is equivalent to claiming that we can compute the function

$$f : \{0,1\}^* \to \{0,1\}^* \quad f : w \mapsto \begin{cases} 1^x & \text{if } w \text{ is the binary representation of a positive integer } x \\ w & \text{else} \end{cases}$$

in time $\mathrm{O}(|f(w)|)$ (i.e., linear in the length of the output) and the function

$$g : \{0,1\}^* \to \{0,1\}^* \qquad g : w \mapsto |w| \text{ (in binary)}$$

in time $\mathrm{O}(n)$ (i.e., linear in the length of the input). To compute f we construct the following two-tape DTM M_f. On an input w, M_f first verifies whether w starts with the symbol 1. If not, it accepts, else let x be the integer represented as w in binary. M_f copies x on the second tape in binary leaving the first tape blank and then it starts subtracting 1 from it (using only the second tape), each time writing an additional 1 on the first tape. It halts when there is only the symbol 0 left on the second tape. Clearly, M_f computes the function f.

We are going to bound the total time of the computation, as it is done in amortized analyses. To subtract 1 from x, M_f needs $\mathrm{O}(\log x)$ steps (which is the length of the representation of x in binary). However, usually it needs just $\mathrm{O}(1)$ steps. To be more precise, the length of the binary representation of x is $\lfloor \log x \rfloor + 1$ and the $(\lfloor \log x \rfloor + 1)$st digit is "changed" exactly once (when all the other digits right of it are 0). The $\lfloor \log x \rfloor$th digit gets changed at most twice, the $(\lfloor \log x \rfloor - 1)$st digit gets changed at most 4 times ... Because M_f needs $\mathrm{O}(i)$ steps to change the ith digit, it makes

$$\sum_{i=1}^{\lfloor \log x \rfloor + 1} 2^{\lfloor \log x \rfloor + 1 - i} \, \mathrm{O}(i)$$

steps overall. Considering that the sum

$$\sum_{i=1}^{\infty} 2^{1-i} i$$

converges, we get that

$$\sum_{i=1}^{\lfloor \log xn \rfloor + 1} 2^{\lfloor \log x \rfloor + 1 - i} \, \mathrm{O}(i) = \mathrm{O}(2^{\lfloor \log x \rfloor}),$$

which is $\mathrm{O}(x)$. Hence, M_f computes f in time that is linear in the length of the output.

To show how to compute g in linear time, we construct a two-tape DTM M_g that on an input w first copies w on the second tape, leaving only one 0 on the input tape. Then it increases the number (in binary) on the input tape one by one, each time reducing the number of non-blank symbols on the second tape by one. M_g halts when it runs out of non-blank symbols on the second tape. Using the same analysis as for M_f, it is clear that M_g runs in linear time. ■

3.5 The Church-Turing Thesis

While in Section 3.3 we compared different models of Turing machines, in this section we discuss how powerful the Turing machines are compared to other computational models, like models of our computers and how the time classes defined above reflect the real world.

The *Church-Turing thesis* tells us that decidable problems are a superset of the decision problems that can be solved in the real world. The thesis is the following [2, Chapter 1]:

> *Every physically realizable computation device*[2]*—whether it is based on silicon, DNA, neurons or some other alien technology—can be simulated by a Turing machine.*

The thesis dates back for more than 70 years and it has still not been disproved. Because of its informal nature it cannot be viewed as a theorem that can be proven, rather than a belief about the nature of the world as we currently understand it.

The thesis is not just explanatory, it is also very useful. It tells us that if we can explain an algorithm that solves some problem, then we can also construct a Turing machine that solves the same problem. A stronger statement known as the *strong version of the Church-Turing thesis* [2, Chapter 1] has also been proposed:

> *Every physically realizable computation device can be simulated by a deterministic Turing machine with polynomial overhead, i.e., t steps on the device correspond to at most t^c steps on the DTM, where c is a constant that depends upon the model.*

Unlike the standard Church-Turing thesis, its strong form is somewhat controversial, in particular because of models such as quantum computers [2, Chapter 10], which do not appear to be efficiently simulatable on DTMs. However, it is still unclear if reasonably big quantum computers can be physically realized.

What supports the strong version of the Church-Turing thesis most is that idealized models of our computers can be simulated by Turing machines with polynomial overhead [24, Chapter 2.8]. This fact is useful, for example, when proving that some problem is in P because we do not need to describe a polynomial-time Turing machine that solves the problem, it is enough to just describe an algorithm that would run in polynomial time on a computer and would solve the problem.

It is worth mentioning that Turing machines are somewhat stronger than real-world computers because the latter have a limited amount of memory.

3.6 Encoding Turing Machines

There are several good ways of how to encode Turing machines. We present here the encoding from the paper [11] of the author. Let a *code* of a k-tape NTM $M = (Q, \Sigma, \Gamma, \llcorner, \delta, q_{\text{o}}, q_{\text{ACC}}, q_{\text{REJ}})$ be a code of a $|Q| \times |Q|$ matrix A, where $A[i, j]$ is a (possibly empty) list of all the triples

$$\big((a_1, a_2 \dots a_k), (b_1, b_2 \dots b_k), (d_1, d_2 \dots d_k)\big) \in \Gamma^k \times \Gamma^k \times \{-1, 0, 1\}^k$$

such that M can come in one step from the state q_i to the state q_j replacing the symbols $a_1, a_2 \dots a_k$ below the heads by the symbols $b_1, b_2 \dots b_k$ (respectively) and moving the heads in the directions $d_1, d_2 \dots d_k$. In other words, $A[i, j]$ is a list of all the triples

$$\big((a_1, a_2 \dots a_k), (b_1, b_2 \dots b_k), (d_1, d_2 \dots d_k)\big) \in \Gamma^k \times \Gamma^k \times \{-1, 0, 1\}^k$$

[2] We assume that the computation device takes a string from Σ^* as input and it returns an output from Σ^*. This assumption is shown important in [14].

such that

$$\big((b_1, b_2 \dots b_k), q_j, (d_1, d_2 \dots d_k)\big) \in \delta\big((a_1, a_2 \dots a_k), q_i\big).$$

We assume that the indices i and j range from 0 to $|Q| - 1$ and that the index 0 corresponds to the starting state, the index $|Q|-2$ corresponds to the accepting state and the index $|Q|-1$ corresponds to the rejecting state. We also assume that each symbol of Σ as well as the blank symbol have a universal unique code over Σ.

It is clear that a code of M is of length $\mathrm{O}(q^2 k 3^k |\Gamma|^{2k} \log |\Gamma|)$. However, for applications that follow we will need arbitrary long codes, hence we define a *padded code of* M as a code of M, padded in front by any number of 0s followed by a redundant 1. Thus the padded code of an NTM can be arbitrarily long. Note that given a padded code of M, the code of M can be constructed in linear time.

An interesting property of our encoding is that we can compute compositions of Turing machines in linear time, as can be seen in Figure 3.4. The *composition* of NTMs M_1 and M_2 is the NTM that starts computing as M_1, but has the starting state of M_2 instead of M_1's accepting state. When the starting state of M_2 is reached, it computes as M_2. If M_1 rejects, it rejects.

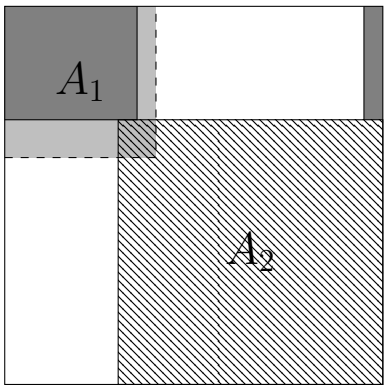


Figure 3.4: The code of a composition of two Turing machines. Suppose that we want to compute the code of a composition of Turing machines M_1 and M_2. Let A_1 and A_2 be the corresponding matrices that were used to encode M_1 and M_2. Then we can erase the last two lines of the matrix A_1 (they correspond to the halting states of M_1) and adjust A_2 "below" A_1 as shown in the figure. Note that the column of the accepting state of M_1 coincides with the column of the starting state of M_2. The last column of A_1 that corresponds to the rejecting state of M_1 is flushed to the right. To compute the code of the composition of two Turing machines, we have to compute the code of this matrix, which can be done in linear time given the codes of A_1 and A_2.

3.6.1 Universal Turing Machine

Now that we know how to encode Turing machines, a natural thing to do is to consider an algorithm to simulate them. The next two propositions describe two such algorithms.

Proposition 3.6.1. *There exists a 3-tape NTM U (called the* universal Turing machine*) that given a (code of a) multi-tape NTM M together with an input w for M, it simulates M on w and returns the same as M on w. If M runs in time $T(n)$, then U on inputs (M, w) makes at most $C_M\big(T(|w|) + |w|\big)$ steps for some constant C_M depending only on M.*

Proof. The universal Turing machine U computes as follows. On input (M, w), where M is a multi-tape NTM that runs in time $T(n)$, U first writes w on the second tape, leaving only the description of M on the first tape. Then using the input tape and the third tape it transforms the code of M into a code of a 2-tape NTM $\tilde{M}$ that simulates M on two tapes in time $\mathrm{O}(T(n))$ and uses the tape alphabet $\Sigma \cup \{\_\}$. This can be done by Corollary 3.3.11 because the corollary was proven constructively. After this initial work, U has the code of $\tilde{M}$ on the input tape, w on the second tape and the third tape is empty. Now U simulates $\tilde{M}$ step by step in the following way: it keeps track of which state $\tilde{M}$ currently is in the first tape while having the content of the second and the third tape as well as the positions of the heads on the second and the third tape exactly the same as $\tilde{M}$. To simulate a step of $\tilde{M}$, U does some computation on the input tape to figure out a next step of $\tilde{M}$ and then it simulates it. Hence, U returns the same as $\tilde{M}$ does on the input w which is the same as M on the input w.

If M is fixed, then U makes $\mathrm{O}(|w|)$ steps before it starts to simulate $\tilde{M}$ and then for each step of $\tilde{M}$ it makes a constant number (dependant only on M, not on w) of steps. Because $\tilde{M}$ runs in time $\mathrm{O}(T(n))$, U makes $\mathrm{O}\left(T(|w|) + |w|\right)$ steps. □

For deterministic Turing machines, the following proposition applies.

Proposition 3.6.2. *There exists a 3-tape DTM U (called the* deterministic universal Turing machine*) that given a (code of a) multi-tape DTM M together with an input w for M, it simulates M on w and returns the same as M on w. If M runs in time $T(n)$, then U on inputs (M, w) makes at most $C_M\big(T(|w|)\log T(|w|) + |w|\big)$ steps for some constant C_M depending only on M.*

Proof. The proof is the same as for Proposition 3.6.1, only that Corollary 3.3.13 is used instead of Corollary 3.3.11. □

3.7 Classes NP and co-NP

Let us give another characterization of the class NP.

Proposition 3.7.1. *A language $L \subseteq \Sigma^*$ is in* NP *if and only if there exist positive integers k, D and a polynomial-time DTM M such that for every $w \in \Sigma^*$*

$$w \in L \iff \exists u \in \Sigma^* : \big(|u| \leq |w|^k + D\big) \text{ and } M \text{ accepts the input } (w, u).$$

If $L \in \mathsf{NP}$ and we have k, D and M as in the proposition, then for any $w \in L$ and $u \in \Sigma^*$ such that $\big(|u| \leq |w|^k + D\big)$ and M accepts the input (w, u), we call u a *certificate* for w. Before we prove the proposition, let us give an example of use.

Example. Let us show that the decision problem Hamiltonian Cycle is in NP. First, we will show this by definition of NP and then by using Proposition 3.7.1.

To prove that Hamiltonian Cycle $\in$ NP by definition, we have to construct a polynomial-time NTM $\tilde{M}$ that decides the problem Hamiltonian Cycle. What $\tilde{M}$ does is the following: on an input w which represents a graph on n vertices, it non-deterministically chooses a sequence of n vertices and then it deterministically verifies whether this sequence forms a Hamiltonian cycle. If so, it accepts, else it rejects. We can make $\tilde{M}$ to run in polynomial time, which implies that Hamiltonian Cycle $\in$ NP.

To prove that HAMILTONIAN CYCLE $\in$ NP using the new characterization from Proposition 3.7.1, we just need to give the appropriate certificate for each graph that admits a Hamiltonian cycle: the easiest one is just the cycle itself. Now let us show why this is good enough. Let us take a multi-tape DTM M that given a code of a graph G together with a code of a sequence u of its vertices, decides whether u is a Hamiltonian cycle of G. We can make M to run in polynomial time. If we take $k = D = 1$ and consider that the code of a Hamiltonian cycle of G is not longer than the code of G (assuming a natural encoding), it is clear that

$$w \text{ encodes a Hamiltonian graph}$$

$$\iff$$

$$\exists u \in \Sigma^* : \big(|u| \leq |w| + 1\big) \text{ and } M \text{ accepts the input } (w, u),$$

hence HAMILTONIAN CYCLE $\in$ NP. ■

Now let us prove the proposition.

Proof. If $L \in \mathsf{NP}$ then there exists a one-tape NTM $\tilde{M}$ that decides L in polynomial time. Let M be a multi-tape DTM that given an input (w, u) where $w, u \in \Sigma^*$, it simulates exactly one computation of $\tilde{M}$ on the input w following non-deterministic choices encoded in u. If u does not encode a sequence of non-deterministic choices, then M rejects. How this can be done so that M runs in polynomial time is evident in the proof of Proposition 3.3.14. Let us define integers k and D so that, for all n, it will hold:

$$n^k + D \geq \text{the length of a longest code of a sequence of choices in a computation of } \tilde{M} \text{ on inputs of length } n.$$

Note that such k and D exist because $\tilde{M}$ runs in polynomial time and thus makes only polynomially many non-deterministic choices. Because $\tilde{M}$ accepts w if and only if there is an accepting computation of $\tilde{M}$ on w, we have

$$w \in L \iff \exists u \in \Sigma^* : \big(|u| \leq |w|^k + D\big) \text{ and } M \text{ accepts the input } (w, u).$$

To show the *if* part of the proposition, let L, M, k and D be as in the proposition and let an NTM $\tilde{M}$ be defined as follows. On an input w of length n, $\tilde{M}$ first computes $n^k + D$ and it non-deterministically chooses a string $u \in \Sigma^*$ of length at most $n^k + D$. Then it simulates M on (w, u) and it accepts if and only if M accepts. Clearly, $\tilde{M}$ runs in polynomial time and decides L. □

3.7.1 Reductions and Complete problems

If we compare the classes P, NP and co-NP, we get the Figure 3.5. Although we do not know whether P = NP, we can say something about which problems in NP are hard if P $\neq$ NP. Such problems are called NP-complete.

Reductions

A language $L_1 \subseteq \Sigma^*$ is *reducible* to a language $L_2 \subseteq \Sigma^*$ if there exists a computable function $f : \Sigma^* \to \Sigma^*$ called a *reduction* such that for every $w \in \Sigma^*$,

$$w \in L_1 \iff f(w) \in L_2.$$

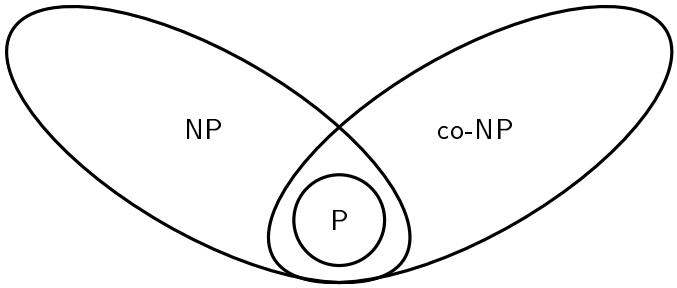


Figure 3.5: The complexity classes P, NP and co-NP. While $\mathsf{P} \subseteq \mathsf{NP} \cap \text{co-}\mathsf{NP}$, we do not know whether the inclusion is strict. We also do not know whether $\mathsf{NP} \subseteq \text{co-}\mathsf{NP}$ or $\text{co-}\mathsf{NP} \subseteq \mathsf{NP}$ although it is widely believed that none of these two options are true.

In the literature there are more types of reductions. However, we will only use the type of reductions we defined, which are also known as Karp reductions or multi-one reductions in the literature.

If f is computable in polynomial time by a DTM, we say that f is a *polynomial-time reduction* and that L_1 is *polynomial-time reducible* to L_2.

Complete Problems

A language $L \subseteq \mathsf{NP}$ is called NP-complete if

1. $L \in \mathsf{NP}$ and
2. Each language $\tilde{L} \in \mathsf{NP}$ is polynomial-time reducible to L.

A decision problem is called NP-complete if its corresponding language is NP-complete. Note that, at this point, we did not prove that NP-complete problems actually exist. This will be done later in Proposition 3.7.4.

The following proposition tells that, if $\mathsf{P} \neq \mathsf{NP}$, NP-complete languages cannot be solved in deterministic polynomial time.

Proposition 3.7.2. *If some* NP-complete *language can be decided in polynomial time by a DTM, then* $\mathsf{P} = \mathsf{NP}$.

Proof. Let L be an arbitrary language in NP and suppose that an NP-complete language L_0 can be solved in polynomial time by a DTM M_0. Because L_0 is NP-complete there exists a polynomial-time reduction f that reduces L to L_0. Let us consider the following deterministic algorithm to decide L:

On an input w, compute $f(w)$ and then run M_0 on $f(w)$.

Because f runs in polynomial time, the length of $f(w)$ is polynomial in the length of w and because M_0 also runs in polynomial time, this algorithm decides L in deterministic polynomial time. Because $L \in \mathsf{NP}$ was arbitrary, it follows $\mathsf{P} = \mathsf{NP}$. □

A language $L \in \text{co-}\mathsf{NP}$ is called co-NP-complete if

1. $L \in \text{co-}\mathsf{NP}$ and
2. Each language $\tilde{L} \in \text{co-}\mathsf{NP}$ is polynomial-time reducible to L.

Recall that a language L is in NP if and only if its complement $\overline{L}$ is in co-NP. The same is true for complete languages.

Proposition 3.7.3. *A language $L \subseteq \Sigma^*$ is* NP-complete *if and only if $\overline{L}$ is* co-NP-complete.

Proof. Let L be NP-complete and let L' be an arbitrary language in co-NP. By definition, $\overline{L} \in$ co-NP and $\overline{L'} \in$ NP. Because L is NP-complete, there exists a polynomial-time reduction f of $\overline{L'}$ to L. It is easy to see that f is also a reduction of L' to $\overline{L}$, which proves that $\overline{L}$ is co-NP-complete. The proof of the *if* part of the proposition is symmetric. □

A possible (and widely believed) relation between the classes P, NP, co-NP and the complete problems is shown in Figure 3.6.

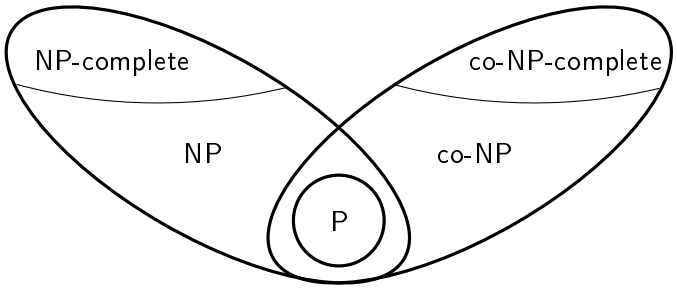


Figure 3.6: The complexity classes P, NP and co-NP together with complete problems. Note that if there would be an NP-complete problem in P, the whole figure would collapse to P. Because of the symmetry, the same would happen if there would be a co-NP-complete problem in P. We leave to the reader to show that if there would be a problem that would be NP-complete and co-NP-complete, then NP = co-NP.

We still did not give any example of a complete problem. However, an NP-complete language is not hard to find, especially if we consider that all languages in NP are given by some NTM. Let us define a decision problem NP-HALT as:

Given an NTM M and an input w for M, does M make more than $2|w| + 1$ steps on some computation on the input w?

Proposition 3.7.4. *The problem* NP-HALT *is* NP-complete.

Proof. The problem NP-HALT is in NP because its every YES-instance (M, w), where w is an input for an NTM M, has a certificate: a sequence of first $2|w| + 2$ non-deterministic choices of M on input w for which M makes $2|w| + 2$ steps or more.

To show that NP-HALT is NP-complete, let L be some language in NP. By definition of NP there exists an NTM M that decides L in non-deterministic time $p(n)$ for some polynomial p. Let us define an NTM $\tilde{M}$ that given an input of the form 0^i1w where $w \in \Sigma^*$ and $i \in \mathbb{N}$, it first erases 0^i1 and then it simulates M on w. The only difference with M is that instead of going to the accept state, $\tilde{M}$ starts an infinite loop (e.g. its heads starts to move right forever). Note that

$$\begin{aligned} M \text{ accepts } w &\iff \tilde{M} \text{ on input } 0^{p(|w|)}1w \text{ starts an infinite loop on some computation} \\ &\iff \tilde{M} \text{ on input } 0^{p(|w|)}1w \text{ makes more than } 2\left|0^{p(|w|)}1w\right| + 1 \\ &\qquad \text{steps on some computation.} \\ &\iff (\tilde{M}, 0^{p(|w|)}1w) \in \text{NP-HALT}. \end{aligned}$$

Define a function $f : \Sigma^* \to \Sigma^*$ as

$$w \mapsto \text{a code of the pair } (\tilde{M}, 0^{p(|w|)}1w).$$

It is clear that f is computable in polynomial time. Hence, f is a polynomial-time reduction of L to NP-HALT, which proves the proposition. □

Later in Section 6.2 we will give more examples of NP-complete problems (actually co-NP-complete problems). While they will arise from natural (theoretic) questions, they will speak about Turing machines, which are quite abstract. However, there are very many more natural practical NP-complete problems, one of them being HAMILTONIAN CYCLE. In this dissertation, we will not prove that this problem is NP-complete, the reader can find the proof for that in e.g. Garey and Johnson [12], where also several other natural NP-complete problems are described.

Chapter 4

Diagonalization and Relativization

Diagonalization is a well known proof method. In this chapter we use it to prove that some explicit problems are not decidable and to prove separation of several complexity classes. In Section 4.3 we define a new type of Turing machines, namely oracle Turing machines, and prove that the separation results in this chapter hold also for oracle Turing machines with a fixed oracle. Such results are said to be *relativizing*. We finish the chapter with Theorem 4.3.3 which tells that if we will ever solve the P versus NP question, the result will not relativize.

We begin this section with examples of two famous proofs that use diagonalization, none of which is closely related to computer science. We present them to show the main idea of diagonalization: observing a diagonal.

Example. For the first example we take the Cantor's proof that there exists no surjection of $\mathbb{N}$ into the positive real numbers which proves that the set $\mathbb{R}_{>0}$ has "more" elements than $\mathbb{N}$. Suppose for the purpose of contradiction that a surjective function $f : \mathbb{N} \to \mathbb{R}_{>0}$ exists. This allows us to list the real numbers $f(0), f(1), f(2) \ldots$ using their decimal representation one below the other so that the decimal points are aligned. We may assume that all numbers have infinitely many decimals, some of them having last decimals equal to zero.

$$
\begin{array}{rl}
x_{n_1,1}x_{n_1-1,1}x_{n_1-2,1}\cdots x_{2,1}x_{1,1} \cdot & \mathbf{y_{1,1}}y_{2,1}y_{3,1}\cdots \\
x_{n_2,2}x_{n_2-1,2}\cdots x_{2,2}x_{1,2} \cdot & y_{1,2}\mathbf{y_{2,2}}y_{3,2}\cdots \\
\cdots x_{2,3}x_{1,3} \cdot & y_{1,3}y_{2,3}\mathbf{y_{3,3}}\cdots \\
\cdots x_{2,4}x_{1,4} \cdot & y_{1,4}y_{2,4}y_{3,4}\mathbf{y_{4,4}}\cdots \\
\cdots \quad . \quad \cdots & \ddots \\
\cdots x_{2,i}x_{1,i} \cdot & y_{1,i}y_{2_i} \quad \cdots \quad \mathbf{y_{i,i}}\cdots \\
\cdots \quad . \quad \cdots &
\end{array}
$$

Now define the following real number $r = 0.\tilde{y}_{1,1}\tilde{y}_{2,2}\tilde{y}_{2,2}\tilde{y}_{2,2}\cdots$, where $\tilde{y}_{i,i} = y_{i,i}+5 \mod 10$. Because r is nowhere in the above list, f could not be surjective. ■

Example. For the second example we take Russell's paradox. Not to go into formal logic, let us observe the following naive description of what a set is:

Everything that can be written as $S = \{x; \varphi(x)\}$, where φ is some formula that can be either true or false, is a set. Additionally, for each object x, if $\varphi(x)$, then we write $x \in S$ and we say that x is an element of S.

We claim that such a description of a set is not appropriate. For each ordered pair of sets, the first set is either an element of the second or it is not (see Table 4.1). Now the following set has its diagonal entry "negated":

$$S_R = \{S;\ S \notin S\}.$$

We see that S_R has $\in$ on the diagonal of Table 4.1 if and only if $S_R \notin S_R$, which is a contradiction. Hence, one has to be more careful when defining what a set is. ■

	S_1	S_2	S_3	$\cdots$	S_λ	$\cdots$
S_1	$\notin$	$\in$	$\notin$	$\cdots$	$\in$	$\cdots$
S_2	$\notin$	$\notin$	$\notin$	$\cdots$	$\in$	$\cdots$
S_3	$\notin$	$\in$	$\in$	$\cdots$	$\notin$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
S_λ	$\in$	$\in$	$\in$	$\cdots$	$\notin$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$

Table 4.1: A table of whether the set in the leftmost column is an element of the set in the top row[2]. Although there are more than countably many sets, we started indexing them with integers so that we were able to draw a picture.

4.1 Halting Problems

Since we can encode Turing machines as finite strings (see Section 3.6), there are only countably many of them. However, there are uncountably many languages over Σ^*, which implies that most of them are undecidable. In this section we will present a famous explicit undecidable problem, the halting problem. Because we will present several variations of the halting problem, the title of this section is in plural.

The halting problem D-HALT is the following.

Given a multi-tape DTM M and an input w for M, does M halt on the input w?

There are several variations of the problem, like the more general problem HALT, which is

Given a multi-tape NTM M and an input w for M, does M halt on all computations on the input w?

And the more specific problem D-HALT[1], which is

Given a one-tape DTM M and an input w for M, does M halt on the input w?

[2]In ZFC theory there would be only $\notin$ on the diagonal because of Axiom of regularity.

Analogously we also define the problem HALT^1. In essence these problems are the same, because given a Turing machine of one type, we can transform it into a Turing machine of another type using the transformations described in Section 3.3. Note that these transformations preserve the finiteness of a computation on any input. To prove that all these problems are undecidable, it is enough to prove that the problem D-HALT^1 is undecidable because a Turing machine that would solve any of the problems HALT, D-HALT or HALT^1 would also solve the problem D-HALT^1.

Theorem 4.1.1. *The problem* D-HALT^1 *is undecidable.*

Proof. Because there are countably many one-tape DTMs, we can index them by positive integers. Additionally, for each Turing machine M_j let $\langle M_j \rangle$ denote the lexicographically first of the codes of M_j. Then, for each ordered pair of one-tape DTMs (M_i, M_j), M_i either halts on the input $\langle M_j \rangle$ or it does not halt (see Table 4.2).

	M_1	M_2	M_3	$\cdots$	M_i	$\cdots$
M_1	∞	∞	$\downarrow$	$\cdots$	$\downarrow$	$\cdots$
M_2	∞	$\downarrow$	$\downarrow$	$\cdots$	$\downarrow$	$\cdots$
M_3	$\downarrow$	∞	∞	$\cdots$	∞	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
M_i	$\downarrow$	$\downarrow$	$\downarrow$	$\cdots$	∞	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$

Table 4.2: A table of whether the one-tape DTM in the leftmost column halts ($\downarrow$) or it runs forever (∞) on an input which is the lexicographically first code of the one-tape DTM in the top row.

Suppose for the purpose of contradiction that a one-tape DTM H exists that solves the problem D-HALT^1. Let us observe the diagonal entry in Table 4.2 of the following one-tape DTM M:

> On an input w, M first runs H to verify whether the one-tape DTM M_w encoded as w halts on the input w. If it does, M starts an infinite loop, else it halts.

Thus M halts on the input w if and only if M_w does not halt on the input w. It follows that M halts on the input $\langle M \rangle$ if and only if M does not halt on the input $\langle M \rangle$. This implies that the Turing machine H cannot exist. □

The fact that the halting problem is undecidable has consequences also in real-world applications. Because we can write a program that simulates Turing machines on a given input, the undecidability of D-HALT tells us that there is no automated procedure (i.e., a Turing machine) that would solve the problem:

> Given a code of a program in Java and an input for it, would the program ever terminate if we would run it on that input, or would it run forever?

4.1.1 Proving Undecidability of Problems

In Section 3.7.1 we defined reductions of languages. It is clear that if we find a reduction of an undecidable problem to some other problem, this proves that the other problem is undecidable. We illustrate this method by proving that even a simpler problem than D-HALT^1 is undecidable. Define the problem D-HALT$^1_\varepsilon$ as

Given a one-tape DTM M, does M halt on the empty input ε?

We will use the well known fact that this problem is undecidable a lot in Section 6.1, thus we state it as a lemma.

Lemma 4.1.2. *The problem* D-HALT$^1_\varepsilon$ *is undecidable.*

Proof. Let us give a reduction of the problem D-HALT^1 to the problem D-HALT$^1_\varepsilon$. On an input (M, w) where w is an input of a one-tape DTM M, construct a one-tape DTM $\tilde{M}$ which on the empty input first writes w on the tape and then it computes as M.

It is clear that $\tilde{M}$ halts on input ε if and only if M halts on the input w, which proves that we reduced the problem D-HALT^1 to the problem D-HALT$^1_\varepsilon$. □

4.2 Time Hierarchy Theorems

In this section we prove the following two hierarchies:

$$\mathsf{DTIME}(n) \subsetneq \mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^{10}) \subsetneq \mathsf{P} \subsetneq \mathsf{DTIME}(2^n) \subsetneq \mathsf{EXP}$$

and

$$\mathsf{NTIME}(n) \subsetneq \mathsf{NTIME}(n^2) \subsetneq \mathsf{NTIME}(n^{10}) \subsetneq \mathsf{NP} \subsetneq \mathsf{NTIME}(2^n) \subsetneq \mathsf{NEXP}.$$

We will first introduce the notion of a time constructible function and then we will use these functions to prove the hierarchies.

4.2.1 Time Constructible Functions

A function $f : \mathbb{N} \to \mathbb{N}$ is *time constructible* if the function $\tilde{f} : w \mapsto 1^{f(|w|)}$ is computable by a DTM that runs in time $\mathrm{O}(f(n))$. Note that it does not matter whether $\tilde{f}$ gives the output in unary (as in the case of our definition) or in binary because we can transform binary to unary and vice versa in time that is linear in the length of the unary representation (see Section 3.4.3).

Example. Many simple functions such as n, $n\lfloor \log n \rfloor$, $\lceil n\sqrt{n} \rceil$, n^2, n^{10}, 2^n, $n!$ are clearly time constructible. To compute, say, the value $n\lfloor \log n \rfloor$ given the input of length n, we first transform 1^n into binary in time $\mathrm{O}(n)$. The length of the binary representation of n is $\lfloor \log n \rfloor + 1$. Then we compute in polynomial time (in the length of the binary representation of n which is $\mathrm{O}(\log n)$) the binary representation of $n\lfloor \log n \rfloor$. Hence, the binary representation of $n\lfloor \log n \rfloor$ can be computed in linear time. ■

4.2.2 The Deterministic Time Hierarchy

The following theorem, called also the *deterministic time hierarchy theorem*, tells that DTMs that are allowed to run for a relatively small factor longer, can decide strictly more languages.

Theorem 4.2.1. *For any time constructible function $f : \mathbb{N} \to \mathbb{N}$ such that $n = \mathrm{o}(f(n))$ and for any function $g : \mathbb{N} \to \mathbb{N}$ such that $g(n) \log g(n) = \mathrm{o}(f(n))$, there exists a language*

$$L \in \mathsf{DTIME}(f(n)) \backslash \mathsf{DTIME}(g(n)).$$

Proof. We assume that there are infinitely many padded codes for each DTM, thus each DTM has arbitrary long padded codes. For a padded code w of a DTM M, let M_w denote the DTM M. Let us observe ordered pairs (M_{w_i}, w_j) where w_i and w_j are padded codes of DTMs. For each such a pair, M_{w_i} either accepts the input w_j or it rejects it (see Table 4.3).

	w_1	w_2	w_3	$\cdots$	w_i	$\cdots$
M_{w_1}	**accept**	reject	accept	$\cdots$	accept	$\cdots$
M_{w_2}	reject	**reject**	accept	$\cdots$	accept	$\cdots$
M_{w_3}	accept	accept	**reject**	$\cdots$	accept	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
M_{w_i}	reject	accept	reject	$\cdots$	**reject**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$

Table 4.3: A table of whether the DTM M_{w_i} in the leftmost column accepts an input w_j in the top row, where w_i and w_j are padded codes of DTMs. It suffices for our proof to have only (padded codes of) DTMs that run in time $\mathrm{O}(g(n))$ in this table. Because each DTM has infinitely many padded codes, each DTM that runs in time $\mathrm{O}(g(n))$ is present infinitely many times in the leftmost column of the table.

Let M be the following 4-tape DTM: on an input w, which is a padded code of the DTM M_w, it first computes $1^{f(|w|)}$ and stores it on the fourth tape. Then it computes just as the deterministic universal 3-tape Turing machine from Proposition 3.6.2 on the input (M_w, w), i.e., it efficiently simulates M_w on the input w. For each step that M makes, it erases one symbol 1 from the fourth tape. The fourth tape thus serves as a counter and when there are no more ones on it, M rejects. If the simulation of M_w finished when there was still at least one symbol 1 on the fourth tape, M returns the opposite of what M_w would return, i.e., if M_w rejects w then M accepts, else M rejects. Let L be the language decided by M.

We see that M runs in time $\mathrm{O}(f(n))$, hence $L \in \mathsf{DTIME}(f(n))$. It is enough to prove that $L \notin \mathsf{DTIME}(g(n))$. If $L \in \mathsf{DTIME}(g(n))$, then there exists a DTM $M_\bot$ that decides L in time $\mathrm{O}(g(n))$. Let us observe the diagonal entries for $M_\bot$ in Table 4.3.

For a padded code w of $M_\bot$, M on the input w makes at most

$$\mathrm{O}\left(g(|w|) \log g(|w|) + |w|\right) = \mathrm{o}(f(|w|))$$

steps when simulating $M_\bot$ on w (see Proposition 3.6.2). Thus, for long enough padded codes w of $M_\bot$, M halts because $M_\bot$ halts and not because M would run out of 1s on the fourth tape. Hence, if w is long enough, M on the input w simulates all the steps of $M_\bot$ on w and it returns the opposite as $M_\bot$ does. It follows that M accepts w if and only if $M_\bot$ rejects w, which is a contradiction with the definition of $M_\bot$, because it should accept exactly the same inputs as M. □

Corollary 4.2.2. *For any integer $k \geq 1$,*

$$\mathsf{DTIME}(n^k) \subsetneq \mathsf{DTIME}(n^k \log^2 n) \subsetneq \mathsf{DTIME}(n^{k+1}).$$

4.2.3 The Non-Deterministic Time Hierarchy

The deterministic time hierarchy gave us

$$\mathsf{DTIME}(n) \subsetneq \mathsf{DTIME}(n^2) \subsetneq \mathsf{DTIME}(n^{10}) \subsetneq \mathsf{P} \subsetneq \mathsf{DTIME}(2^n) \subsetneq \mathsf{EXP},$$

however the same idea does not go through to prove the inclusions

$$\mathsf{NTIME}(n) \subsetneq \mathsf{NTIME}(n^2) \subsetneq \mathsf{NTIME}(n^{10}) \subsetneq \mathsf{NP} \subsetneq \mathsf{NTIME}(2^n) \subsetneq \mathsf{NEXP}.$$

While it is easy to flip the answer of a DTM after the simulation:

If the DTM goes to q_{ACC}, reject, else accept,

this is not so easy in the non-deterministic case because there are several possible computations on each input. To flip the answer of an NTM, we would have to know if all of the computations on the input are rejecting or if there exists an accepting computation. Nevertheless, it turns out that we can overcome this difficulty by a method called *lazy diagonalization*.

Theorem 4.2.3. *For any time constructible function $f : \mathbb{N} \to \mathbb{N}$ such that $n = \mathrm{o}(f(n))$ and for any function $g : \mathbb{N} \to \mathbb{N}$ such that $g(n+1) = \mathrm{o}(f(n))$, there exists a language*

$$L \in \mathsf{NTIME}(f(n)) \backslash \mathsf{NTIME}(g(n)).$$

Note that this theorem gives a more strict hierarchy for non-deterministic classes than Theorem 4.2.1 does for deterministic classes, because in this theorem there is no $\log(g(n))$ factor in $g(n+1) = \mathrm{o}(f(n))$. Such a strong hierarchy was proven first by Seiferas, Fischer, and Meyer in 1978 [27] and the proof was simplified by Žák in 1983 [31]. However, we use the idea for the proof from Fortnow and Santhanam [8].

Proof. Recall from Section 3.3.5 the definition of a *two-choice NTM* which is an NTM that has at most two possible non-deterministic choices in each step. In the proof of this theorem, we will use a special encoding for pairs (M, y), where M is a two-choice NTM and $y \in \{0,1\}^*$. If u is a binary code of a multi-tape two-choice NTM M, then let i_u be the positive integer that is represented as $1u$ in binary. Now the pair (M, y) is encoded as $1^{i_u}01^m0y$, where m is any non-negative integer. Hence, each pair (M, y) has arbitrarily long codes and, given such a code, we can get the code of M in linear time. For a code w of the pair (M, y), let M_w denote the NTM M.

Let us observe the ordered pairs (M_{x_i}, x_j) where x_i and x_j are codes of (M_{x_i}, ε) and (M_{x_j}, ε), respectively. For each such a pair, M_{x_i} either accepts the input x_j or it rejects it (see Table 4.4).

Let a constant $C \in \mathbb{N}$ be such that f can be computed in time $Cf(n)$ and let M be the following 4-tape NTM. On an input $w = 1^i01^m0y$, it first computes $1^{f(|w|)}$ and stores it on the fourth tape. Denote $x = 1^i01^m0$ and let M compute $f(|x|-1)$, but only if the computation does not take more than $Cf(|w|)$ steps. If it does, M rejects. Then we have two cases.

(i) If $|y| < f(|x|-1)$, then M computes just as the universal 3-tape (non-deterministic) Turing machine from Proposition 3.6.1 on the inputs $(M_w, w0)$ and $(M_w, w1)$, i.e., it simulates M_w on the inputs $w0$ and $w1$ one after another. For each step that M makes, it erases one symbol

	x_1	x_2	x_3	$\cdots$	x_i	$\cdots$
M_{x_1}	**reject**	reject	accept	$\cdots$	accept	$\cdots$
M_{x_2}	accept	**reject**	accept	$\cdots$	reject	$\cdots$
M_{x_3}	accept	accept	**reject**	$\cdots$	accept	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
M_{x_i}	reject	accept	reject	$\cdots$	**accept**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$

Table 4.4: A table of whether the two-choice NTM M_{x_i} in the leftmost column accepts the input x_j in the top row, where x_i and x_j are codes of (M_{x_i}, ε) and (M_{x_j}, ε), respectively. It suffices to have only NTMs that run in time $\mathrm{O}(g(n))$ in this table. Because each two-choice NTM $\overline{M}$ has infinitely many codes of $(\overline{M}, \varepsilon)$, each two-choice NTM that runs in time $\mathrm{O}(g(n))$ is present infinitely many times in the leftmost column of the table.

1 from the fourth tape. The fourth tape thus serves as a counter and when there are no more ones on it, M rejects. If the two simulations of M_w finished when there was still at least one symbol 1 on the fourth tape, M accepts if and only if M_w accepted both inputs $w0$ and $w1$ on the two simulated computations.

(ii) If $|y| \geq f(|x| - 1)$, then M simulates the computation of M_w on the input x determined by the non-deterministic choices encoded by y. Because M_w is two-choice, y encodes $|y|$ non-deterministic choices. Again, the fourth tape serves as a counter and for each step that M makes, it erases one symbol 1 from the fourth tape. If the simulation of M_w finished when there was still at least one symbol 1 on the fourth tape and if M_w rejects the input x on the computation encoded by y, M accepts the input w, else it rejects it. If the counter on the fourth tape ever hits 0 or if y does not encode enough non-deterministic choices of M_w to finish with the simulation, M rejects.

Let us show how M can efficiently do the simulation in case (ii). A straightforward deterministic simulation would not suffice because the simulation of the deterministic universal Turing machine has too much overhead.

- First, M computes a code of a DTM $\tilde{M}$ that computes as follows: given a pair of strings (x_1, y_1), $\tilde{M}$ first copies y_1 to a non-input tape and then it computes the same as M_w on x_1 using the non-deterministic choices encoded by y_1, only that if M_w accepts, then $\tilde{M}$ rejects and if M_w rejects, then $\tilde{M}$ accepts. If y_1 does not encode enough non-deterministic choices, $\tilde{M}$ rejects.

  Clearly M can construct such $\tilde{M}$ that uses $\mathrm{O}(1)$ steps to simulate one step of M_w, only in the beginning of a computation it uses $\mathrm{O}(n)$ steps to copy y_1 to a special tape so that it can read it in parallel while simulating M_w. Note that the number of steps needed to construct $\tilde{M}$ depends only on M_w and is independent of m and y.
- Next, M computes as the universal (non-deterministic) Turing machine from Proposition 3.6.1 and simulates $\tilde{M}$ on the input (x, y).

Let L be the language decided by M. We see that M runs in time $\mathrm{O}(f(n))$, hence $L \in \mathsf{NTIME}(f(n))$.

It is enough to prove that $L \notin \mathsf{NTIME}(g(n))$. If $L \in \mathsf{NTIME}(g(n))$, then there exists some NTM $M_\perp$ that decides L in time $\mathrm{O}(g(n))$. By Proposition 3.3.16 we may assume that $M_\perp$ is two-choice, hence $M_\perp$ is present in Table 4.4. The next two paragraphs are technical and are used to find an appropriate code for the pair $(M_\perp, \varepsilon)$ such that that its diagonal entry in Table 4.4 cannot take any value. If the reader wants to skip the technicalities, we recommend jumping to the last paragraph of the proof.

Let $x = 1^i 0 1^m 0$ be a code of the pair $(M_\perp, \varepsilon)$. Because $n = \mathrm{o}(f(n))$, there exists an infinite increasing sequence of integers $m < m_1 < m_2 < m_3 < \cdots$ such that $f(i + m_j + 1) < f(\xi)$ for all positive integers j and $\xi > i + m_j + 1$, i.e., f takes only bigger values than $f(i + m_j + 1)$ for bigger arguments. Note that if we only consider the codes $x = 1^i 0 1^{m_j} 0$ for $(M_\perp, \varepsilon)$, the Turing machine M on the inputs xy for $y \in \{0,1\}^*$ always computes $f(|x| - 1)$, because it has enough time.

Next, we show that for large enough m_j, M on inputs $w = xy$ where $x = 1^i 0 1^{m_j} 0$ and $y \in \{0,1\}^*$ never runs out of 1s on the fourth tape. By Proposition 3.6.1, M makes at most $\mathrm{O}(g(|w| + 1)) = \mathrm{o}(f(|w|))$ steps when simulating $M_\perp$ on the inputs $w0$ and $w1$ in case (i), where the constant behind the little o can depend on $M_\perp$. Hence, for large enough m_j, M does not halt in case (i) because it would run out of 1s on the fourth tape. In case (ii), M has to simulate $\mathrm{O}(g(|x|))$ steps of the computation of $M_\perp$ on the input x, where the computation is determined by the non-deterministic choices given by y. Although M has to simulate just one fixed computation, it uses non-determinism to do it faster as we described in the description of M. This way M simulates $\mathrm{O}(g(|x|))$ steps of $M_\perp$ in time

$$\mathrm{O}\left(g(|x|) + |x| + |y|\right) = \mathrm{o}\left(f(|x| - 1)\right) + \mathrm{O}\left(|y|\right),$$

which is $\mathrm{o}(f(|w|))$ because $f(|x|-1) < f(|w|)$ by the definition of m_j. This implies that, for large enough m_j, M does not run out of time on the fourth tape. Additionally, because $|y| \geq f(|x|-1)$, and because M needs only to simulate $\mathrm{O}(g(|x|)) = \mathrm{o}(f(|x| - 1)$ steps of $M_\perp$, for large enough m_j, y encodes enough non-deterministic choices for the simulation.

To finish the proof, let m_j be big enough so that, for any input $w = xy$ where $x = 1^i 0 1^{m_j} 0$ and $y \in \{0,1\}^*$, M on input w

- never runs out of time on the fourth tape and
- if we have case (ii), M simulates $M_\perp$ until the end (i.e., y encodes enough non-deterministic choices).

Considering the diagonal entry for x in Table 4.4, we get

$$\begin{aligned}
& M_\perp \text{ accepts the input } x \\
\iff & M \text{ accepts the input } x \\
\iff & M \text{ accepts the inputs } x0 \text{ and } x1 \\
\iff & M \text{ accepts all the inputs } xy \text{ where } y \in \{0,1\}^2 \\
\iff & M \text{ accepts all the inputs } xy \text{ where } y \in \{0,1\}^3 \\
\iff & M \text{ accepts all the inputs } xy \text{ where } y \in \{0,1\}^{f(|x|-1)} \\
\iff & M_\perp \text{ rejects } x \text{ on all computations} \\
\iff & M_\perp \text{ rejects } x,
\end{aligned}$$

which is a contradiction. □

Corollary 4.2.4. *For any integer $k \geq 1$,*

$$\mathsf{NTIME}(n^k) \subsetneq \mathsf{NTIME}(n^k \log(\log n)) \subsetneq \mathsf{NTIME}(n^k \log n) \subsetneq \mathsf{NTIME}(n^{k+1}).$$

4.3 Relativization

In the previous section we managed to separate several complexity classes by efficiently simulating Turing machines. In all the simulations that were analyzed, we did not care about how a simulated Turing machine computes, we just simulated each step using a Turing machine of the same kind. This way we were able to build a hierarchy of non-deterministic classes and a hierarchy of deterministic classes. However, we still do not know how to compare deterministic and non-deterministic complexity classes like P and NP. In this section we present a well known result that using only brute force simulations, we cannot solve the P versus NP problem.

4.3.1 Oracle Turing Machines

Oracle Turing machines are a much stronger model of computation than the standard Turing machines and the Church-Turing thesis does not hold for them. They compute relative to some *oracle* which is a language over Σ^*. In particular, their computation is just as a computation of a standard Turing machine only that, time to time, they can "ask the oracle" whether some string from Σ^* is in the oracle. Hence, oracle Turing machines can decide any undecidable language L if they compute with the help of the oracle L. If some result (lemma, proposition, theorem ...) about Turing machines together with its proof is also valid for oracle Turing machines with the oracle O, for all oracles $O \subseteq \Sigma^*$, then we say that the result *relativizes* because it is valid *relative* to any oracle. Later in Section 4.3.4 we prove that a solution to the P versus NP problem must not relativize.

Oracle Turing machines are usually defined (see e.g. [2, Chapter 3.4]) to be just as normal Turing machines, only with an additional tape called the *oracle tape* which serves for oracle queries: to query the oracle, the oracle Turing machine just writes a string from Σ^* on the oracle tape and then enters a special query state. Then in one step it magically enters one of two special states, depending on whether the content on the oracle tape was a string from the oracle or not.

However, such a definition will not suffice for us. We want to define oracle Turing machines in such a way that we will be able to have one-tape oracle Turing machines and that we will be able to analyze crossing sequences[3] of such machines. This will be used later to show that the results in Chapter 5 relativize and to argue that using only methods from Chapter 6 we cannot solve the P versus NP problem. However, our definition of the oracle Turing machine will be polynomially equivalent to the standard one in the sense that, using the same oracle, our oracle Turing machine that runs in time $T(n)$ can be simulated by a standard oracle Turing machine in time $T(n)^k$ for some constant k, and vice versa.

A *k-tape non-deterministic oracle Turing machine* (abbreviated as k-tape *NOTM*) is a 10-tuple $M = (Q, \Sigma, \Gamma, \sqcup, \gg, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}}, q_{\text{YES}})$, where

Q	...	a finite set of states,
Σ	...	the input alphabet fixed in Section 2.1.3,
$\Gamma = \Sigma \cup \{\sqcup\}$	...	a (fixed) tape alphabet,

[3] Crossing sequences are defined in Chapter 5.

$\llcorner \in \Gamma \backslash \Sigma$ ... a blank symbol,

$\gg \notin \Gamma$ ... an oracle symbol,

$\delta : (\Gamma \cup \{\gg\}) \times \Gamma^{k-1} \times Q \backslash \{q_{\text{ACC}}, q_{\text{REJ}}\} \to \mathcal{P}(\Gamma^k \times Q \backslash \{q_{\text{YES}}\} \times \{-1, 0, 1\}^k) \backslash \{\emptyset\}$

... a transition function and

$q_0, q_{\text{ACC}}, q_{\text{REJ}}, q_{\text{YES}} \in Q$ ... pairwise distinct starting, accepting, rejecting and YES states.

Note that an NOTM has the same elements as an NTM with an additional special state q_{YES}, an additional symbol $\gg$ and it has a fixed tape alphabet (the reason for the latter is given below). The NOTM M always computes relative to some oracle $O \subseteq \Sigma^*$ and we denote by M^O the NOTM M with the oracle O. Its computation is just as by an ordinary NTM with the following distinctions:

- On the input tape, there is always exactly one special symbol $\gg$ that divides the input tape into the left part, called the *oracle part* and the right part, called the *standard part* of the tape. All other tapes always contain only symbols from Γ.
- Before a computation begins, the symbol $\gg$ is in cell -1, just left of the input (see Figure 3.1) and it remains in this cell after each step of the computation. If the transition function wants to replace it with some other symbol, the new symbol is ignored.
- The major difference in the computation of an NOTM compared to an NTM is the following. When M^O reads the symbol $\gg$ on the input tape and then wants to move the head on the input tape to the right, "magic" happens.
  - If there is a string from O written left next to the symbol $\gg$ followed to the left by only blank symbols, then M^O ignores the transition function δ in this step and it goes to the state q_{YES}, moving the head on the input tape for one cell to the right (where it should go without magic) and changing nothing on the other tapes.
  - Else, it keeps computing as the transition function δ dictates.

Note that (normal) Turing machines with the tape alphabet $\Sigma \cup \{\llcorner\}$ are oracle Turing machines with the oracle which is the empty language (up to the symbol $\gg$ on the input tape) because such oracle Turing machines never go to the state q_{YES}. The reason for why we decided to fix the tape alphabet of oracle Turing machines to $\Sigma \cup \{\llcorner\}$ lies in Section 3.3.1 where we discussed reductions of tape alphabets. The results in this section trivially relativize if the tape alphabet is $\Sigma \cup \{\llcorner\}$, however if we allowed a general tape alphabet $\Gamma \supset \Sigma$, we would have troubles reducing it to $\Sigma \cup \{\llcorner\}$ without making considerably more steps.

A *k-tape deterministic oracle Turing machine* (abbreviated as k-tape *DOTM*) M is the same as a k-tape NOTM only that in each step M has only one possible move.

Relativized Complexity Classes

For a fixed oracle O, all complexity classes from Section 3.4.1 could be defined also with oracle Turing machines. The classes obtained this way are called *relativized* and are denoted with the superscript O. This way we get the classes $\mathsf{DTIME}^O(T(n))$, $\mathsf{NTIME}^O(T(n))$, P^O, NP^O, $\mathsf{co\text{-}NP}^O$ ... For example, the class P^O is the class of languages decidable in polynomial time by a multi-tape DOTM with the oracle O. We also define relativized classes of complete problems for NP^O and $\mathsf{co\text{-}NP}^O$ by using polynomial-time (standard) DTMs for reductions.

4.3.2 Encodings of Oracle Turing Machines

NOTMs can have the same encoding as the ordinary NTMs (see Section 3.6) with the exception of the symbol $\gg$ and the state q_{YES} that need to be marked somehow. It is worth noting that an oracle is not part of the code of the oracle Turing machine. While in Section 3.6 we stated that using our encoding we can compute the composition of two NTMs in linear time, we do not claim the same for NOTMs, mainly because of the state q_{YES} which makes it harder to define a composition of two NOTMs. However, in our applications (Section 6.2.3) it will always be the case that the first Turing machine in a composition of two Turing machines will not need an oracle, hence it can be treated as an NTM and in such a case we can compute a composition of an NTM and an NOTM in linear time.

Note that an NOTM with an oracle O can be simulated by an NOTM with the same oracle.

4.3.3 Results that Relativize

In this section we argue why all results so far in this chapter relativize and we discuss how statements from Chapter 3 should change in order to hold also for oracle Turing machines. We will not give rigorous proofs of the corresponding results with oracle Turing machines, we will just comment on how the proof or the statement of a result should be adjusted.

Results from Section 3.3

In Section 3.3 we compared how different attributes of Turing machines influence time complexity. While most of the statements and the proofs remain essentially the same if we use oracle Turing machines with a fixed oracle, there are some exceptions.

- Lemma 3.3.1 relativizes.
- In Subsection 3.3.2 we analyzed linear speedup. The proofs of those results do not relativize because we cannot compress the content on the oracle part of the input tape.
- In Proposition 3.3.8 we simulated a multi-tape Turing machine M on a one-tape Turing machine $\tilde{M}$ with a quadratic overhead. The same cannot be done so easily with oracle Turing machines because it is hard to keep track of the position of the head on the oracle part of the input tape, especially with the tape alphabet $\Sigma \cup \{\llcorner\}$. However, for oracle Turing machines, we can prove a slightly weaker result, namely a multi-tape Turing machine M can be simulated on a one-tape Turing machine $\tilde{M}$ with a cubic overhead. This can be proven by simulating all tapes of M on the standard part of the tape of $\tilde{M}$ and copying the oracle queries to the oracle part of the tape when queried. Now if M runs in time $T(n)$, then because copying a query of size ℓ to the oracle part of the tape takes $\mathrm{O}(\ell^2)$ steps on one-tape oracle Turing machines, we get that $\tilde{M}$ runs in time $\mathrm{O}(T^3(n))$.

  The same holds for Corollary 3.3.9.
- In Proposition 3.3.10 we simulated a multi-tape NTM M that runs in time $T(n)$ on a 2-tape NTM. We used a bigger tape alphabet than $\Sigma \cup \{\llcorner\}$, however this was not necessary. The input can be stored on the second tape and the content of the second tape can be encoded in binary. To show that the result relativizes, the input tape should only be used to simulate the input tape of M. When simulating non-input tapes of M, the role of the input tape and the second tape should exchange in order for the head on the input tape to avoid the oracle part of

the tape. Not to produce too much overhead when exchanging the role of the input tape and the second tape, we should start with $\ell = (\text{the length of the input})$ and, if $T(n_0) < n_0 + 1$ for some $n_0 \in \mathbb{N}$, we should use Lemma 3.3.2.

- In Proposition 3.3.12 we simulated a multi-tape DTM M on a 2-tape DTM. The way the proof is written it cannot be straightforwardly relativized since the input tape is used in both directions to simulate all the tapes of M. Hence, the result can be changed a bit: we can simulate a multi-tape DOTM M that runs in time $T(n)$ on a 3-tape DOTM that runs in time $\mathrm{O}\big(T(n)\log T(n)\big)$. This result can be obtained by simulating the input tape separately and the rest of the tapes as in Proposition 3.3.12. Thus, we do not need other symbols than $\Sigma \cup \{\llcorner\}$ on the input tape. On the other two tapes we can encode each new symbol in binary and the same proof goes through.

  The same remark holds for Corollary 3.3.13.

- In the proof of Proposition 3.3.14, we used the input tape only for storing the input. It can as well be used to query oracles, so the result relativizes.

  On the other hand, the proof of Corollary 3.3.15 does not relativize, because we have to reduce a multi-tape Turing machine to a one-tape Turing machine. However, the same proof gives that if a language $L \subseteq \Sigma^*$ is decided in time $T(n)$ by a multi-tape NOTM M, then it is also decided by some one-tape DOTM in time $2^{\mathrm{O}(T(n)^3)}$.

- Proposition 3.3.16 relativizes.

Universal Oracle Turing Machines

In the proof of Proposition 3.6.1 we describe a universal 3-tape NTM U. If we change the purpose of the first and the second tape of U, this result relativizes. Similarly, Proposition 3.6.2 relativizes, however the resulting universal DTM has 4 tapes.

Results Proven by Diagonalization

It is clear that the undecidability of the halting problem relativizes (for each oracle we get another halting problem). What is more, the deterministic and the non-deterministic time hierarchy theorems relativize (the proofs are essentially the same as in the non-relativized setting). Hence, we have the following strict inclusions for each oracle O:

$$\mathsf{DTIME}^O(n) \subsetneq \mathsf{DTIME}^O(n^2) \subsetneq \mathsf{DTIME}^O(n^{10}) \subsetneq \mathsf{P}^O \subsetneq \mathsf{DTIME}^O(2^n) \subsetneq \mathsf{EXP}^O$$

and

$$\mathsf{NTIME}^O(n) \subsetneq \mathsf{NTIME}^O(n^2) \subsetneq \mathsf{NTIME}^O(n^{10}) \subsetneq \mathsf{NP}^O \subsetneq \mathsf{NTIME}^O(2^n) \subsetneq \mathsf{NEXP}^O.$$

4.3.4 Limits of Proofs that Relativize

In this section we prove that a solution to the P versus NP problem must not relativize. More specifically, we will exhibit two oracles A and B such that $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B \neq \mathsf{NP}^B$. Such oracles were first found by Baker, Gill and Solovay [3].

Lemma 4.3.1. *There exists an oracle $A \in \mathsf{EXP}$ such that $\mathsf{P}^A = \mathsf{NP}^A$.*

Proof. Let A be a language corresponding to the following decision problem:

Given a one-tape DTM M and an input w for M, does M accept the input w in at most $2^{|w|}$ steps?

First, we prove that $\mathsf{EXP} \subseteq \mathsf{P}^A$. Let L be a language from EXP and let M be a one-tape DTM that decides L in time $2^{p(n)}$ for some polynomial $p : \mathbb{N} \to \mathbb{N}$. Let $\tilde{M}$ be a one-tape DTM that on inputs of the form $0^i 1w$ where $w \in \Sigma^*$ and $i \in \mathbb{N}$, it first erases $0^i 1$ and then it computes exactly as M on w. Now the following DOTM M_1 decides L with the use of oracle A in polynomial time. M_1^A on an input w writes a code x of the pair $(\tilde{M}, 0^{p(|w|)}1w)$ on the oracle part of the input tape and then it accepts if $x \in A$, else it rejects. It is clear that M^A runs in polynomial time. Using the inequality $1 + x \leq 2^x$ for $x \in \mathbb{N}$, we get

$$\begin{aligned} w \in L &\implies M \text{ accepts } w \text{ in at most } 2^{p(|w|)} \text{ steps} \\ &\implies \tilde{M} \text{ accepts } 0^{p(|w|)}1w \text{ in at most } 2^{p(|w|)} + p(|w|) + 1 \text{ steps} \\ &\implies \tilde{M} \text{ accepts } 0^{p(|w|)}1w \text{ in at most } 2^{p(|w|)+1+|w|} \text{ steps} \\ &\implies M_1^A \text{ accepts } w \end{aligned}$$

and

$$\begin{aligned} w \notin L &\implies M \text{ rejects } w \text{ in at most } 2^{p(|w|)} \text{ steps} \\ &\implies \tilde{M} \text{ rejects } 0^{p(|w|)}1w \text{ in at most } 2^{p(|w|)} + p(|w|) + 1 \text{ steps} \\ &\implies \tilde{M} \text{ rejects } 0^{p(|w|)}1w \text{ in at most } 2^{p(|w|)+1+|w|} \text{ steps} \\ &\implies M_1^A \text{ rejects } w. \end{aligned}$$

Hence, $L \in \mathsf{P}^A$ which implies $\mathsf{EXP} \subseteq \mathsf{P}^A$.

It is clear that $A \in \mathsf{EXP}$: to verify whether a given one-tape DTM M accepts a given input w in at most $2^{|w|}$ steps, we first compute $2^{|w|}$ and then simulate M on the input w for $2^{|w|}$ steps. We return the same as M returns or, if M does not halt, we reject.

Next, we show that $\mathsf{NP}^A \subseteq \mathsf{EXP}$. Let L be a language from NP^A and let M_1 be an NOTM that decides L with the help of the oracle A in time $p(n)$ for some monotonically increasing polynomial $p : \mathbb{N} \to \mathbb{N}$. Now the following DTM M decides L in exponential time. M on an input w simulates all possible computations of M_1^A on the input w and, for each oracle query of M^A, it computes the output itself. There are $2^{\mathrm{O}(p(|w|))}$ possible computations of M_1^A on input w, hence M needs to simulate at most $p(|w|)2^{\mathrm{O}(p(|w|))}$ steps of M_1^A altogether. Because $A \in \mathsf{EXP}$, A can be decided by a DTM in time $\mathrm{O}(2^{\tilde{p}(n)})$ for some polynomial $\tilde{p}$. Because each oracle query that needs to be answered for M_1^A on the input w is at most $p(|w|)$ long, M spends $\mathrm{O}(2^{\tilde{p}(p(|w|))})$ steps answering it. Hence, M needs to simulate at most exponentially many steps of M_1^A and each step can be simulated in exponential time (in the length of the input). This implies that M runs in exponential time, hence $L \in \mathsf{EXP}$.

To sum up, we have proven

$$\mathsf{EXP} \subseteq \mathsf{P}^A \subseteq \mathsf{NP}^A \subseteq \mathsf{EXP},$$

which implies $\mathsf{P}^A = \mathsf{NP}^A$. □

Lemma 4.3.2. *There exists an oracle $B \in \mathsf{EXP}$ such that $\mathsf{P}^B \neq \mathsf{NP}^B$.*

Proof. We will actually prove a bit stronger statement. We will exhibit an oracle $B \in \mathsf{EXP}$ and a language $L \in \mathsf{NP}^B$ such that, for each function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that $f(n) = \mathrm{o}(2^n)$, it will hold $L \notin \mathsf{DTIME}^B(f(n))$. It is clear that this implies $\mathsf{P}^B \neq \mathsf{NP}^B$.

Let us have some enumeration of DOTMs such that each DOTM appears infinitely many times in the sequence M_1, M_2, M_3 ... and given an index i in binary, we can construct a code of M_i in time $\mathrm{O}(\log i)$, i.e., in linear time. Such an enumeration can be obtained in the following way. If u is a padded binary code of a DOTM M, then let i_u be the positive integer that is represented as $1u$ in binary. We say that i_u represents M and we define $M_{i_u} = M$. For all integers that do not represent a DOTM, we define that they represent some fixed DOTM, like for example a DOTM that rejects every input in one step.

Algorithm 1: $\mathcal{B}$

Input: $w \in \Sigma^*$

$B = \emptyset$

/* B will be a "growing" oracle that will contain only strings of size at most $|w|$ and at most one string of each size. */

$S = \emptyset$

/* S will be a set of all oracle queries inside the following for loop. */

for $i = 1, 2 \ldots |w|$ **do**

  Simulate M_i^B on the input 1^i for 2^{i-1} steps (or until M_i^B halts) and add all oracle queries of M_i^B on this computation to S.

  /* Note that we are making a sort of diagonalization: The machine M_i^B has the same index as the input length, which determines also the bound for the number of steps. */

  if M_i^B *rejects during the simulation* **then**

    Let x be the lexicographically first string from Σ^i that is not in S

    /* Note that the string x always exists because S has at most

    $$2^0 + 2^1 + 2^2 + \cdots + 2^{i-1} = 2^i - 1$$

    elements and there are at least 2^i strings in Σ^i. */

    $B = B \cup \{x\}$

    /* Note that because $x \notin S$, for $j \leq i$, adding x to B does not change the computation of M_j^B on the input 1^j in the first 2^{j-1} steps. */

if $w \in B$ **then accept**

else reject

Figure 4.1: Algorithm $\mathcal{B}$ that defines the oracle B.

Consider the algorithm $\mathcal{B}$ in Figure 4.1 and let B be the language decided by Algorithm $\mathcal{B}$. Note that if the for loop in Algorithm $\mathcal{B}$ would run forever (not until $i = |w|$), then the set B from the algorithm would "converge" to the language B. What is more, for each i, the DOTM M_i with the oracle B in the first 2^{i-1} steps on the input 1^i computes the same as in the algorithm where the oracle B is only partially built.

It is clear that the Algorithm $\mathcal{B}$ runs in exponential time, hence $B \in \mathsf{EXP}$. Now define the language

$$L = \{1^m; m \in \mathbb{N} \text{ and there exists a string } x \in \Sigma^m \text{ such that } x \in B\}.$$

Clearly, $L \in \mathsf{NP}^B$; the certificate for $1^m \in L$ is $x \in \Sigma^m \cap B$. Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function such that $f(n) = \mathrm{o}(2^n)$. The only thing left to prove is that $L \notin \mathsf{DTIME}^B(f(n))$.

If $L \in \mathsf{DTIME}^B(f(n))$, then there exists a DOTM M with oracle B that decides L in time $\mathrm{O}(f(n))$. Because $f(n) = \mathrm{o}(2^n)$ and because M is represented by infinitely many integers, we can choose a large enough k so that $M = M_k$ and M_k on inputs of length k makes at most 2^{k-1} steps. By Algorithm $\mathcal{B}$, we have that M_k^B rejects the string 1^k if and only if B contains a string of length k, which is true if and only if $1^k \in L$ by the definition of L. This is a contradiction with the fact that M_k^B decides L. □

Together, Lemma 4.3.1 and Lemma 4.3.2 imply the following theorem.

Theorem 4.3.3. *There exist oracles A and B in EXP such that $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B \neq \mathsf{NP}^B$.*

Chapter 5

Crossing Sequences

While the Chapters 2, 3 and 4 of the dissertation discussed results that are at least mentioned in most standard textbooks, this chapter is more specific. It speaks about crossing sequences, a notion that is defined only for one-tape Turing machines and it helps us to understand why one-tape Turing machines are weaker than multi-tape Turing machines. In particular, crossing sequences help us to prove that deciding the problem PALINDROME takes at least quadratic time on one-tape Turing machines, while linear time is enough on multi-tape Turing machines (see Proposition 5.1.10).

We begin by introducing the notion of crossing sequences and then we present the standard cut-and-paste technique that is used in most of the results in this chapter, at least implicitly. We prove that one-tape Turing machines that run in time $o(n \log n)$ actually run in linear time and accept a regular language. The main theorem in this section, called the compactness theorem (Theorem 5.2.1), tells essentially that to verify whether a given one-tape Turing machine runs in a specified linear time bound, we only need to verify that it does not run for too long on short inputs.

All the main results in this chapter relativize. We actually defined the oracle Turing machines in such a way that the results in this chapter would relativize. Although we will only talk about NTMs and DTMs, we will mention NOTMs where there needs to be a special notice.

5.1 Definition and Basic Results

For a one-tape Turing machine M, we can number the cells of its tape with integers so that the cell 0 is the one where M starts its computation. Using this numbering we can number the boundaries between cells as shown in Figure 5.1.

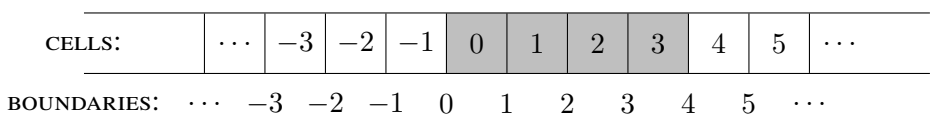


Figure 5.1: Numbering of tape cells and boundaries of a one-tape Turing machine. The shaded part is a potential input of length 4. If the Turing machine is an oracle Turing machine, there is the symbol $\gg$ in the cell -1.

Intuitively, a *crossing sequence generated by a one-tape NTM M after t steps of a computation ζ on an input w at a boundary i* is a sequence of states of M in which M crosses the ith boundary of its tape when considering the first t steps of the computation ζ on the input w. We assume that,

in each step, M first changes the state and then moves the head. A more formal definition is given in the next paragraph.

Suppose that a one-tape NTM M in the first $t \in \mathbb{N} \cup \{\infty\}$ steps of a computation ζ on an input w crosses a boundary i of its tape at steps $t_1, t_2 \ldots$ (this sequence can be finite or infinite). If M was in a state q_j after the step t_j for all j, then we say that M produces the *crossing sequence* $\mathcal{C}_i^t(M, \zeta, w) = q_1, q_2 \ldots$ and we denote its length by $|\mathcal{C}_i^t(M, \zeta, w)| \in \mathbb{N} \cup \{\infty\}$. The sequence alternates left-moves and right-moves and begins with a left move if and only if $i \leq 0$. Note that $\mathcal{C}_i^t(M, \zeta, w)$ contains all information that the machine M carries across the ith boundary of the tape in the first t steps of the computation ζ. If we denote $\mathcal{C}_i(M, \zeta, w) = \mathcal{C}_i^{|\zeta|}(M, \zeta, w)$, the following trivial identity holds because the head of M must move in each step:

$$|\zeta| = \sum_{i=-\infty}^{\infty} |\mathcal{C}_i(M, \zeta, w)|.$$

5.1.1 The Cut-and-Paste Technique

Let τ be the tape of a one-tape NTM M with some symbols written on it. We can cut the tape on finitely many boundaries to get *tape segments* $\tau_1, \tau_2 \ldots \tau_k$ so that $\tau = \tau_1\tau_2 \cdots \tau_k$, where τ_1 is left infinite, τ_k is right infinite and the other segments are finite. We can also start with a tuple of segments $\tilde{\tau}_1, \tilde{\tau}_2 \ldots \tilde{\tau}_l$ and glue them together to get the tape $\tilde{\tau} = \tilde{\tau}_1\tilde{\tau}_2 \cdots \tilde{\tau}_l$. This can be done if $\tilde{\tau}_1$ is left infinite, $\tilde{\tau}_l$ is right infinite, other segments are finite and exactly one of the segments has a prescribed location for the cell 0 (where M starts its computation). We consider a tape in the same way as an input, i.e., we can give a tape to M and it will start computing on it as its transition function determines. Hence, if M is given a tape τ, then for a step t of a computation ζ of M on τ, the crossing sequence generated at a boundary i is denoted $\mathcal{C}_i^t(M, \zeta, \tau)$. Although a tape can be filled with random symbols, in our main applications it will correspond to some input w being written on it.

Because the crossing sequences contain all the information the Turing machine carries across some boundary of the tape, the next proposition is very intuitive. The proof is a slight generalization of a result of Hennie [17].

Proposition 5.1.1. *Let M be a one-tape NTM and let $\tau_1\tau_2$ and $\tilde{\tau}_1\tilde{\tau}_2$ be two tapes. Let t be a step of a computation ζ of M on the tape $\tau_1\tau_2$ and let $\tilde{t}$ be a step of a computation $\tilde{\zeta}$ of M on the tape $\tilde{\tau}_1\tilde{\tau}_2$. Suppose that the segments τ_1 and τ_2 are joined at a boundary $i > 0$ and the segments $\tilde{\tau}_1$ and $\tilde{\tau}_2$ are joined at a boundary $\tilde{i} > 0$. If*

$$\mathcal{C}_i^t(M, \zeta, \tau_1\tau_2) = \mathcal{C}_{\tilde{i}}^{\tilde{t}}(M, \tilde{\zeta}, \tilde{\tau}_1\tilde{\tau}_2),$$

then there exists a computation $\zeta_{1\tilde{2}}$ of M on the tape $\tau_1\tilde{\tau}_2$ and a step $t_{1\tilde{2}} \in \mathbb{N} \cup \{\infty\}$ such that,

a) the crossing sequences generated by M at the corresponding boundaries of the segment τ_1 in the first t steps of the computation ζ and in the first $t_{1\tilde{2}}$ steps of the computation $\zeta_{1\tilde{2}}$ are identical,

b) the crossing sequences generated by M at the corresponding boundaries of the segment $\tilde{\tau}_2$ in the first $\tilde{t}$ steps of the computation $\tilde{\zeta}$ and in the first $t_{1\tilde{2}}$ steps of the computation $\zeta_{1\tilde{2}}$ are identical.

What this proposition (together with its proof) tells is that, for tapes $\tau_1\tau_2$ and $\tilde{\tau}_1\tilde{\tau}_2$ and for computations ζ and $\tilde{\zeta}$ of an NTM M as in the proposition, we can cut the tapes and glue them to get the tape $\tau_1\tilde{\tau}_2$ and there will exist a computation of M on the tape $\tau_1\tilde{\tau}_2$ that will act as ζ on the part τ_1 and as $\tilde{\zeta}$ on the part $\tilde{\tau}_2$ for the first few steps. If $t = |\zeta|$ and $\tilde{t} = |\tilde{\zeta}|$, then there will exist a computation of M on the tape $\tau_1\tilde{\tau}_2$ that will act as ζ on the part τ_1 and as $\tilde{\zeta}$ on the part $\tilde{\tau}_2$ until the end of the computation.

Proof. Let $\mathcal{C} = q_1, q_2 \ldots$ be the crossing sequence produced by M on the tape $\tau_1\tau_2$ at the boundary i after the first t steps of the computation ζ. This means that the boundary i is crossed to the right in the state q_1 for the first time and that the tape segment τ_2 is such that the head of M can return across the ith boundary in the state q_2. Then the head of M is on the left side of the boundary i and it does not know anything about the other side of the boundary except that it was able to "return the head" when M went to the state q_2. Then M computes on the left side of the boundary i until it again crosses the ith boundary in the state q_3 and the right side of the boundary i is such that M can cross back in the state $q_4 \ldots$ The analogous situation is happening during the computation $\tilde{\zeta}$ of M on the tape $\tilde{\tau}_1\tilde{\tau}_2$. Hence, the tape segments τ_2 and $\tilde{\tau}_2$ are such that if M enters them from the left in the state q_1, then there exist computations of M such that M can leave these segments in the state q_2. If M then again enters in the state q_3, M can cross back in the state $q_4 \ldots$ The same philosophy is for the left tape segments τ_1 and $\tilde{\tau}_1$. They are such that M can leave them in the state q_1 and if M enters back in the state q_2, it can leave them again in the state $q_3 \ldots$

Hence, there exists a computation $\zeta_{1\tilde{2}}$ of M on the tape $\tau_1\tilde{\tau}_2$ and $t_{1\tilde{2}} \in \mathbb{N} \cup \{\infty\}$ such that a) and b) hold. The computation $\zeta_{1\tilde{2}}$ is just as ζ on the tape segment τ_1 in the first t steps and $\tilde{\zeta}$ on the tape segment $\tilde{\tau}_2$ in the first $\tilde{t}$ steps. □

Note that the conditions $i > 0$ and $j > 0$ cause the cell 0 to be in the left tape segments, which makes it possible to swap the right tape segments. The same result also holds if we put $i, j \leq 0$. However, for $i, j \leq 0$ (or $i, j \leq -1$) the result does not hold for NOTMs, because changes in the oracle part of the tape influence a computation when crossing the boundary 0.

The following corollary is now trivial.

Corollary 5.1.2. *Let $\tau_1\tau_2\tau_3$ be the tape of a one-tape NTM M. Suppose that the segments τ_1 and τ_2 are joined at a boundary $i > 0$ and the segments τ_2 and τ_3 are joined at a boundary j. If M on a computation ζ on the tape $\tau_1\tau_2\tau_3$ produces the same crossing sequence at the boundaries i and j after $t \in \mathbb{N} \cup \{\infty\}$ steps then, for each $n \in \mathbb{N}$, there exists a computation ζ_n of M on the tape $\tau_1(\tau_2)^n\tau_3$ and a step $t_n \in \mathbb{N} \cup \{\infty\}$ such that M on the tape $\tau_1(\tau_2)^n\tau_3$ on the first t_n steps of the computation ζ_n produces the same crossing sequences at the corresponding boundaries of the segments τ_1, τ_3 and of each copy of the segment τ_2 as in the first t steps of the computation ζ on the tape $\tau_1\tau_2\tau_3$.*

Note that the condition $i > 0$ could be replaced by $j \leq 0$. In other words, if the same crossing sequence appears on both ends of some tape segment that does not contain the cell 0, then we can remove this segment or add extra copies of it next to each other without affecting the result of the computation. The same result holds also for NOTMs, but only for $i > 0$.

Proof. Consider two copies of the tape $\tau_1\tau_2\tau_3$, one with τ_1 and τ_2 joined into a left segment and one with τ_2 and τ_3 joined into a right segment and apply Proposition 5.1.1. The corollary follows by induction. □

5.1.2 One-Tape Turing Machines that Run in Time $o(n \log n)$

In this section we prove that a one-tape NTM (and also an NOTM) that runs in time $o(n \log n)$ actually runs in linear time (Corollary 5.1.6) and that it accepts a regular language (Corollary 5.1.7).

The next lemma, implicit in Kobayashi [20], and in Tadaki, Yamakami and Lin [29] tells what is so special with one-tape NTMs that run in time $o(n \log n)$.

Lemma 5.1.3. *Let $T : \mathbb{N} \to \mathbb{R}_{>0}$ be a function such that $T(n) = o(n \log n)$ and let*

$$g(n) = \begin{cases} \frac{n \log n}{T(n)} & ; \quad n \geq 2 \\ 1 & ; \quad n = 0, 1. \end{cases}$$

Then, for any integer $q \geq 2$, there exists a constant c such that any one-tape NTM with q states that runs in time $T(n)$, on each computation on each input produces only crossing sequences of lengths bounded by c. What is more, c can be any constant satisfying $c \geq \max\{T(0), T(1)\}$ and the following inequality:

$$3\frac{qn^{(\log q)/g(n)^{1/2}} - 1}{q - 1} \leq n - 3 - \frac{n}{g(n)^{1/2}} + c\frac{g(n)^{1/2}}{\log n} \tag{5.1}$$

for all $n \geq 2$.

Note that since $\lim_{n\to\infty} g(n) = \infty$, then, for any $q \geq 2$, there exists a constant $c \geq \max\{T(0), T(1)\}$ such that Inequality (5.1) holds for all $n \geq 2$. The lemma holds also for one-tape NOTMs.

Proof. Let M be a one-tape NTM with q states that runs in time $T(n)$. Let $c \geq \max\{T(0), T(1)\}$ be such that Inequality (5.1) holds for all $n \geq 2$ and suppose that M produces a crossing sequence of length more than c on some input. Let w be a shortest such input, let ζ be the corresponding computation and let $n_0 = |w|$. Note that $n_0 \geq 2$ since $c \geq \max\{T(0), T(1)\}$. Suppose w was given to M and M followed the steps of the computation ζ.

Let h be the number of boundaries from $\{1, 2 \ldots n_0 - 1\}$ at which crossing sequences of lengths less than $(\log n_0)/g(n_0)^{1/2}$ were produced. Then we have

$$\frac{n_0 \log n_0}{g(n_0)} = T(n_0) > c + (n_0 - 2 - h)\frac{\log n_0}{g(n_0)^{1/2}}$$

and hence

$$\begin{aligned} h &> n_0 - 2 - \frac{n_0}{g(n_0)^{1/2}} + c\frac{g(n_0)^{1/2}}{\log n_0} \\ &\geq 3\frac{qn_0^{(\log q)/g(n_0)^{1/2}} - 1}{q - 1} + 1 \\ &= 3\frac{q^{(\log n_0)/g(n_0)^{1/2}+1} - 1}{q - 1} + 1. \end{aligned}$$

Moreover, a simple counting shows that there are at most

$$\frac{q^{(\log n_0)/g(n_0)^{1/2}+1} - 1}{q - 1}$$

distinct crossing sequences of lengths less than $(\log n_0)/g(n_0)^{1/2}$.

Hence, by the pigeonhole principle, there exist at least four boundaries in $\{1, 2 \dots n_0 - 1\}$ at which the same crossing sequence $\mathcal{C}$ was produced. Now if a crossing sequence of length more than c was produced at some boundary $i \in \mathbb{Z}$, we can find two boundaries in $\{1, 2 \dots n_0 - 1\}$ at which $\mathcal{C}$ was produced, such that i does not lie between them. If we cut away the substring of w between those two boundaries, we get an input for M of length less than n_0 on which M produces a crossing sequence of length more than c. This contradicts the selection of w and completes the proof of the lemma. □

This lemma has some interesting consequences.

Corollary 5.1.4. *If a one-tape NTM M runs in time $T(n) = \mathrm{o}(n \log n)$, then there exists a constant D, such that M on each input w visits at most $|w| + D$ cells.*

Proof. By Lemma 5.1.3, the length of crossing sequences produced by M is bounded by a constant and thus M produces only constantly many distinct crossing sequences. If K is this constant, let us prove that M visits at most K cells to the right of the input.

Suppose that this is not true for some input w. If we run M on w, then there are at least two boundaries with index greater than or equal to $|w|$, say i and j, that produce the same non-empty crossing sequence. At the beginning of the computation we only have blank symbols between those two boundaries, thus all boundaries $i + k|j - i|$ for $k \in \mathbb{N}$ produce the same crossing sequence. This is a contradiction with M running in finite time, thus M visits at most K cells to the right of the input.

The same way we can show that M visits at most K cells to the left of the input, which completes the proof. □

While the above corollary holds also for NOTMs, we have to be more careful with the last sentence of the proof because it claims something about the oracle part of the tape. Actually, if M is an NOTM, a different argument is needed to show that it visits only $\mathrm{O}(1)$ cells to the left of an input.

Corollary 5.1.5. *If a one-tape NOTM M runs in time $T(n) = \mathrm{o}(n \log n)$, then there exists a constant D, such that M on each input w visits at most $|w| + D$ cells.*

Proof. We are only left to show that M visits $\mathrm{O}(1)$ cells to the left of each input. Suppose that this is not true. Hence, for each cell with a negative index, M visits this cell on some computation on some input. Recall that by Lemma 5.1.3, M produces only constantly many distinct crossing sequences. Hence, there exist a crossing sequence $\mathcal{C}$, non-empty inputs $w_0, w_1 \dots$ and computations $\zeta_0, \zeta_1 \dots$ of M on these inputs such that M produces the crossing sequence $\mathcal{C}$ at the boundary 1 on each of these computations and, for each $i \in \mathbb{N}$, M visits the cell $-i$ on the computation ζ_i. For each $i \in \mathbb{N}$, let $w_i = a_i \tilde{w}_i$ for $a_i \in \Sigma$ and $\tilde{w}_i \in \Sigma^*$. By Proposition 5.1.1 it follows that M on inputs $a_0\tilde{w}_0, a_1\tilde{w}_0, a_2\tilde{w}_0 \dots$ visits each of the cells with a negative index on appropriate computations, because left of the boundary 1 M can compute as $\zeta_0, \zeta_1, \zeta_2 \dots$ and right from the boundary 1 it can compute as ζ_0. This is a contradiction with the fact that M makes at most $T(|w_0|)$ steps on the inputs $a_0\tilde{w}_0, a_1\tilde{w}_0, a_2\tilde{w}_0 \dots$ which are all of length $|w_0|$. □

This corollary is interesting also because it implies that a one-tape NOTM that runs in time $\mathrm{o}(n \log n)$ makes only a constant number of distinct oracle queries on all computations on all inputs. This furthermore implies that an oracle query can be computed in constant time, which implies that a language that is decided by some one-tape NOTM that runs in time $\mathrm{o}(n \log n)$ can also be decided

by a (standard) one-tape NTM in time $o(n \log n)$. This claim will be even more strengthened by Proposition 5.1.7.

The next corollary, proven also by Pighizzini [26], tells us that time $o(n \log n)$ for one-tape Turing machines actually means linear time. It holds also for NOTMs.

Corollary 5.1.6. *If a one-tape NTM runs in time $o(n \log n)$, then it runs in linear time.*

Proof. Let M be a one-tape NTM that runs in time $o(n \log n)$. The main observation is that M on each input w on each computation ζ halts exactly after $\sum |\mathcal{C}|$ steps, where the sum is over all crossing sequences $\mathcal{C}$ produced at boundaries of the tape. From Lemma 5.1.3 it follows that each addend is bounded by a constant and from Corollary 5.1.4 it follows that there are at most $|w| + D$ of them for some constant D. □

Finally, we show that one-tape NTMs that run in time $o(n \log n)$ accept only regular languages. The idea of the proof is from Pighizzini [26] and it holds also for NOTMs because a one-tape NOTM that runs in time $o(n \log n)$ can be simulated by a one-tape NTM that runs in time $o(n \log n)$ (see the text before Corollary 5.1.6).

Proposition 5.1.7. *If a one-tape NTM runs in time $o(n \log n)$, then it decides a regular language.*

Proof. Let M be a one-tape NTM that runs in time $o(n \log n)$ and let L be the language that M decides. We may assume that, for all n, M on inputs of length n in the last step of each of its computations crosses the boundary n of its tape (in one of its halting states). This can be assumed because M can, for example, in the first $O(n)$ steps mark the cell with the last symbol of an input (using additional symbols) and when the computation is finished, M can search for this cell (in linear time by Corollary 5.1.4) and halt while moving its head to the neighboring cell on the right.

Let us describe an NFA $\tilde{M} = (Q, \Sigma, \delta, q_s, F)$ that decides L. By Lemma 5.1.3, the length of crossing sequences produced by M is bounded by a constant and thus M produces only constantly many crossing sequences. Let S be the set of these crossing sequences and let $F \subseteq S$ be the subset of S of such crossing sequences that end with an accepting state. Note that the crossing sequences from F can only be produced at the rightmost boundary of some input. Define $Q = S \cup \{q_s\}$ for some new state q_s and, if M accepts input ε, add q_s to F. The definition of δ is the following.

- For each $\mathcal{C} \in S$ and for each $a \in \Sigma$, let there exist an edge from q_s to $\mathcal{C}$ with the weight a if there exists some input w of M that begins with the symbol a and a computation ζ of M on w such that M produces the crossing sequence $\mathcal{C}$ at the boundary 1 on computation ζ.

- For each $\mathcal{C}_1, \mathcal{C}_2 \in S$ and for each $a \in \Sigma$, let there exist an edge from $\mathcal{C}_1$ to $\mathcal{C}_2$ with the weight a if there exists some input w of M with the $(i > 1)$th symbol a and a computation ζ of M on w such that M on computation ζ produces the crossing sequence $\mathcal{C}_1$ at the boundary $i-1$ and the crossing sequence $\mathcal{C}_2$ at the boundary i.

It is clear that if M accepts an input w, then $\tilde{M}$ also accepts the input w: $\tilde{M}$ just follows the sequence of states that are crossing sequences produced by an accepting computation of M on w at boundaries $1, 2 \ldots |w|$.

Now suppose that $\tilde{M}$ accepts an input $\tilde{w}$.

- If $|\tilde{w}| = 0$, then $\tilde{w} = \varepsilon$ and M accepts $\tilde{w}$.

- If $|\tilde{w}| = 1$, then $\tilde{M}$ accepts $|\tilde{w}|$ in one step, hence there exists an edge from q_s to some $\mathcal{C} \in F$. This implies that there exists an input w of M that begins with $\tilde{w}$ and M produces the crossing sequence $\mathcal{C}$ at the boundary 1 on some computation. Because $\mathcal{C}$ ends with the accepting state and because M always finishes its computation at the end of its input, it follows that $|w| = 1$ which implies $\tilde{w} = w$ and M accepts the input $\tilde{w}$.

- If $|\tilde{w}| = k > 1$, then let $\tilde{w} = \tilde{a}_1\tilde{a}_2 \dots \tilde{a}_k$ where, for each i, $\tilde{a}_i \in \Sigma$. Let $q_s, \mathcal{C}_1, \mathcal{C}_2 \dots \mathcal{C}_k$ be a sequence of states of $\tilde{M}$ that on the input $\tilde{w}$ lead to an accepting state $\mathcal{C}_k \in F$. By the definition of $\tilde{M}$ there exists an input w_1 that begins with the symbol $\tilde{a}_1$ such that M on the input w_1 produces the crossing sequence $\mathcal{C}_1$ on some computation ζ_1 at the boundary 1. Similarly, for each $1 < j \leq k$ there exists an input w_j with the non-first symbol $\tilde{a}_j$ such that M on the input w_j on some computation ζ_j produces the crossing sequences $\mathcal{C}_{j-1}$ left of the symbol $\tilde{a}_j$ and the crossing sequence $\mathcal{C}_j$ right of the symbol $\tilde{a}_j$. Note that because $\mathcal{C}_k$ ends with an accepting state and the computation ζ_k produces the crossing sequence $\mathcal{C}_k$ on the right of the symbol $\tilde{a}_k$, this symbol is the last symbol of the input w_k. Let ζ be the computation of M on the input $\tilde{w}$ that computes as

  - ζ_1 when the head of M is over the first symbol of $\tilde{w}$ or left of it,
  - ζ_j when the head of M is over the jth symbol of $\tilde{w}$ for $1 < j < k$,
  - ζ_k when the head of M is over the last symbol of $\tilde{w}$ or right of it.

  It is clear that M on the computation ζ produces the crossing sequences $\mathcal{C}_1$, $\mathcal{C}_2 \dots \mathcal{C}_k$ at the boundaries $1, 2 \dots k$, respectively. Hence, the computation ζ of M on the input $\tilde{w}$ is accepting.

To sum up, the NTM M and the NFA $\tilde{M}$ accept the same language L, thus L is regular. □

An "algorithmic" version of the above proof will be given later in Theorem 5.2.10.

5.1.3 Simple Applications

In this section we prove lower and upper bounds on the number of steps required to solve the problems Compare Length and Palindrome by one-tape Turing machines. The lower bounds will be proven with the help of crossing sequences.

The problem Compare Length was defined in Section 1.1 and it asks whether a given string is of the form 0^k1^k for some $k \in \mathbb{N}$. Let L_c be the corresponding language

$$L_c = \{0^k1^k;\ k \in \mathbb{N}\} \subseteq \Sigma^*.$$

The problem Palindrome was defined in Section 2.2.1 and it asks whether a given string is a palindrome. We denote by $L_p \subseteq \Sigma^*$ the language of palindromes. In Section 2.2.1 we proved that the language L_p is not regular and in the next lemma we use crossing sequences to show that the language L_c is not regular either.

Lemma 5.1.8. *The language L_c is not regular.*

Proof. If the language L_c was regular, then it would be recognized by some DFA, hence there would exist a one-tape DTM M that would decide L_c and would generate only crossing sequences of length 1 at all boundaries on all inputs. Hence, there would exist two different integers $k_1, k_2 > 0$

such that the crossing sequence produced by M on the input $0^{k_1}1^{k_1}$ at the boundary k_1 would be the same as the crossing sequence produced by M on the input $0^{k_2}1^{k_2}$ at the boundary k_2. Because both of these inputs are accepting, M also accepts the input $0^{k_1}1^{k_2}$ by Lemma 5.1.1 which gives a contradiction. □

The next proposition gives a tight bound on how fast a one-tape Turing machines can solve the problem Compare Length. Together with Proposition 5.1.7 it also implies that the time bounds $\Theta(n \log n)$ are the tightest that allow a one-tape Turing machine to recognize a non-regular language.

Proposition 5.1.9. *The language $L_c = \{0^k1^k;\ k \in \mathbb{N}\}$ can be decided in time $\mathrm{O}(n \log n)$ by a one-tape DTM, but not in time $\mathrm{o}(n \log n)$ by any one-tape NOTM with any oracle.*

Proof. Because L_c is not regular, it cannot be solved in time $\mathrm{o}(n \log n)$ by any one-tape NOTM with any oracle by Proposition 5.1.7. Hence we only have to prove the upper bound. To do so, let M be the following one-tape DTM. On an input w of length n, M first verifies in $\mathrm{O}(n)$ steps whether the input is of the form 0^k1^ℓ for some $k, \ell \in \mathbb{N}$ and if not, it rejects. Additionally, M writes some special symbols like $\#$ in the cell left and in the cell right of the input to mark the input part of the tape. In all steps that follow, M will pass through the input part of the tape from one symbol $\#$ to the other one, each time making $\mathrm{O}(n)$ steps. In one pass, M can verify whether k and ℓ are of the same parity. If not, it rejects, else, it erases $\lceil k/2 \rceil$ zeros and $\lceil l/2 \rceil$ ones in one pass by erasing every second 0 and every second 1. Now there are exactly $\lfloor k/2 \rfloor$ zeros and $\lfloor l/2 \rfloor$ ones remaining on the tape. M again verifies the parity of $\lfloor k/2 \rfloor$ and $\lfloor l/2 \rfloor$ and if they are distinct, M rejects. If they are the same, M erases one half of 0s and one half of 1s that are left on the tape. It continues this way until there are still some 0s and 1s on the tape. If at the end all 0s and all 1s have been erased, then M accepts, else it rejects. What is M actually doing is verifying whether the digits of the binary representation of the numbers k and l match (starting from the last digit). For each digit M needs $\mathrm{O}(n)$ steps, hence $\mathrm{O}(n \log n)$ altogether. □

The next proposition tells that the problem Palindrome is harder to solve on one-tape Turing machines than the problem Compare Length. This result can be found in Kozen [21, Chapter 1]. The idea of the proof is from Hennie [17]. The proposition holds for NOTMs as well.

Proposition 5.1.10. *If an NTM M decides the language of palindromes L_p, then it does not run in time $\mathrm{o}(n^2)$.*

Proof. Let an NTM M with q states decide the language of palindromes. Let us observe, for a fixed integer $k > \log q$, the inputs of the form

$$a_1 0 a_2 0 \dots 0 a_k 0 b_1 0 b_2 0 \dots b_k 0 1 1 0 b_k 0 b_{k-1} 0 \dots 0 b_1 0 a_k 0 a_{k-1} 0 \dots 0 a_1 \tag{5.2}$$

where $a_i, b_i \in \Sigma$ for all i. Note that two symbols 1 are consecutive only in the middle of the observed palindromes. Let us fix an accepting computation of M on each of these inputs (because they are palindromes, such a computation exists).

First note that, over all inputs of the form (5.2) and over all boundaries $1, 2, \dots 4k$ (these are the boundaries on the left of the two consecutive ones), M produces pairwise distinct crossing sequences. If not, then there exist inputs

$$a_1 0 a_2 0 \dots 0 a_k 0 b_1 0 b_2 0 \dots b_k 0 1 1 0 b_k 0 b_{k-1} 0 \dots 0 b_1 0 a_k 0 a_{k-1} 0 \dots 0 a_1$$

and

$$c_1 0 c_2 0 \dots 0 c_k 0 d_1 0 d_2 0 \dots d_k 0 1 1 0 d_k 0 d_{k-1} 0 \dots 0 d_1 0 c_k 0 c_{k-1} 0 \dots 0 c_1$$

such that we can cut them somewhere on the left of the two consecutive ones and cross-join them, which would result in an input accepted by M. However, such crossbreeding inputs are not palindromes.

Next, we claim that there exist symbols $a_1, a_2 \dots a_k \in \Sigma$ such that, for each $b_1, b_2 \dots b_k \in \Sigma$, M on the fixed accepting computations on inputs

$$a_1 0 a_2 0 \dots 0 a_k 0 b_1 0 b_2 0 \dots b_k 0 1 1 0 b_k 0 b_{k-1} 0 \dots 0 b_1 0 a_k 0 a_{k-1} 0 \dots 0 a_1$$

produces only crossing sequences of length more than

$$\frac{k - \lceil \log q \rceil}{\lceil \log q \rceil}$$

at boundaries $2k, (2k+1), (2k+2) \dots (4k-1)$, which are the boundaries between the symbols b_i and symbols 0. If this is true, then M on such inputs makes at least

$$2k \frac{k - \lceil \log q \rceil}{\lceil \log q \rceil} = \Omega(k^2)$$

steps, hence it does not run in time $o(n^2)$.

To prove that the desired symbols $a_1, a_2 \dots a_k \in \Sigma$ exist, let us assume the contrary. Then, for each of the 2^k possible beginnings $a_1 0 a_2 0 \dots 0 a_k$, there exist symbols $b_1, b_2 \dots b_k \in \Sigma$ and a boundary from $\{2k, (2k+1), (2k+2) \dots (4k-1)\}$ such that a crossing sequence of length at most

$$\frac{k - \lceil \log q \rceil}{\lceil \log q \rceil}$$

is produced on it. Note that these "short" crossing sequences have to be pairwise distinct over all observed inputs. Because there are only

$$1 + q + q^2 + \dots + q^{(k - \lceil \log q \rceil)/\lceil \log q \rceil} = \frac{q^{k/\lceil \log q \rceil} - 1}{q - 1}$$

distinct crossing sequences of length at most

$$\frac{k - \lceil \log q \rceil}{\lceil \log q \rceil},$$

which is strictly less than 2^k, we came to a contradiction. □

We have a tight bound on how fast a one-tape Turing machines can solve the problem PALINDROME.

Corollary 5.1.11. *The language of palindromes L_p can be decided in time $O(n^2)$ by a one-tape DTM, but not in time $o(n^2)$ by any one-tape NOTM with any oracle.*

Proof. The lower bound is proven by Proposition 5.1.10 and the upper bound follows by the following algorithm that can be implemented in $O(n^2)$ time on a one-tape DTM. Given an input w, verify whether the first symbol and the last symbol are the same and delete them. If they were not the same, reject, else continue comparing the first and the last symbol until there is only one or no symbols left. Then accept. □

Interestingly, the complement of the problem PALINDROME can be solved faster then the problem itself by one-tape NTMs.

Proposition 5.1.12. *The complement $\overline{L_p}$ of the language of palindromes L_p can be decided in time $\mathrm{O}(n \log n)$ by a one-tape NTM, but not in time $\mathrm{o}(n \log n)$ by any one-tape NOTM with any oracle.*

Proof. Because regular languages are closed under complementation, the language $\overline{L_p}$ is not regular and the lower bound follows. For the upper bound, consider the following algorithm that can be implemented in $\mathrm{O}(n \log n)$ time on a one-tape NTM. Given an input w, non-deterministically guess the middle of the input and mark it. If $|w|$ is even, then insert a new (arbitrary) symbol between the middle two symbols and mark it. Next, non-deterministically choose and mark one symbol left from the middle and one symbol right from the middle. If the symbols are the same, reject, else we have the situation as in Figure 5.2. Everything until now can be done in linear time. Next, verify whether there are equally many symbols of the input left from $\#1$ as they are right from $\#2$ and if not, reject. This can be done deterministically in time $\mathrm{O}(n \log n)$ as in the proof of Proposition 5.1.9. Next, verify whether there are equally many symbols between $\#1$ and $\#$ as they are between $\#$ and $\#2$ and if not, reject. Else, accept. Again, this can be done deterministically in time $\mathrm{O}(n \log n)$ as in the proof of Proposition 5.1.9. It is clear that there exists an accepting computation if and only if the input was not a palindrome. □

Figure 5.2: Suppose that a one-tape NTM guessed the middle symbol $\#$ (wrongly) and the symbols $\#1$ and $\#2$ left and right from the middle symbol. The shaded part is the input.

5.2 The Compactness Theorem

In this section, we present the compactness theorem proven by the author in [11]. Simply put, if we want to verify that an NTM M runs in time $Cn + D$, we only need to verify the number of steps that M makes on inputs of some bounded length. The result can also be found, in a weaker form, in [10].

The main technique used to prove the compactness theorem is the cut-and-paste technique explained in Section 5.1.1. We show that a Turing machine that runs in time $Cn + D$ must produce some identical crossing sequences on each computation, if the input is long enough. Thus, when considering some fixed computation, we can partition the input on some parts where identical crossing sequences are generated, and analyze each part independently. We prove that it is enough to consider small parts of the input.

Later in Section 5.2.3 we prove some supplementary results to the compactness theorem. Among other we give an explicit upper bound on the length of the crossing sequences that are produced by a one-tape NTM that runs in time $Cn + D$. We also give an algorithm that takes a one-tape NTM M and integers $C, D \in \mathbb{N}$ as input and, if M runs in time $Cn + D$, returns an equivalent NFA.

5.2.1 Computation on a Part

Before we formally state the compactness theorem, let us define $t_M(w, \mathcal{C})$. Intuitively, $t_M(w, \mathcal{C})$ is the maximum number of steps that a one-tape NTM M makes on a **part** w of an imaginary input, if we only consider such computations on which M produces the crossing sequence $\mathcal{C}$ at both (the left and the right) boundaries of w. To define it more formally, we will describe a valid *computation of M on a part w with frontier crossing sequence $\mathcal{C} = (q_1, q_2 \dots q_l)$*. A similar but slightly less general definition was given also by Pighizzini [26]. We will use the term *standard case* to refer to the definition of a computation of an NTM on a given input (not on a part). Assume $|w| = n \geq 1$ and let $M = (Q, \Sigma, \Gamma, \textvisiblespace, \delta, q_0, q_{\text{ACC}}, q_{\text{REJ}})$.

- A *valid configuration* is a 5-tuple $(\mathcal{C}_1, \tilde{w}, i, \tilde{q}, \mathcal{C}_2)$, where $\mathcal{C}_1$ is the *left crossing sequence*, $\tilde{w}$ is some string from Γ^n, $0 \leq i \leq n-1$ is the position of the head, $\tilde{q} \in Q$ is the current state of M and $\mathcal{C}_2$ is the *right crossing sequence*. Intuitively, $\mathcal{C}_1$ and $\mathcal{C}_2$ are the suffixes of $\mathcal{C}$ that still need to be matched.

- The *starting configuration* is $((q_2, q_3 \dots q_l), w, 0, q_1, (q_1, q_2 \dots q_l))$. As in the standard case, we imagine the input being written on the tape of M with the first symbol in the cell 0 (where also the head of M is). The head will never leave the portion of the tape where the input is written. Note that q_1 is missing in the left crossing sequence because we pretend that the head just moved from the cell -1 to the cell 0.

- Valid configurations $A = (\mathcal{C}_{1A}, w_A, i, q_A, \mathcal{C}_{2A})$ and $B = (\mathcal{C}_{1B}, w_B, j, q_B, \mathcal{C}_{2B})$ are *successive*, if one of the following holds:
  - the transition function of M allows (w_A, i, q_A) to change into (w_B, j, q_B) as in the standard case, $\mathcal{C}_{1A} = \mathcal{C}_{1B}$ and $\mathcal{C}_{2A} = \mathcal{C}_{2B}$,
  - $i = j = 0$, $\mathcal{C}_{1A}$ is of the form $(\tilde{q}, q_B, \mathcal{C}_{1B})$, $w_A = a\tilde{w}$, $w_B = b\tilde{w}$, $(\tilde{q}, b, -1) \in \delta(q_A, a)$ and $\mathcal{C}_{2A} = \mathcal{C}_{2B}$, or
  - $i = j = n-1$, $\mathcal{C}_{2A}$ is of the form $(\tilde{q}, q_B, \mathcal{C}_{2B})$, $w_A = \tilde{w}a$, $w_B = \tilde{w}b$ and $(\tilde{q}, b, 1) \in \delta(q_A, a)$ and $\mathcal{C}_{1A} = \mathcal{C}_{1B}$.

- There is a special *ending configuration* that can be reached from configurations of the form
  - $((q_l), a\tilde{w}, 0, \tilde{q}, ())$, if $(q_l, b, -1) \in \delta(\tilde{q}, a)$ for some $b \in \Gamma$ or
  - $((), \tilde{w}a, n-1, \tilde{q}, (q_l))$, if $(q_l, b, 1) \in \delta(\tilde{q}, a)$ for some $b \in \Gamma$.

- A *valid computation of M on the part w with frontier crossing sequence $\mathcal{C}$* is any sequence of successive configurations that begins with the starting configuration and ends with the ending configuration.

Similar to the standard case, we can define $\mathcal{C}_i(M, \zeta, w, \mathcal{C})$ to be the crossing sequence generated by M on the computation ζ on the part $w \in \Sigma^n$ with the frontier crossing sequence $\mathcal{C}$ at the boundary i $(1 \leq i \leq n-1)$. We define

$$|\zeta| = |\mathcal{C}| + \sum_{i=1}^{n-1} |\mathcal{C}_i(M, \zeta, w, \mathcal{C})|$$

as the *length of the computation ζ (on the part w)*. Figure 5.3 justifies this definition.

We define $t_M(w, \mathcal{C}) \in \mathbb{N} \bigcup \{-1\}$ as the length of the longest computation of M on the part w with the frontier crossing sequence $\mathcal{C}$. If there is no valid computation of M on the part w with the frontier crossing sequence $\mathcal{C}$ or $|\mathcal{C}| = \infty$, then we define $t_M(w, \mathcal{C}) = -1$.

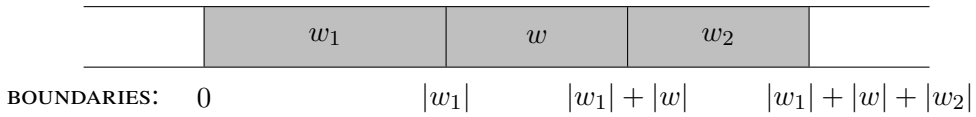


Figure 5.3: Suppose an input w_1ww_2 is given to M, $|w_1|, |w| \geq 1$ and let a computation ζ produce the same crossing sequence $\mathcal{C}$ at boundaries $|w_1|$ and $|w_1| + |w|$. If ζ_1 is the corresponding computation of M on the part w, then M on the computation ζ spends exactly $|\zeta_1|$ steps on the part w. What is more, if the input w_1w_2 is given to M (we cut out w) and we look at the corresponding computation ζ_2 extracted from ζ thus forming a crossing sequence $\mathcal{C}$ at the boundary $|w_1|$, then $|\zeta_2| = |\zeta| - |\zeta_1|$. Such considerations will be very useful in the proof of the compactness theorem.

5.2.2 The Compactness Theorem

For a positive integer n and a one-tape NTM M, define

$$\mathcal{S}_n(M) = \{\mathcal{C}_i^t(M, \zeta, w);\ |w| = n,\ 1 \leq i \leq n,\ \zeta \text{ computation on input } w,\ t \leq |\zeta|\}.$$

Thus $\mathcal{S}_n(M)$ is the set of all possible beginnings of the crossing sequences that M produces on the inputs of length n at the boundaries $1, 2 \ldots n$.

Theorem 5.2.1 (The compactness theorem)**.** *Let M be a one-tape NTM with q states and let $C, D \in \mathbb{N}$. Denote $\ell = D + 8q^C$, $r = D + 12q^C$ and $\mathcal{S} = \bigcup_{n=1}^{\ell} \mathcal{S}_n(M)$. It holds:*
*M runs in time $Cn + D$ **if and only if***

*a) for each input w of length at most ℓ and for each computation ζ of M on w, it holds $|\zeta| \leq C|w| + D$ **and***

b) for each $\mathcal{C} \in \mathcal{S}$ and for each part w of length at most r, for which $t_M(w, \mathcal{C}) \geq 0$, it holds $t_M(w, \mathcal{C}) \leq C|w|$.

Before going to the proof, let us argue that the theorem is in fact intuitive. If a Turing machine M runs in time $Cn + D$, then a) tells us that M must run in that time for small inputs and b) tells us that on small parts w that can be "inserted" into some input from a), M must make at most $C|w|$ steps. For the opposite direction, one can think about constructing each input for M from several parts from b) inserted into some input from a) at appropriate boundaries, which results in running time $Cn + D$.

The following lemma already proves one direction of the compactness theorem.

Lemma 5.2.2. *Let all assumptions be as in Theorem 5.2.1. If b) does not hold, then there exists some input z of length at most $\ell + (Cr + D)r$ such that M makes more than $C|z| + D$ steps on z on some computation.*

Proof. If b) does not hold, then there exists some finite crossing sequence $\mathcal{C} \in \mathcal{S}$, a part w of length at most r and a valid computation ζ_1 of M on the part w with the frontier crossing sequence $\mathcal{C}$, such that $|\zeta_1| \geq C|w| + 1$. From the definition of $\mathcal{S}$ we know that there exist strings w_1 and w_2 such that $|w_1| \geq 1$ and $|w_1| + |w_2| \leq \ell$ and a computation ζ_2, such that $\mathcal{C}$ is generated by M at the boundary $|w_1|$ on the input w_1w_2 on the computation ζ_2 after some number t of steps. As in Figure 5.3, we can now insert w between w_1 and w_2. In fact we can insert as many copies of w

between w_1 and w_2 as we want, because the crossing sequence $\mathcal{C}$ will always be formed between them.

Let us look at the input $z = w_1 w^{Cr+D} w_2$ for M. Let ζ be a computation of M on z that on the part w_1 (and left of it) and on the part w_2 (and right of it) it acts like the first t steps of ζ_2, and on the copies of w it acts like ζ_1. Note that after ζ spends t steps on the parts w_1 and w_2, the crossing sequence $\mathcal{C}$ is generated at the boundaries $|w_1|, (|w_1| + |w|) \dots (|w_1| + (Cr + D)|w|)$ and by that time M makes at least $t + (Cr + D)(C|w| + 1)$ steps. Using $t \geq 1$ and $r \geq \ell \geq |w_1| + |w_2|$, we see that M makes at least

$$C(Cr + D)|w| + C(|w_1| + |w_2|) + D + 1 = C|z| + D + 1$$

steps on the computation ζ on the input z. Because $|w| \leq r$ and $|w_1| + |w_2| \leq \ell$, we have $|z| \leq \ell + (Cr + D)r$ and the lemma is proven. □

Next, we prove the main lemma for the proof of the other direction of the compactness theorem.

Lemma 5.2.3. *Let C and D be non-negative integers, M a one-tape q-state NTM and w an input for M of length n. Assume that, on some computation on the input w after at most $Cn + D$ steps, each crossing sequence produced by M at the boundaries $1, 2 \dots n$ appears at most k times. Then $n \leq D + 4kq^C$.*

Proof. Let M make at least $t \leq Cn + D$ steps on a computation ζ on the input w and suppose that each crossing sequence produced by M on ζ after t steps at the boundaries $1, 2 \dots n$ appears at most k times. We know that $Cn + D \geq t \geq \sum_{i=1}^{n} |\mathcal{C}_i^t(M, \zeta, w)|$, thus

$$\begin{aligned}
n &\leq D + (C+1)n - \sum_{i=1}^{n} |\mathcal{C}_i^t(M, \zeta, w)| \\
&= D + \sum_{i=1}^{n} (C + 1 - |\mathcal{C}_i^t(M, \zeta, w)|) \\
&\leq D + \sum_{j=0}^{C+1} \sum_{\substack{i=1 \\ |\mathcal{C}_i^t(M,\zeta,w)|=j}}^{n} (C + 1 - j) \\
&\leq D + \sum_{j=0}^{C+1} kq^j (C + 1 - j) \\
&\leq D + 4kq^C,
\end{aligned}$$

where the last inequality follows by a technical lemma proven next. □

Lemma 5.2.4. *For every $q \geq 2$ and $C \in \mathbb{N}$, it holds*

$$\sum_{j=0}^{C} q^j (C - j) = \frac{q^{C+1} - (C+1)q + C}{(q-1)^2} \leq 4q^{C-1}.$$

Proof.

$$\begin{aligned}\sum_{j=0}^{C} q^j(C-j) &= C\sum_{j=0}^{C} q^j - q\frac{d}{dq}\left(\sum_{j=0}^{C} q^j\right)\\ &= C\frac{q^{C+1}-1}{q-1} - q\frac{d}{dq}\left(\frac{q^{C+1}-1}{q-1}\right)\\ &= \frac{q^{C+1}-(C+1)q+C}{(q-1)^2}.\end{aligned}$$

It is easy to see that, for $q \geq 2$, it follows

$$\begin{aligned}\frac{q^{C+1}-(C+1)q+C}{(q-1)^2} &\leq \frac{q^{C+1}}{(q-1)^2}\\ &\leq 4q^{C-1}.\end{aligned}$$

□

Before going into the proof of the compactness theorem, let us recall the definition of $w(i,j)$ which is the substring of a string w, containing symbols from ith to jth, including ith and excluding jth (we start counting with 0). Alternatively, if w is written on a tape of a Turing machine, $w(i,j)$ is the string between the ith and jth boundary.

Proof of the compactness theorem (Theorem 5.2.1). If M runs in time $Cn+D$, then a) obviously holds and b) holds by Lemma 5.2.2. Now suppose that a) and b) hold. We will make a proof by contradiction, so suppose that M does not run in time $Cn+D$. Let w be a shortest input for M such that there exists a computation of M on w of length more than $C|w|+D$. Denote this computation by ζ and let $n=|w|$, $t=Cn+D$.

Before we continue, let us give an outline of what follows in one paragraph. Our first goal is to find closest boundaries j_1 and j_2 such that M produces the same crossing sequence $\mathcal{C} = \mathcal{C}^{t+1}_{j_1}(M,\zeta,w) = \mathcal{C}^{t+1}_{j_2}(M,\zeta,w)$ at them after the (t)th and the $(t+1)$st step of the computation ζ (see Figure 5.4). Then using the fact that w is a shortest input for M such that there exists a computation of M on w of length more than $C|w|+D$, we argue that $t_M(w(j_1,j_2),\mathcal{C}) > C|w(j_1,j_2)|$. Now the most important part of w is between the boundaries j_1 and j_2, so we want to cut out the superfluous parts to the left of j_1 and to the right of j_2 (see Figure 5.5). After the cutting out we get an input $w_1w(j_1,j_2)w_2$ on which M on the computation corresponding to ζ on the time-step corresponding to t generates the crossing sequence $\mathcal{C}$ at boundaries $|w_1|$ and $|w_1|+j_2-j_1$ and all other crossing sequences are generated at most 3 times at boundaries $1,2\ldots(|w_1|+j_2-j_1+|w_2|)$: once left from $w(j_1,j_2)$, once at the boundaries of $w(j_1,j_2)$ and once right from $w(j_1,j_2)$. Using Lemma 5.2.3 twice, we see that $|w_1w_2| \leq \ell$ and $|w_1w(j_1,j_2)w_2| \leq r$, which implies $\mathcal{C} \in \mathcal{S}$ and $|w(j_1,j_2)| \leq r$. This contradicts b) because $t_M(w(j_1,j_2),\mathcal{C}) > C|w(j_1,j_2)|$.

As we stated in the above outline, our first goal is to find boundaries j_1 and j_2. From a) it follows that $n > \ell = D+4\cdot 2q^C$, so by Lemma 5.2.3 there exist at least three identical crossing sequences produced by M on the input w on the computation ζ after t steps at the boundaries $1,2\ldots n$. Let these crossing sequences be generated at boundaries $i_1 < i_2 < i_3$ (see Figure 5.4). Because $\mathcal{C}^t_{i_1}(M,\zeta,w)$ and $\mathcal{C}^t_{i_3}(M,\zeta,w)$ are of equal length, the head of M is, before the $(t+1)$st step of the computation ζ, left of the boundary i_1 or right of the boundary i_3. Without the loss of generality we can assume that the head is right from i_3 (if not, we can rename $i_1 = i_2$ and $i_2 = i_3$ and continue with the proof). Thus, no crossing sequence at the boundaries $i_1,(i_1+1)\ldots i_2$ changes in the $(t+1)$st step of the computation ζ. Let $i_1 \leq j_1 < j_2 \leq i_2$ be closest boundaries such that

$\mathcal{C}_{j_1}^{t+1}(M,\zeta,w) = \mathcal{C}_{j_2}^{t+1}(M,\zeta,w)$. Then the crossing sequences $\mathcal{C}_j^t(M,\zeta,w)$, for $j_1 \leq j < j_2$, are pairwise distinct and do not change in the $(t+1)$st step of the computation ζ.

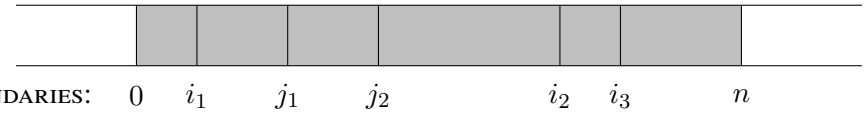


Figure 5.4: Finding boundaries j_1 and j_2. The shaded area represents the input w. First, we find boundaries i_1, i_2 and i_3 at which the same crossing sequence is generated after t steps of the computation ζ. Because the crossing sequences generated at the boundaries i_1, i_2 and i_3 are of the same length, after t steps of the computation ζ the head of M is on some cell left of the boundary i_1 or on some cell right of the boundary i_3, hence either the crossing sequences generated at the boundaries between (and including) i_1 and i_2 remain intact in the $(t+1)$st step of the computation ζ, either the crossing sequences generated at the boundaries between (and including) i_2 and i_3 remain intact in the $(t+1)$st step of the computation ζ. Without loss of generality we may assume that the former holds. We choose $i_1 \leq j_1 < j_2 \leq i_2$ to be closest boundaries such that $\mathcal{C}_{j_1}^{t+1}(M,\zeta,w) = \mathcal{C}_{j_2}^{t+1}(M,\zeta,w)$.

Let ζ_1 be the computation on part $w(j_1,j_2)$ with frontier crossing sequence $\mathcal{C}$ that corresponds to ζ and let ζ_2 be a computation on input $w(0,j_1)w(j_2,n)$ such that its first $t+1-|\zeta_1|$ steps correspond to the first $t+1$ steps of ζ. Because the input $w(0,j_1)w(j_2,n)$ is strictly shorter than n, M makes at most $C(|w(0,j_1)|+|w(j_2,n)|)+D$ steps on any computation on this input, thus

$$\begin{aligned} t+1-|\zeta_1| &\leq |\zeta_2| \\ &\leq C(|w(0,j_1)|+|w(j_2,n)|)+D. \end{aligned}$$

From $t = Cn+D$ and $n = |w(0,j_1)|+|w(j_2,n)|+j_2-j_1$ it follows that

$$\begin{aligned} |\zeta_1| &\geq t+1-C(|w(0,j_1)|+|w(j_2,n)|)-D \\ &= C(j_2-j_1)+1, \end{aligned}$$

thus $t_M(w(j_1,j_2),\mathcal{C}) > C|w(j_1,j_2)|$.

Next, we will cut out some pieces of w to eliminate as many redundant parts as possible (if they exist), while leaving the part of w between the boundaries j_1 and j_2 intact. Redundant parts are those where identical crossing sequences are generated on the computation ζ after t steps. We will cut out parts recursively and the result will not necessarily be unique (see Figure 5.5).

Suppose that $\mathcal{C}_k^t(M,\zeta,w) = \mathcal{C}_l^t(M,\zeta,w)$ for $1 \leq k < l \leq j_1$ or $j_2 \leq k < l \leq n$. Cut out the part of w between the kth and lth boundary. Let w' be the new input. Let the boundaries j_1' and j_2' for the input w' correspond to the boundaries j_1 and j_2 for the input w. Let ζ' be a computation on w' that corresponds to ζ (at least for the first t steps of ζ) and let t' be the step in the computation ζ' that corresponds to the step t of the computation ζ. Now recursively find new k and l. The recursion ends when there are no k,l to be found.

From the recursion it is clear that at the end we will get an input for M of the form $w_0 = w_1w(j_1,j_2)w_2$, where $|w_1| \geq 1$. Let ζ_0 be a computation that corresponds to ζ after the cutting out (at least for the first t steps of ζ) and let t_0 be the step in ζ_0 that corresponds to t. If we denote $n_0 = |w_0|$, then it holds $t_0 \leq Cn_0+D$ because either there was nothing to remove and $w_0 = w$, $t_0 = t$, or w_0 is a shorter input than w and $t_0 \leq Cn_0+D$ must hold by the minimality in the

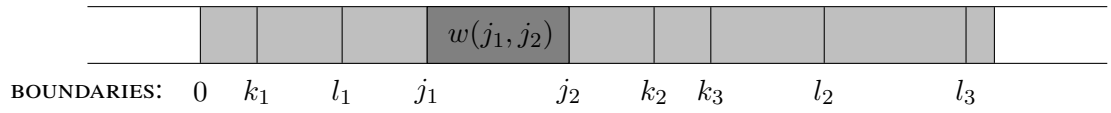


Figure 5.5: Cutting out parts of w to the left and to the right of $w(j_1, j_2)$. If M on the input w (shaded) on the computation ζ after t steps produces the same crossing sequence at boundaries k_1 and l_1, then we can cut out $w(k_1, l_1)$. The same holds also for pairs (k_2, l_2) and (k_3, l_3). What is more, we can cut out both $w(k_1, l_1)$ and $w(k_2, l_2)$ if $\mathcal{C}^t_{k_1}(M, \zeta, w) = \mathcal{C}^t_{l_1}(M, \zeta, w)$ and $\mathcal{C}^t_{k_2}(M, \zeta, w) = \mathcal{C}^t_{l_2}(M, \zeta, w)$. However, we cannot cut out both $w(k_2, l_2)$ and $w(k_3, l_3)$ because they overlap, and we may get a different outcome if we cut out $w(k_2, l_2)$ or $w(k_3, l_3)$.

definition of w. From the construction it is clear that M on input w_0 on computation ζ_0 after t_0 steps generates the crossing sequence $\mathcal{C}$ at the boundaries $|w_1|$ and $|w_1| + j_2 - j_1$. What is more, the crossing sequences at the boundaries $1, 2 \dots |w_1|$ are pairwise distinct. The same is true for the crossing sequences at the boundaries $(|w_1| + 1), (|w_1| + 2) \dots (|w_1| + j_2 - j_1)$ and the crossing sequences at the boundaries $(|w_1| + j_2 - j_1), (|w_1| + j_2 - j_1 + 1) \dots n_0$. Because $t_0 \leq Cn_0 + D$, we get that $n_0 \leq D + 4 \cdot 3q^C = r$ by Lemma 5.2.3, hence $|w(j_1, j_2)| \leq r$.

Denote $\tilde{w} = w_1 w_2$ and $\tilde{n} = |w_1| + |w_2|$. Let the computation $\tilde{\zeta}$ on $\tilde{w}$ be a computation that corresponds to ζ_0 (at least for the first t_0 steps of ζ_0) and let $\tilde{t}$ be the time step of $\tilde{\zeta}$ that corresponds to the time step t_0 of ζ_0. Because $\tilde{n} < n_0 \leq n$ and because w is a shortest input for M that violates the bound $Cn + D$, M makes at most $C\tilde{n} + D$ steps on any computation on the input $\tilde{w}$, thus also on the computation $\tilde{\zeta}$. Note that no three crossing sequences from $\{\mathcal{C}^{\tilde{t}}_i(M, \tilde{\zeta}, \tilde{w}); 1 \leq i \leq \tilde{n}\}$ are identical, thus by Lemma 5.2.3, $\tilde{n} \leq D + 4 \cdot 2q^C = \ell$. Because $\mathcal{C}^{\tilde{t}}_{|w_1|}(M, \tilde{\zeta}, \tilde{w}) = \mathcal{C}$, it follows that $\mathcal{C} \in \mathcal{S}$, which together with $|w(j_1, j_2)| \leq r$ and $t_M(w(j_1, j_2), \mathcal{C}) > C|w(j_1, j_2)|$ contradicts b).□

5.2.3 Supplementary Results to the Compactness Theorem

In this section we prove several corollaries of the results in the previous section, Section 5.2.2, which supplement the compactness theorem. For all positive integers C and D, we will show that a one-tape NTM runs in time $Cn + D$ if and only if it runs in that time on short inputs. We will use this to construct an algorithm that takes integers $C, D \in \mathbb{N}$ and a one-tape NTM M as inputs and if M runs in time $Cn + D$, returns an equivalent finite automaton. We will also give some results that hold if we have a q-state one-tape NTM M that runs in time $Cn + D$, most notable an explicit upper bound on the length of crossing sequences that M can produce and a description of a structure on Σ^* that is induced by M.

The following corollary reveals why we use the name *compactness* theorem. It is because it implies that a fixed linear running time of a Turing machine has to be verified only on finitely many inputs.

Corollary 5.2.5. *For positive integers C and D and for the polynomial*

$$p(C, D) = 1 + C + D + CD + D^2 + CD^2,$$

a one-tape q-state Turing machine runs in time $Cn + D$ if and only if, for each input of length $n \leq 144p(C, D)q^{2C}$, it makes at most $Cn + D$ steps.

Proof. The only if part is trivial and the if part follows by Theorem 5.2.1 and Lemma 5.2.2. □

Corollary 5.2.6. *Let all assumptions be as in Theorem 5.2.1. If the NTM M runs in time $Cn + D$, then it only produces crossing sequences of length at most $C\ell + D$.*

Proof. Let $\mathcal{C}$ be a crossing sequence produced by M on a computation ζ on the input w. Let us use induction on the length of $|w|$ to prove that $|\mathcal{C}| \leq C\ell + D$. If $|w| \leq \ell$, then $|\mathcal{C}| \leq C|w| + D$ and hence $|\mathcal{C}| \leq C\ell + D$. If $|w| > \ell$ then by Lemma 5.2.3 there exist at least three boundaries from $\{1, 2 \dots |w|\}$ such that M on the computation ζ produces the same crossing sequence at them. We can choose two of these three boundaries $i < j$, such that the crossing sequence $\mathcal{C}$ is produced at some boundary that is not strictly between them. If we cut out $w(i, j)$ from w, then the computation that corresponds to ζ on the input $w(0, i)w(j, |w|)$ produces the crossing sequence $\mathcal{C}$ at some boundary. By the induction hypothesis, $|\mathcal{C}| \leq C\ell + D$. □

Corollary 5.2.7. *Let all assumptions be as in Theorem 5.2.1. If the NTM M runs in time $Cn + D$ and it produces a crossing sequence $\mathcal{C}$ on a computation ζ on an input w at some boundary $1, 2 \dots |w|$, then $\mathcal{C} \in \mathcal{S}$.*

To tell the corollary in other words, if the NTM M runs in time $Cn+D$, then $\mathcal{S} = \bigcup_{n=1}^{\infty} \mathcal{S}_n(M)$.

Proof. We use induction on the length $|w|$. If $|w| \leq \ell$, the corollary follows by the definition of the set $\mathcal{S}$. If $|w| > \ell$, then by Lemma 5.2.3 there exist at least three boundaries from $\{1, 2 \dots |w|\}$ such that M on the computation ζ produces the same crossing sequence at them. We can choose two of these three boundaries $i < j$, such that the crossing sequence $\mathcal{C}$ is produced at some boundary from $\{1, 2 \dots |w|\}$ that is not strictly between them. If we cut out $w(i, j)$ from w, then the computation that corresponds to ζ on the input $w(0, i)w(j, |w|)$ produces the crossing sequence $\mathcal{C}$ at some boundary from $\{1, 2 \dots |w(0, i)w(j, |w|)|\}$. By the induction hypothesis, $\mathcal{C} \in \mathcal{S}$. □

Corollary 5.2.8. *Let all assumptions be as in Theorem 5.2.1. If the NTM M runs in time $Cn + D$ then, for every string $w \in \Sigma^*$, for every computation ζ of M on the input w and for every two indices $1 \leq i < j \leq |w|$ such that the crossing sequences produced by M on the computation ζ at the boundaries $i, i + 1 \dots (j - 1)$ are pairwise distinct, it holds $j - i \leq r$.*

Proof. We use induction on the length $|w|$. If $|w| \leq r$, the corollary holds. If $|w| > r$ then by Lemma 5.2.3 there exist at least four boundaries $i_1 < i_2 < i_3 < i_4$ from $\{1, 2 \dots |w|\}$ at which the same crossing sequence is produced by M on the computation ζ on the input w. Because the crossing sequences produced at the boundaries $i, i + 1 \dots (j - 1)$ are pairwise distinct, it either holds $i_2 \leq i$ or $i_3 \geq j$. Hence, we can cut out one of the substrings $w(i_1, i_2)$ or $w(i_3, i_4)$ from w such that the substring $w(i, j)$ remains intact. Then M on the new input on the computation that corresponds to ζ produces pairwise distinct crossing sequences at the boundaries corresponding to $i, i + 1 \dots (j - 1)$. Because the new input is shorter than w, it holds $j - i \leq r$ by induction. □

The above corollary has an interesting implication. Recall Proposition 5.1.7 which stated that every one-tape NTM that runs in time $o(n \log n)$ decides a regular language. Now suppose that an NTM M with q states runs in time $Cn + D$ and it decides a language L. In the proof of Proposition 5.1.7 we explained how an NFA $\tilde{M}$ can be defined so that it will accepted the language L. The states of $\tilde{M}$ were all possible crossing sequences that M can produce, hence $\tilde{M}$ could have up to $q^{\Omega(q^C)}$ states. What Corollary 5.2.8 tells is that every (non-self-intersecting) path in the graph of $\tilde{M}$ is at most $r = O(q^C)$ states long.

The next simple corollary induces a structure on Σ^* that is related to some one-tape linear-time NTM.

Corollary 5.2.9. *Let all assumptions be as in Theorem 5.2.1. If the NTM M runs in time $Cn + D$, then for every string $w \in \Sigma^*$ and for every computation ζ of M on the input w, at least one of the following holds:*

1. *$|w| \leq \ell$ or*
2. *there exist indices $1 \leq i < j \leq |w|$ such that $j - i \leq r$ and M on the computation ζ on the input w produces the same crossing sequence at boundaries i and j.*

This corollary could be rephrased as follows. If a one-tape NTM M runs in time $Cn + D$, then we can construct every string from Σ^* the following way. Begin with some string $w_0 \in \Sigma^*$ of length at most ℓ and a computation ζ_0 of M on w_0. Next, choose a boundary from $\{1, 2 \dots |w_0|\}$ and insert some part $y_0 \in \Sigma^*$ of length at most r at this boundary, where $t_M(y_0, \mathcal{C}_0) \geq 0$ and $\mathcal{C}_0$ is the crossing sequence produced by M on ζ_0 at the chosen boundary. Let w_1 be the obtained string and let ζ_1 be a computation of M on w_1 that computes as ζ_0 on the parts of w_0. Because $t_M(y_0, \mathcal{C}_0) \geq 0$, the computation ζ_1 exists. Next, choose a boundary from $\{1, 2 \dots |w_1|\}$ and insert some part $y_1 \in \Sigma^*$ at this boundary, where $t_M(y_1, \mathcal{C}_1) \geq 0$ and $\mathcal{C}_1$ is the crossing sequence produced by M on ζ_1 at the chosen boundary. Continue in this a way for finitely many steps.

Proof. Suppose $|w| > \ell$. By Lemma 5.2.3 it holds that M on the computation ζ produces some crossing sequence at least at three boundaries from $\{1, 2 \dots |w|\}$. Let $i < j$ be such two boundaries at which the same crossing sequence $\mathcal{C}$ is produced and the crossing sequences produced at the boundaries $i, i + 1 \dots (j - 1)$ are pairwise distinct. By Corollary 5.2.8 it holds that $j - i \leq r$. □

We finish this chapter with an algorithmic adornment of Proposition 5.1.7.

Theorem 5.2.10. *There exists an algorithm that takes integers $C, D \in \mathbb{N}$ and a one-tape NTM M as inputs and, if M runs in time $Cn + D$, it returns an equivalent NFA.*

By Proposition 2.2.2, the algorithm could as well return an equivalent DFA. This result relativizes, however the algorithm has to use the same oracle as the input NOTMs. To prove the relativized version, note that, given a one-tape NOTM that runs in time $Cn + D$, we can construct an equivalent one-tape NTM that runs in time $\tilde{C}n + \tilde{D}$ for some $\tilde{C}, \tilde{D} \in \mathbb{N}$ using ideas from the proof of Corollary 5.1.5 (see also the paragraph after the corollary).

Proof. Let integers $C, D \in \mathbb{N}$ and a one-tape NTM M with q states be given. We can use Corollary 5.2.5 to verify whether M runs in time $Cn + D$. If this is the case, we can define an equivalent NFA $\tilde{M} = (Q, \Sigma, \delta, q_s, F)$ as in the proof of Proposition 5.1.7, only that this time we do it constructively.

We may assume that, for all n, M on inputs of length n in the last step of each of its computations crosses the boundary n of its tape (in one of its halting states). For $\ell = D + 8q^C$, let S be the set of all the crossing sequences up to the length $C\ell + D$. Note that the set of crossing sequences that can be produced by M is a subset of S by Corollary 5.2.6. Let $F \subseteq S$ be the subset of S of such crossing sequences that are formed by an NTM M on some input w on some accepting computation at the boundary $|w|$. Note that by Corollary 5.2.7 we can search for such crossing sequences by only considering inputs of length at most ℓ. Define $Q = S \cup \{q_s\}$ for some new state q_s and, if M accepts the input ε, add q_s to F. The definition of δ is the following.

- For each $\mathcal{C} \in S$ and for each $a \in \Sigma$, there is an edge from q_s to $\mathcal{C}$ with the weight a if there exists some input w of M that begins with the symbol a and a computation ζ of M on w

such that M produces the crossing sequence $\mathcal{C}$ at the boundary 1 on the computation ζ. By Corollary 5.2.9 it is enough to consider only inputs of length at most ℓ, because longer inputs can be appropriately shortened.

- For each $\mathcal{C}_1, \mathcal{C}_2 \in S$ and for each $a \in \Sigma$, there is an edge from $\mathcal{C}_1$ to $\mathcal{C}_2$ if there exists a valid computation of M on the part a with frontier crossing sequences $\mathcal{C}_1$ and $\mathcal{C}_2$. Although such a computation was formally defined only for $\mathcal{C}_1 = \mathcal{C}_2$, the definition for $\mathcal{C}_1 \neq \mathcal{C}_2$ is intuitive and analogous.

Clearly, $\tilde{M}$ can be constructed from C, D and M.

First suppose that M accepts an input w. If $|w| = 0$ then $\tilde{M}$ accepts w, else there exists an accepting computation ζ of M on w that produces crossing sequences $\mathcal{C}_1, \mathcal{C}_2 \ldots \mathcal{C}_{|w|}$ at the boundaries $1, 2 \ldots |w|$. By the comments in the definition of $\tilde{M}$ there exists a computational path $q_s, \mathcal{C}_1, \mathcal{C}_2 \ldots \mathcal{C}_{|w|}$ for $\tilde{M}$ on the input w where $\mathcal{C}_{|w|} \in F$ which means that $\tilde{M}$ accepts w.

Now suppose that $\tilde{M}$ accepts an input w. If $|w| = 0$ then M accepts w, else there exists an accepting computational path $q_s, \mathcal{C}_1, \mathcal{C}_2 \ldots \mathcal{C}_{|w|}$ for $\tilde{M}$ on the input w. We claim that we can build a computation of M on w that produces the crossing sequences $\mathcal{C}_1, \mathcal{C}_2 \ldots \mathcal{C}_{|w|}$ at the boundaries $1, 2 \ldots |w|$. For each non-first symbol a of w, we know that there exists a computation on the part a that produces the desired left and right crossing sequence. We also know that there exists an input w_0 that begins with the same symbol as w and a computation ζ_0 of M on w_0 such that M forms the crossing sequence $\mathcal{C}_1$ at the boundary 1. Furthermore, because $\mathcal{C}_{|w|} \in F$, we know that there exists an input w_1 and an accepting computation ζ_1 of M on w_1 that produces the crossing sequence $\mathcal{C}_{|w|}$ at the boundary $|w_1|$. Now the computation of M on w that produces the crossing sequences $\mathcal{C}_1, \mathcal{C}_2 \ldots \mathcal{C}_{|w|}$ at the boundaries $1, 2 \ldots |w|$ is the following: on the first symbol and left of this symbol it computes like ζ_0, right from the last symbol it computes as ζ_1 and above each of the non-first symbols of w it computes as the desired computation. Hence, M accepts w, which implies that M and $\tilde{M}$ accept the same language. □

Chapter 6

Verifying Time Complexity of Turing Machines

This chapter contains the main results of the author [10, 11]. For a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we show the following in the first part of the chapter, Section 6.1.

- The problem of whether a given multi-tape Turing machine runs in time $T(n)$ is undecidable if and only if, for all $n \in \mathbb{N}$, $T(n) \geq n + 1$ (Theorem 6.1.3).
- The problem of whether a given one-tape Turing machine runs in time $T(n)$ is undecidable if $T(n) = \Omega(n \log n)$ and $T(n) \geq n + 1$ for all $n \in \mathbb{N}$ (Theorem 6.1.5).
- The problem of whether a given one-tape Turing machine runs in time $T(n)$ is decidable if T is "nice" and $T(n) = \mathrm{o}(n \log n)$ (Theorem 6.1.10).
- The problem of whether a given one-tape or multi-tape Turing machine runs in time $\mathrm{O}(T(n))$ is undecidable for all reasonable functions T (Theorem 6.1.6).

All these results hold for deterministic as well as non-deterministic Turing machines. In the second part of the chapter, Section 6.2, we prove Theorem 1.2.1, which is stated in the introduction. It characterizes computational complexity of the problems of verifying whether a given one-tape Turing machine runs in time $Cn + D$, for parameters $C, D \in \mathbb{N}$. In Section 6.2.5 we argue that our techiques relativize.

6.1 Decidability Results

First, we define the problems that will be in our interest in this section. For a class of functions $\mathcal{F} \subseteq \{T : \mathbb{N} \to \mathbb{R}_{\geq 0}\}$, define the problem $\textsc{Halt}_{\mathcal{F}}$ as

Given a multi-tape NTM, does it run in time $T(n)$ for some $T \in \mathcal{F}$?

and the problem D-$\textsc{Halt}_{\mathcal{F}}$ as

Given a multi-tape DTM, does it run in time $T(n)$ for some $T \in \mathcal{F}$?

The problems $\textsc{Halt}^1_{\mathcal{F}}$ and D-$\textsc{Halt}^1_{\mathcal{F}}$ are defined analogously for one-tape Turing machines as inputs.

If $\mathcal{F}$ has only one element, we write $\text{HALT}_{\{T\}} = \text{HALT}_{T(n)}$, thus $\text{HALT}_{T(n)}$ is the problem of whether a given multi-tape NTM runs in time $T(n)$. If $\mathcal{F}$ is the class of polynomials, we write $\text{HALT}_{\mathcal{F}} = \text{HALT}_{\mathsf{P}}$, thus HALT_{P} is the problem of whether a given multi-tape NTM runs in polynomial time. For a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, if $\mathcal{F} = \{f : \mathbb{N} \to \mathbb{R}_{\geq 0}; f(n) = \text{O}(T(n))\}$, we write $\text{HALT}_{\mathcal{F}} = \text{HALT}_{\text{O}(T(n))}$, thus $\text{HALT}_{\text{O}(T(n))}$ is the problem of whether a given multi-tape NTM runs in $\text{O}(T(n))$ time.

Problems $\text{HALT}^1_{T(n)}$, $\text{HALT}^1_{\mathsf{P}}$ and $\text{HALT}^1_{\text{O}(T(n))}$ are defined similarly for one-tape NTMs and the problems D-$\text{HALT}_{T(n)}$, D-$\text{HALT}^1_{T(n)}$, D-HALT_{P}, D-$\text{HALT}^1_{\mathsf{P}}$, D-$\text{HALT}_{\text{O}(T(n))}$ and D-$\text{HALT}^1_{\text{O}(T(n))}$ are defined similarly for DTMs. We will prove undecidability results only for the problems involving DTMs and decidability results only for the problems involving NTMs. This implies that all the results in this section hold for DTMs as well as for NTMs.

6.1.1 Folkloric Results and Extended Considerations

In this section we prove that all the "basic" problems D-$\text{HALT}^1_{\mathcal{F}}$ are undecidable (hence also all basic problems D-$\text{HALT}_{\mathcal{F}}$ are undecidable), we give a tight bound on the function T for which the problems $\text{HALT}_{T(n)}$ and D-$\text{HALT}_{T(n)}$ are decidable and we prove undecidability of D-$\text{HALT}^1_{T(n)}$ for all functions $T(n) = \Omega(n \log n)$ with $T(n) \geq n + 1$.

Let us begin with an easy positive result. It gives a reason for why we need the technical condition $T(n) \geq n + 1$ when proving undecidability results.

Lemma 6.1.1. *Let $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function such that, for some $n_0 \in \mathbb{N}$, it holds $T(n_0) < n_0 + 1$. Then the problem $\text{HALT}_{T(n)}$ is decidable.*

Proof. Let n_0 be such that $T(n_0) < n_0 + 1$ and let a multi-tape NTM M be given. We will describe an algorithm which decides whether M runs in time $T(n)$, thus proving decidability of $\text{HALT}_{T(n)}$:

- First, check if the length of each computation of M on inputs of lengths $n \leq n_0$ is at most $T(n)$. If not, **reject**. Else, let T_w be the length of a longest computation of M on inputs of length n_0 and suppose this maximum is achieved on an input w.
- If $T_w \leq T(n)$ for all $n > n_0$, **accept**. Else, **reject**.

To prove finiteness and correctness of the algorithm, note that if M makes at most $T(n_0)$ steps on all computations on inputs of size n_0, then by Lemma 3.3.1 M never reads the (n_0+1)st symbol of any input. In this case M makes at most T_w steps on each computation on inputs of length more than n_0. Moreover, for each $n \geq n_0$, there exists an input of length n on which M makes exactly T_w steps on some computation (all inputs that begin with w are such). There are only finitely many possibilities for T_w because $T_w \leq T(n_0)$, thus the last line of the algorithm can be done in constant time. □

The following lemma proves the converse of Lemma 6.1.1.

Lemma 6.1.2. *Let $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function such that, for all $n \in \mathbb{N}$, it holds $T(n) \geq n + 1$. Then the problem D-$\text{HALT}_{T(n)}$ is undecidable.*

Proof. We will describe a reduction of the complement of the problem D-$\text{HALT}^1_{\varepsilon}$ to the problem D-$\text{HALT}_{T(n)}$. Because the problem D-$\text{HALT}^1_{\varepsilon}$ is undecidable by Lemma 4.1.2, this implies that D-$\text{HALT}_{T(n)}$ is also undecidable. The reduction is as follows.

Given a one-tape DTM H, construct a 2-tape DTM $\tilde{H}$ that on the input tape always moves its head to the right and halts when it reaches a blank symbol and on the work tape it simulates H on input ε. If H halts, $\tilde{H}$ starts an infinite loop (and does not halt when it reaches a blank symbol on the input tape). It is clear that $\tilde{H}$ runs in time $T(n)$ if and only if H does not halt on the empty input. □

Combining Lemma 6.1.1 and Lemma 6.1.2 we get a tight bound on a function T for when the problems $\text{HALT}_{T(n)}$ and D-$\text{HALT}_{T(n)}$ are decidable.

Theorem 6.1.3. *For a function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$, the problem $\text{HALT}_{T(n)}$ is undecidable if and only if the problem D-$\text{HALT}_{T(n)}$ is undecidable if and only if, for all $n \in \mathbb{N}$, $T(n) \geq n + 1$.*

Proof. Combine Lemma 6.1.1 and Lemma 6.1.2. □

There is no such a sharp bound for one-tape Turing machines as it is $n+1$ for multi-tape. As discussed in the introduction, the decidability of D-$\text{HALT}^1_{T(n)}$ changes roughly at $T(n) = \Theta(n \log n)$ and the next lemma will be the main tool in proving undecidability of D-$\text{HALT}^1_{T(n)}$ for $T(n) = \Omega(n \log n)$. In the proof of the lemma we will see how timekeeping and simulating another Turing machine can be done fast on one-tape Turing machines.

Lemma 6.1.4. *Let $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function such that $T(n) = \Omega(n \log n)$ and, for all $n \in \mathbb{N}$, it holds $T(n) \geq n + 1$. Then there exists an algorithm that takes as input a one-tape DTM H and returns a one-tape DTM $\tilde{H}$ such that*

$$H(\varepsilon) = \infty \iff \tilde{H} \text{ runs in time } T(n) \iff \tilde{H} \text{ always halts.}$$

In the statement of the lemma we used the notation $H(\varepsilon) = \infty$ to denote that H does not halt on input ε.

Proof. Because $T(n) = \Omega(n \log n)$, there exist constants $C, n_0 \in \mathbb{N}$ such that $6 \leq C \leq n_0$ and, for all $n \geq n_0$, it holds

$$T(n) \geq 3n \log_C n + 6n + 1.$$

For an arbitrary one-tape DTM H with a tape alphabet Γ, let us describe a new one-tape DTM $\tilde{H}$:

- The tape alphabet of $\tilde{H}$ is $\Gamma \bigcup \Gamma' \bigcup \{\&, \#\}$, where $\Gamma' = \{a'; a \in \Gamma\}$. Without loss of generality we can assume that the sets $\{\&, \#\}$, Γ and Γ' are pairwise disjoint.
- On an input w of length n, $\tilde{H}$ first reads the input and if $n < n_0$, accepts in $n + 1$ steps. If $n \geq n_0$, then $\tilde{H}$ overwrites the input with

  $$\#1^{n-1}\#,$$

  leaving the head above the last written one. This all can be done in $\mathbf{n + 1}$ **steps**.
- $\tilde{H}$ will never again write or overwrite the symbol $\#$, which will serve as the left and the right border for the head. From now on, the head will move exactly from the right $\#$ to the left $\#$ and vice versa. Thus we only need to count how many times the head will pass from one $\#$ to another and multiply the result with n to get how many steps were done. A transition of the head form one $\#$ to another will be called a *(head) pass*.

- For $m = \lceil \log_C n \rceil$, $\tilde{H}$ can transform its tape into

$$\#\textvisiblespace^m\textvisiblespace'\textvisiblespace\&^{n-3-m}\#$$

in the next $\mathbf{2m + 2}$ **head passes**[1].

This can be done if on each pass to the right, $\tilde{H}$ turns $C-1$ successive 1s into symbols $\&$, leaves the next symbol 1, turns the next $C-1$ successive 1s into $\&$s ... until it comes to $\#$. Also when passing to the right, it adds another blank symbol after the rightmost previously written blank symbol. When passing to the left it changes nothing. When there are no more 1s, it makes two additional passes to write $\textvisiblespace'\textvisiblespace$ after the previously written blank symbols.

Until this point we did not need any information about H.

- The tape is now prepared for the simulation of H on input ε. The symbols from Γ tell us how the tape of H looks like and the (only) symbol from Γ' tells us the current head position in H. Because $\tilde{H}$ will simulate at most m steps of H, it will need at most m tape cells to the left of $\textvisiblespace'$. $\tilde{H}$ will also not run out of blank symbols to the right of the symbol from Γ', because during the simulation the symbols $\&$ will gradually get replaced by blank symbols.

- The simulation goes as follows: in each pass, $\tilde{H}$ turns $C-1$ successive $\&$s into blank symbols, leaves out the next symbol $\&$, turns the next $C-1$ successive $\&$s into blank symbols ... and when the head comes to the "simulation part" of the tape, it simulates one step of H if and only if the head of H would move in the same direction as the head of $\tilde{H}$ is currently moving. Thus $\tilde{H}$ simulates at least one and at most two steps of H in two head passes.

- If $\tilde{H}$ runs out of $\&$s on its tape before the simulation of H has finished, it halts (e.g. goes in q_{ACC}). Else, $\tilde{H}$ starts an infinite loop so that $\tilde{H}(w) = \infty$.

- From $6 \leq C \leq n_0 \leq n$ it follows that $n - 3 - m \geq C^{m-2}$, so $\tilde{H}$ needs at least $m-1$ head passes to erase all $\&$s.

  Thus if $\tilde{H}$ halts, this means that H does not complete its computation on input ε in $\lfloor \frac{m-1}{2} \rfloor$ steps. In this case $\tilde{H}$ makes at most $\mathbf{m}$ **head passes** from the beginning of the simulation until it halts and thus makes at most $T(n)$ steps altogether on the input w.

  If $\tilde{H}$ does not halt, this means that H halts on input ε.

Note that since $m = \Omega(\log n) \neq \mathrm{O}(1)$ it holds that

$$H(\varepsilon) = \infty \iff \tilde{H} \text{ runs in time } T(n) \iff \tilde{H} \text{ always halts.}$$

To sum up, we have described a desired one-tape DTM $\tilde{H}$ and it is clear from the description that there exists an algorithm that constructs it from H. □

The following theorem is a direct corollary of Lemma 6.1.4.

Theorem 6.1.5. *Let $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function such that $T(n) = \Omega(n \log n)$ and, for all $n \in \mathbb{N}$, it holds $T(n) \geq n + 1$. Then the problem* D-$\textsc{Halt}^1_{T(n)}$ *is undecidable.*

Proof. Given a one-tape DTM H, let $\tilde{H}$ be a one-tape DTM that runs in time $T(n)$ if and only if $H(\varepsilon) = \infty$. By Lemma 6.1.4 we can construct it from H, thus we can reduce the complement of the problem $\textsc{Halt}^1_\varepsilon$ to the problem $\textsc{Halt}^1_{T(n)}$. □

[1] Note that $C^{m-1} \leq n - 1 < C^m$.

For the last thing in this section, we prove that all "basic" problems D-$\text{HALT}^1_{\mathcal{F}}$ (and hence also D-$\text{HALT}_{\mathcal{F}}$) are undecidable.

Theorem 6.1.6. *Let $\mathcal{F} \subseteq \{f : \mathbb{N} \to \mathbb{R}_{\geq 0}\}$ be a class of functions that contains arbitrarily large constants. Then the problem* D-$\text{HALT}^1_{\mathcal{F}}$ *is undecidable.*

Proof. Define the class $F = \{T \in \mathcal{F},\ T(n) \geq n + 1 \text{ for all } n\}$ and consider two separate cases:

1. If for all functions $T \in F$, it holds $T(n) \neq \Omega(n \log n)$, then the following is a reduction of D-$\text{HALT}^1_{\varepsilon}$ to D-$\text{HALT}^1_{\mathcal{F}}$. Given a one-tape DTM H, construct a one-tape DTM $\tilde{H}$ that works as follows on an input w:

   - $\tilde{H}$ simulates $|w|$ steps of H on input ε.
   - If H halts in less than $|w|$ steps, then $\tilde{H}$ also halts and does not make any additional steps.
   - Else, $\tilde{H}$ makes at least additional $|w| \log |w|$ arbitrary steps and halts.

   This can easily be done, for example, by using the input portion of the tape only for counting steps and the left portion of the tape for simulation of (H on input ε). We do not need $\tilde{H}$ to efficiently simulate H, but it is necessary that $\tilde{H}$ runs in constant time if H halts on ε.

   It is clear that H halts on input ε if and only if $\tilde{H}$ runs in constant time, which happens if and only if $\tilde{H}$ runs in time $\tilde{T}(n)$ for some function $\tilde{T}(n)$ that is not $\Omega(n \log n)$.

   Now if $\tilde{H} \in$ D-$\text{HALT}^1_{\mathcal{F}}$, then there exists a function $T \in \mathcal{F}$, such that $\tilde{H}$ runs in time $T(n)$. If $T \in F$, then $T(n) \neq \Omega(n \log n)$, thus H halts on input ε. If $T \notin F$, then by the definition of F there exists n_0 such that $T(n_0) < n_0 + 1$, which by Lemma 3.3.1 implies that $\tilde{H}$ runs in constant time and consequentially H halts on input ε, which implies $H \in \text{HALT}^1_{\varepsilon}$.

   If $\tilde{H} \notin$ D-$\text{HALT}^1_{\mathcal{F}}$, then $\tilde{H}$ does not run in constant time because $\mathcal{F}$ contains arbitrarily large constants. So H does not halt on input ε and hence $H \notin \text{HALT}^1_{\varepsilon}$.

   So we have proven $\tilde{H} \in$ D-$\text{HALT}^1_{\mathcal{F}} \iff H \in$ D-$\text{HALT}^1_{\varepsilon}$.

2. If for some function $T \in F$ it holds $T(n) = \Omega(n \log n)$, then the following is a reduction of the complement of D-$\text{HALT}^1_{\varepsilon}$ to D-$\text{HALT}^1_{\mathcal{F}}$.

   For an arbitrary one-tape DTM H, use Lemma 6.1.4 to construct a one-tape DTM $\tilde{H}$ that runs in time $T(n)$ if and only if $\tilde{H}$ always halts which is if and only if $H(\varepsilon) = \infty$. This implies that $H(\varepsilon) = \infty$ if and only if $\tilde{H}$ runs in time $\tilde{T}(n)$ for some function $\tilde{T} \in \mathcal{F}$. $\square$

Some well known corollaries follow from Theorem 6.1.6, namely that the problems D-HALT_{P}, D-$\text{HALT}^1_{\mathsf{P}}$, D-$\text{HALT}_{\text{O}(T(n))}$ and D-$\text{HALT}^1_{\text{O}(T(n))}$ for $T(n) = \Omega(1)$ are undecidable. Consequentially, the problems D-$\text{HALT}_{\text{O}(1)}$ and D-$\text{HALT}^1_{\text{O}(1)}$ are undecidable.

6.1.2 One-Tape Turing Machines and an $\text{o}(n \log n)$ Time Bound

Until this point we know that, for $T(n) = \Omega(n \log n)$, we can solve $\text{HALT}^1_{T(n)}$ if and only if, for some $n_0 \in \mathbb{N}$, it holds $T(n_0) < n_0 + 1$. It remains to see that $\text{HALT}^1_{T(n)}$ is decidable for all nice functions $T(n) = \text{o}(n \log n)$. But first, we have to define "nice".

For a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we say that f *computably converges to* ∞ if, for each $K \in \mathbb{N}$, we can construct $n_K \in \mathbb{N}$ (i.e. the function $K \mapsto n_K$ is computable) such that, for all $n \geq n_K$, it holds $f(n) \geq K$.

Manageable Functions

We say that a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is *manageable* if there exists a Turing machine that, given $A_0, A_1 \dots A_k \in \mathbb{N}\backslash\{0\}$ and $B_0, B_1 \dots B_k \in \mathbb{N}$, it decides whether the inequality

$$f(A_0 + x_1A_1 + x_2A_2 + \cdots + x_kA_k) < B_0 + x_1B_1 + x_2B_2 + \cdots + x_kB_k$$

holds for some $x_1, x_2 \dots x_k \in \mathbb{N}$.

Note that there are only integers on the right-hand side of the inequality. Thus the following holds.

Lemma 6.1.7. *A function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is manageable if and only if its integer part $\lfloor f \rfloor$ is manageable.* □

The next proposition gives examples of manageable functions.

Proposition 6.1.8. *Let $f : \mathbb{N} \to \mathbb{N}$ be a computable function. If*

- *f is linear (i.e. of the form $Cn + D$) or*
- *$\frac{f(n)}{n}$ computably converges to ∞,*

then f is manageable.

Proof. The case when f is linear is easy and is left for the reader, so suppose that $\frac{f(n)}{n}$ computably converges to ∞. The next algorithm proves manageability of f:

- Let $A_0, A_1 \dots A_k \in \mathbb{N}\backslash\{0\}$ and $B_0, B_1 \dots B_k \in \mathbb{N}$ be given.
- Find $C \in \mathbb{N}$ such that, for all $i = 0, 1 \dots k$, it holds $CA_i \geq B_i$.
- Find n_C such that $f(n) \geq Cn$ for all $n \geq n_C$. This can be done because $\frac{f(n)}{n}$ computably converges to ∞.
- For $i = 1, 2 \dots k$, let $y_i \in \mathbb{N}$ be such that $A_0 + y_iA_i \geq n_C$.

  It follows that the inequality

  $$f(A_0 + x_1A_1 + x_2A_2 + \cdots + x_kA_k) \geq B_0 + x_1B_1 + x_2B_2 + \cdots + x_kB_k$$

  holds for $x_1, x_2 \dots x_k \in \mathbb{N}$ if there exists an index i such that $x_i \geq y_i$.

  Indeed, $x_i \geq y_i$ implies $A_0 + x_1A_1 + x_2A_2 + \cdots + x_kA_k \geq n_C$, which implies $f(A_0 + x_1A_1 + x_2A_2 + \cdots + x_kA_k) \geq C(A_0 + x_1A_1 + x_2A_2 + \cdots + x_kA_k)$.
- Check if the inequality

  $$f(A_0 + x_1A_1 + x_2A_2 + \cdots + x_kA_k) < B_0 + x_1B_1 + x_2B_2 + \cdots + x_kB_k$$

  holds for some non-negative integers $x_1 < y_1, x_2 < y_2 \dots x_k < y_k$. □

We just proved (using also Lemma 6.1.7) that $n,\ 3n+2,\ n\sqrt{\log n},\ n^2,\ 2^n$ are all manageable functions. The next lemma tells us that the integer part of a manageable functions cannot be too complicated.

Lemma 6.1.9. *An integer part $\lfloor f \rfloor$ of a manageable function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is a computable function.*

Proof. For $n \in \mathbb{N}$, the following algorithm computes $\lfloor f \rfloor(n)$:

- If $n = 0$ return $\lfloor f(0) \rfloor$. Else, return the largest i for which $f(n) \geq i$. □

The decidability Result

In this section we prove that the problem $\text{HALT}^1_{T(n)}$ is decidable for all nice functions $T(n) = o(n \log n)$.

Theorem 6.1.10. *For any manageable function $T : \mathbb{N} \to \mathbb{R}_{>0}$, for which the function $\frac{n \log n}{T(n)}$ computably converges to ∞, the problem $\text{HALT}^1_{T(n)}$ is decidable.*

Note that Theorem 6.1.10 tells us that we can solve the problem $\text{HALT}_{(n+1)\sqrt{\log(n+2)}}$ as well as the problems HALT_{Cn+D} for constants $C, D \in \mathbb{N}$. The following lemma makes an introduction to the proof of the theorem.

Lemma 6.1.11. *Let $T : \mathbb{N} \to \mathbb{R}_{>0}$ be a function for which $\lfloor T \rfloor$ is computable and the function*

$$g(n) = \begin{cases} \frac{n \log n}{T(n)} & ; \quad n \geq 2 \\ 1 & ; \quad n = 0, 1 \end{cases}$$

computably converges to ∞. Then given $q \in \mathbb{N}$, we can compute a constant upper bound on the length of the crossing sequences produced by any q-state one-tape NTM that runs in time $T(n)$.

Proof. By Lemma 5.1.3, we only need to construct a constant $c \geq \max\{T(0),\ T(1)\}$ which satisfies the inequality

$$3\frac{qn^{(\log q)/g(n)^{1/2}} - 1}{q - 1} \leq n - 3 - \frac{n}{g(n)^{1/2}} + c\frac{g(n)^{1/2}}{\log n} \tag{6.1}$$

for the given q and all $n \geq 2$. The construction of c can go as follows:

- Use computable convergence of g to find $N \in \mathbb{N}$ such that for all $n \geq N$ it holds $g(n) \geq 4(\log q)^2$. Increase N if necessary so that, for all $n \geq N$, it also holds $\sqrt{n} \leq \frac{1}{12}n$ and $g(n) \geq 16$.

  It is easy to see that Inequality (6.1) holds for all $n \geq N$ independently of the value of $c \geq 0$.

- Use computability of $\lfloor T \rfloor$ to find such $c \in \mathbb{N}$ that Inequality (6.1) holds for $2 \leq n < N$.

  Note that $g(n) \geq \frac{1}{\lfloor T(n) \rfloor + 1}$ for $n \geq 2$.

- Increase c to get $c \geq \max\{T(0),\ T(1)\}$. □

The following proof is simpler than the proof from [10] because we use the compactness theorem.

Proof of Theorem 6.1.10. Let $T : \mathbb{N} \to \mathbb{R}_{>0}$ be a manageable function for which the function $\frac{n \log n}{T(n)}$ computably converges to ∞. Because T is manageable, $\lfloor T \rfloor$ is computable by Lemma 6.1.9. The following algorithm verifies whether a one-tape NTM M with q states runs in time $T(n)$, thus solving $\text{HALT}^1_{T(n)}$.

- Use Lemma 6.1.11 to construct an upper bound $C \in \mathbb{N}$ on the length of the crossing sequences produced by any one-tape Turing machine with q states that runs in time $T(n)$.

- Compute
$$K = \frac{q^{C+1} - 1}{q - 1},$$
which is an upper bound on the number of distinct crossing sequences produced by any one-tape Turing machine with q states that runs in time $T(n)$.

- Define $D = 2KC + C$.

  Note that any one-tape NTM with q states that runs in time $T(n)$ also runs in time $Cn + D$ (see Corollaries 5.1.4 and 5.1.6).

- Use Corollary 5.2.5 to verify whether M runs in time $Cn + D$. If not, **reject**.

- Define $\ell = D + 8q^C$ and $r = D + 12q^C$ and construct the set $\mathcal{S} = \bigcup_{n=1}^{\ell} \mathcal{S}_n(M)$. The notation is the same as in the compactness theorem (Theorem 5.2.1).

- Construct the set
$$X = \big\{(w, \zeta);\ |w| \leq \ell \text{ and } \zeta \text{ is a computation of } M \text{ on the input } w\big\}.$$

- For each crossing sequence $\mathcal{C} \in \mathcal{S}$, construct the set
$$Y_{\mathcal{C}} = \big\{(w, \zeta);\ |w| \leq r \text{ and } \zeta \text{ is a computation of } M \text{ on the part } w \text{ with the frontier crossing sequence } \mathcal{C}\big\}.$$

  By the compactness theorem, the set $Y_{\mathcal{C}}$ can be constructed in finite time because for each $(w, \zeta) \in Y_{\mathcal{C}}$ it holds $|\zeta| \leq C|w|$.

  The main observation (and the key idea of the algorithm) is that every pair (w, ζ) where ζ is a computation of M on an input w can be constructed in the following way: Begin with some pair $(w_0, \zeta_0) \in X$ and insert on appropriate places pairs from sets $Y_{\mathcal{C}}$ (an appropriate place for an element of $Y_{\mathcal{C}}$ is where a crossing sequence $\mathcal{C}$ is generated). This follows by Corollaries 5.2.7 and 5.2.9.

  Hence, we divided the computations of M into finitely many parts and we computed all of them. Now we must only verify whether putting these parts together can cause M to run for too long on some input.

- For each pair $(w_0, \zeta_0) \in X$ and for all the choices of subsets $\big(\tilde{Y}_{\mathcal{C}} \subseteq Y_{\mathcal{C}}\big)_{\mathcal{C} \in \mathcal{S}}$,

  - Verify whether some input $\tilde{w}$ can be constructed together with a computation $\tilde{\zeta}$ of M on $\tilde{w}$ by starting from (w_0, ζ_0) and inserting one by one on appropriate places all the pairs (w, ζ) from all the sets $\tilde{Y}_{\mathcal{C}}$. It is enough to restrict that each pair (w, ζ) from each set $\tilde{Y}_{\mathcal{C}}$ is used exactly once, which gives a finite number of options.
    - If not, continue with the for loop.
      In this case it is impossible to use only and all parts from sets $\tilde{Y}_{\mathcal{C}}$ at once in a construction of a pair $(\tilde{w}, \tilde{\zeta})$ as described above.
  - Use manageability of T to check whether the inequality
$$|\zeta_0| + \sum_{\mathcal{C} \in \mathcal{S}} \sum_{(w,\zeta) \in \tilde{Y}_{\mathcal{C}}} k_{\mathcal{C},(w,\zeta)} |\zeta| \leq T\left(|w_0| + \sum_{\mathcal{C} \in \mathcal{S}} \sum_{(w,\zeta) \in \tilde{Y}_{\mathcal{C}}} k_{\mathcal{C},(w,\zeta)} |w|\right)$$
    holds for all $k_{\mathcal{C},y} \in \mathbb{N} \backslash \{0\}$. If it does not, **reject**.

Note that in the argument of T on the right-hand side of the inequality, we have the length of some string constructed by starting with $(w_0, \zeta_0) \in X$ and inserting $k_{\mathcal{C},(w,\zeta)}$ pairs $(w, \zeta) \in \tilde{Y}_{\mathcal{C}}$ on appropriate places. On the left-hand side we have the length of the corresponding computation of M on such an input. The comments before this step imply that all computations on non-empty inputs are considered this way and the condition before the inequality assures that it is possible to use only and all parts from sets $\tilde{Y}_{\mathcal{C}}$ at once.

- **accept**.

The comments inside the description of the algorithm show its finiteness and correctness. □

6.2 Complexity Results

Because we can essentially only verify time bounds $o(n \log n)$ for a given one-tape Turing machine and because such Turing machines actually run in linear time, linear time bounds are the most natural time bounds for one-tape Turing machines that are algorithmically verifiable. The purpose of this section is to prove Theorem 1.2.1, stated in the introduction, that gives sharp complexity (lower and upper) bounds for the problems HALT^1_{Cn+D} and D-HALT^1_{Cn+D}. Before we go into the proofs, let us give the main proof ideas in two paragraphs.

We use the compactness theorem to prove the upper bound (Proposition 6.2.1). To prove the lower bounds, we make reductions from hard problems whose hardness is proven by diagonalization. The diagonalization in Proposition 6.2.7 (non-deterministic lower bound) is straightforward and the diagonalization in Proposition 6.2.5 (co-non-deterministic lower bound) is implicit in the non-deterministic time hierarchy. The reductions are not so trivial and we describe the main idea in the following paragraph.

Suppose that a one-tape non-deterministic Turing machine M solves a computationally hard problem L. Then, for any input w, we can decide whether $w \in L$ by *first* constructing a one-tape Turing machine M_w that runs in time $Cn + D$ if and only if M rejects w *and then* solving the complement of HALT^1_{Cn+D} for M_w. If we can construct M_w efficiently, then because L is computationally hard we get a complexity lower bound for solving the complement of HALT^1_{Cn+D}. The machine M_w is supposed to simulate M on w, but only for long enough inputs because we do not want to violate the running time $Cn + D$. Hence, on the inputs of length n, M_w will first only measure the input length using at most $(C - 1)n + 1$ steps to assure that n is large enough, and then it will simulate M on w using at most n steps. If M accepts, M_w starts an infinite loop. It turns out that the main challenge is to make M_w effectively measure the length of the input with not too many steps and also not too many states. The latter is important because we do not want the input M_w for HALT^1_{Cn+D} to be blown up too much, so that we can prove better lower bounds. We leave the details for Section 6.2.3. Let us mention also Section 6.2.4, where we argue that our method of measuring the length of the input is optimal, which implies that using our methods, we cannot get much better lower bounds.

6.2.1 Encoding of One-Tape Turing Machines

To simplify things, let us fix a tape alphabet Γ, hence we will actually be analyzing the problems $\text{HALT}^1_{Cn+D}(\Sigma, \Gamma)$. This enables us to have the codes of q-state one-tape Turing machines of length $\Theta(q^2)$. Because q now describes the length of the code up to a constant factor, we can express the

complexity of algorithms with a q-state one-tape NTM (or DTM) as input in terms of q instead of $n = \Theta(q^2)$.

Let us state the properties that should be satisfied by the encoding of one-tape Turing machines.

- Given a code of a q-state one-tape NTM M, a multi-tape NTM can simulate each step of M in $\mathrm{O}(q^2)$ time. Similarly, given a code of a q-state one-tape DTM M, a multi-tape DTM can simulate each step of M in $\mathrm{O}(q^2)$ time.
- A code of a composition of two one-tape Turing machines can be computed in linear time by a multi-tape DTM.
- The code of a q-state one-tape Turing machine has to be of length $\Theta(q^2)$. This is a technical requirement that makes arguments easier and it gives a concrete relation between the number of states of a one-tape Turing machine and the length of its code. We can achieve this because we assumed a fixed input and tape alphabet.

An example of such an encoding is given in Section 3.6. It is clear that we can easily convert any standard code of a one-tape Turing machine to ours and vice versa.

6.2.2 The Upper Bound

Let us define the problem $\textsc{Halt}^1_{\_n+\_}$ as

> Given a one-tape NTM M and integers $C, D \in \mathbb{N}$, does M run in time $Cn + D$?

Hence, the problem $\textsc{Halt}^1_{\_n+\_}$ is the same as the problem $\textsc{Halt}^1_{Cn+D}$, only that C and D are parts of the input. Recall that we use an overline to denote the complement of a problem.

Proposition 6.2.1. *There exists a multi-tape NTM that solves* $\overline{\textsc{Halt}^1_{\_n+\_}}$ *in time* $\mathrm{O}(p(C, D)q^{C+2})$ *for some quadratic polynomial* p*.*

Proof. Let us describe a multi-tape NTM M_{mult} that solves $\overline{\textsc{Halt}^1_{\_n+\_}}$.

- On the input (C, D, M), where M is a q-state one-tape NTM, compute $\ell = D + 8q^C$ and $r = D + 12q^C$.
- Non-deterministically choose an input of length $n \leq \ell$ and simulate a non-deterministically chosen computation of M on it. If M makes more than $Cn + D$ steps, accept.
- Non-deterministically choose an input $w_0 = w_1w_2w_3$ such that $|w_1| \geq 1$, $1 \leq |w_2| \leq r$ and $|w_1| + |w_3| \leq \ell$. Initialize $\mathcal{C}_1$ and $\mathcal{C}_2$ to empty crossing sequences and counters $t_0 = C|w_0| + D$, $t_2 = C|w_2|$.
- Simulate a non-deterministically chosen computation ζ of M on the input w_0. After each simulated step t of M, do:
  - decrease t_0 by one,
  - if the head of M is on some cell $|w_1| \leq i < |w_1| + |w_2|$, decrease t_2 by one,
  - update the crossing sequences $\mathcal{C}_1 = \mathcal{C}^t_{|w_1|}(M, \zeta, w_0)$ and $\mathcal{C}_2 = \mathcal{C}^t_{|w_1|+|w_2|}(M, \zeta, w_0)$.
  - If $t_0 < 0$, accept.

– Non-deterministically decide whether to do the following:

  * If $\mathcal{C}_1 = \mathcal{C}_2$ and $t_2 < 0$, accept. Else, reject.

– If M halts, reject.

Note that the counter t_0 counts the number of simulated steps, while the counter t_2 counts the number of steps done on the part w_2.

It is clear that M_{mult} accepts if either a) or b) from the compactness theorem are violated and it rejects if M runs in time $Cn + D$ and b) from the compactness theorem is not violated. Hence M_{mult} correctly solves the problem $\overline{\text{HALT}^1_{\_n+\_}}$.

Because the condition $\mathcal{C}_1 = \mathcal{C}_2$ is verified at most once during the algorithm and

$$\begin{aligned} |\mathcal{C}_1|, |\mathcal{C}_2| &\leq C|w_0| + D \\ &\leq C(\ell + r) + D \\ &= \mathrm{O}\left((CD + C + D + 1)q^C\right), \end{aligned}$$

testing whether $\mathcal{C}_1 = \mathcal{C}_2$ contributes $\mathrm{O}((CD + C + D + 1)q^{C+1})$ time to the overall running time. Because M_{mult} needs $\mathrm{O}(q^2)$ steps to simulate one step of M's computation and it has to simulate at most $C(2\ell + r) + D$ steps, M_{mult} runs in time $\mathrm{O}((CD + C + D + 1)q^{C+2})$. □

Corollary 6.2.2. *The problems* HALT^1_{Cn+D} *and* D-HALT^1_{Cn+D} *are in* co-NP *and their complements can be solved in time* $\mathrm{O}(q^{C+2})$ *by a non-deterministic multi-tape Turing machine.*

6.2.3 The Lower Bounds

Let us state again the idea that we use to prove the lower bounds in Theorem 1.2.1. Suppose a one-tape non-deterministic Turing machine M solves a problem L. Then, for any input w, we can decide whether $w \in L$ by first constructing a one-tape Turing machine M_w that runs in time $Cn + D$ if and only if M rejects w and then solving HALT^1_{Cn+D} for M_w. If we choose L to be a hard language, then we can argue that we cannot solve HALT^1_{Cn+D} fast. The next lemma gives a way to construct M_w.

Lemma 6.2.3. *Let* $C \geq 2$ *and* $D \geq 1$ *be integers, let* $T(n) = Kn^k + 1$ *for some integers* $K, k \geq 1$ *and let* M *be a one-tape* q*-state NTM that runs in time* $T(n)$*. Then there exists an*

$$\mathrm{O}\left(T(n)^{2/(C-1)} + n^2\right) \textit{-time}$$

multi-tape DTM that, given an input w *for* M*, constructs a one-tape NTM* M_w *such that*

$$M_w \textit{ runs in time } Cn + D \iff M \textit{ rejects } w.$$

Proof. Let us first describe the NTM M_w. The computation of M_w on an input $\tilde{w}$ will consist of two phases. In the first phase, M_w will use at most $C - 1$ deterministic passes through the input to assure that $\tilde{w}$ is long enough. We will describe this phase in detail later.

In the second phase, M_w will write w on its tape and simulate M on w. Hence $\mathrm{O}(|w|)$ states and $\mathrm{O}(T(|w|))$ time are needed for this phase (note that q is a constant). If M accepts w, M_w starts an infinite loop, else it halts. Let c be a constant such that M_w makes at most $cT(|w|)$ steps in the second phase before starting the infinite loop.

To describe the first phase, define

$$\begin{aligned} m &= \left\lceil \big(cT(|w|)(C-2)!\big)^{1/(C-1)} \right\rceil \\ &= \mathrm{O}\left(T(|w|)^{1/(C-1)}\right). \end{aligned}$$

In the first phase, the machine M_w simply passes through the input $C-1$ times, each time verifying that $|\tilde{w}|$ is divisible by one of the numbers $m+i$, for $i = 0, 1 \ldots (C-2)$. If this is not the case, M_w rejects. Else, the second phase is to be executed. It suffices to have $m+i$ states to verify in one pass if the length of the input is divisible by $m+i$, so we can make M_w have

$$\begin{aligned} \mathrm{O}\left(\sum_{i=0}^{C-2}(m+i)\right) &= \mathrm{O}\left((C-1)m\right) \\ &= \mathrm{O}(m) \end{aligned}$$

states for the first phase such that it makes at most $(C-1)|\tilde{w}|+1$ steps before entering the second phase[2]. We assume that M_w erases all symbols from the tape in the last pass of the first phase so that the second phase can begin with a blank tape.

If the second phase begins, we know that

$$\begin{aligned} |\tilde{w}| &\geq \operatorname{lcm}\{m, (m+1) \ldots (m+C-2)\} \\ &\geq \frac{m^{C-1}}{(C-2)!} \\ &\geq cT(|w|), \end{aligned}$$

where we used the inequality

$$\operatorname{lcm}\{m, (m+1) \ldots (m+C-2)\} \geq m \cdot \binom{m+C-2}{C-2}$$

proven in [7]. Hence, M_w makes at most $|\tilde{w}|$ steps in the second phase if and only if it does not go into an infinite loop. So we have proven that

$$M_w \text{ runs in time } Cn+1 \iff M_w \text{ runs in time } Cn+D \iff M \text{ rejects } w.$$

To construct M_w, we first compute m which takes $\mathrm{O}(|w|)$ time and then in $\mathrm{O}(m^2)$ time we compute a Turing machine M_1 that does the first phase (the construction is straightforward). Finally we compose M_1 with the Turing machine M, only that M first writes w on the tape and M, instead of going to the accept state, starts moving to the right forever. Because M is not a part of the input and because we can compute compositions of Turing machines in linear time, the description of M_w can be obtained in $\mathrm{O}(m^2 + |w|^2)$ time, which is $\mathrm{O}\left(T(n)^{2/(C-1)} + n^2\right)$. □

We now combine Corollary 6.2.2 and Lemma 6.2.3 to show that most problems $\textsc{Halt}^1_{Cn+D}$ are co-NP-complete.

Proposition 6.2.4. *The problems* $\textsc{Halt}^1_{Cn+D}$ *are* co-NP-complete *for all* $C \geq 2$ *and* $D \geq 1$.

[2]The "plus one" in $(C-1)|\tilde{w}|+1$ is needed because each Turing machine makes at least one step on the empty input. This is also the reason for why we need $D \geq 1$ in the statement of the lemma.

Proof. Corollary 6.2.2 proves that these problems are in co-NP and Lemma 6.2.3 gives a reduction of an arbitrary problem in co-NP to the above ones. □

The first lower bound for the problems HALT^1_{Cn+D} follows. To prove it, we will use Lemma 6.2.3 to translate a hard problem to HALT^1_{Cn+D}.

Proposition 6.2.5. *For all positive integers C and D, the problem $\overline{\text{HALT}^1_{Cn+D}}$ cannot be solved by a multi-tape NTM in time $\mathrm{o}(q^{(C-1)/2})$.*

Proof. For $C \leq 5$, the proposition holds (the length of the input is $\Theta(q^2)$), so suppose $C \geq 6$. By the non-deterministic time hierarchy theorem (Theorem 4.2.3) there exists a language L and a multi-tape NTM M that decides L and runs in time $\mathrm{O}(n^{C-1})$, while no multi-tape NTM can decide L in time $\mathrm{o}(n^{C-1})$. We can reduce the number of tapes of M to get a one-tape NTM M' that runs in time $\mathrm{O}(n^{2(C-1)})$ and decides L (Proposition 3.3.8). By Lemma 6.2.3 there exists a multi-tape DTM M_{mult} that runs in time $\mathrm{O}(n^4)$ and given an input w for M', constructs a one-tape q_w-state NTM M_w such that

$$M_w \text{ runs in time } Cn + D \iff M' \text{ rejects } w.$$

Because the description of M_w has length $\mathrm{O}(|w|^4)$, it follows that $q_w = \mathrm{O}(|w|^2)$.

If there was some multi-tape NTM that could solve $\overline{\text{HALT}^1_{Cn+D}}$ in time $\mathrm{o}\left(q^{(C-1)/2}\right)$, then for all w, we could decide whether $w \in L$ in $\mathrm{o}(n^{C-1})$ non-deterministic time: first run M_{mult} to get M_w and then solve $\overline{\text{HALT}^1_{Cn+D}}$ for M_w. By the definition of L this is impossible, hence the problem $\overline{\text{HALT}^1_{Cn+D}}$ cannot be solved by a multi-tape NTM in time $\mathrm{o}\left(q^{(C-1)/2}\right)$. □

For all of the remaining lower bounds, we need to reformulate Lemma 6.2.3 a bit. Recall from Section 3.3.5 the definition of a *two-choice NTM*, which is an NTM that has at most two possible non-deterministic choices in each step.

Lemma 6.2.6. *Let $C \geq 2$ and $D \geq 1$ be integers and let $T(n) = Kn^k + 1$ for some integers $K, k \geq 1$. Then there exists a multi-tape DTM M_{mult}, which given an input (M, w), where w is an input for a one-tape two-choice q-state NTM M, constructs a one-tape DTM M_w such that*

$$M_w \textit{ runs in time } Cn + D \iff M \textit{ makes at most } T(|w|) \textit{ steps on any computation on the input } w.$$

We can make M_{mult} run in time

$$\mathrm{O}\left(T(|w|)^{4/(C-1)} + q^{\kappa} + |w|^2\right)$$

for some integer $\kappa \geq 1$, independent of C, D, K and k.

Proof. The proof is based on the same idea as the proof of Lemma 6.2.3. The main difference is that this time we will have to count steps while we will simulate M and we will have to use the symbols of an input of the DTM M_w to simulate non-deterministic choices of M.

Again, we begin with the description of M_w. The computation of M_w on an input $\tilde{w}$ will consist of two phases. In the first phase, M_w will use at most $C - 1$ deterministic passes through the input to assure that $\tilde{w}$ is long enough. This phase will be the same as in Lemma 6.2.3, only that we will need more states to measure longer inputs because the second phase will be more time consuming.

This time we define

$$\begin{aligned} m &= \left\lceil \left(\left(cT(|w|)\right)^2 (C-2)! \right)^{1/(C-1)} \right\rceil \\ &= \mathrm{O}\left(T(|w|)^{2/(C-1)} \right) \end{aligned}$$

for some constant c defined later. In the first phase, the machine M_w simply passes through the input $C-1$ times, each time verifying that $|\tilde{w}|$ is divisible by one of the numbers $m+i$, for $i = 0, 1 \ldots (C-2)$. If this is not the case, M_w rejects. Else, the second phase is to be executed. It suffices to have $m+i$ states to verify in one pass if the length of the input is divisible by $m+i$, so we can make M_w have

$$\mathrm{O}\left(\sum_{i=0}^{C-2} (m+i) \right) = \mathrm{O}(m)$$

states for the first phase such that it makes at most $(C-1)|\tilde{w}|+1$ steps before entering the second phase. We also need that while the Turing machine M_w passes through the input in the first phase, it does not change it, except that it erases the first symbol of $\tilde{w}$ (if not, M_w would need $n+1$ steps for one pass through the input). Additionally, if the input $\tilde{w}$ contains some symbol that is not 0 or 1, M_w rejects.

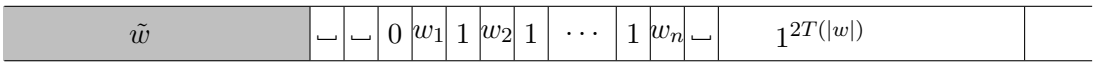


Figure 6.1: The preparation for the simulation in the phase two. After the phase one, the head of M_w could be on the left side of the input $\tilde{w}$ or on the right side of it, depending on the parity of C. Let us assume that C is even and hence the head of M_w is on the right side of $\tilde{w}$ after the phase one. Before the simulation begins, M_w writes the following on the right side of $\tilde{w}$: ␣␣0 followed by w with the symbol 1 inserted between each two of its symbols. Then on the right of w it computes $2T(|w|)$ in unary so that we get the situation as shown in the figure. The input $\tilde{w}$ (without the first symbol) is much longer compared to what is on the right side of it (phase one takes care for that). If C is odd, we can look on the tape of M_w from behind and do everything the same as in the case of C being even.

In the second phase, M_w will compute $T(|w|)$ and simulate M on w for at most $T(|w|)$ steps, using the non-deterministic choices determined by $\tilde{w}$. If M will not halt, M_w will start an infinite loop, else it will halt. In Figure 6.1 we see how M_w makes the preparation for the second phase. Let us call the part of the tape with the symbols of $\tilde{w}$ written on it the *non-deterministic part*, the part of the tape from 0 to w_n the *simulating part* and the part of the tape with $1^{2T(|w|)}$ written on it the *counting part*. During the simulation, the following will hold.

- In the simulating part, it will always be the case that the symbols from the tape of M will be in every second cell and between each two of them will always be the symbol 1, except on the left of the cell with the head of M on it where it will be 0.
- There will always be at least two blank symbols left of the simulating part and there will always be at least one blank symbol right of the simulating part. This will be possible because before each simulated step of M, as explained below, the number of blank symbols left and

right of the simulating part will be increased by two for each side, hence when simulating a step of M, the simulating part can be increased as necessary.

- Before each simulated step of M, M_w will use the rightmost symbol of the non-deterministic part of the tape to determine a non-deterministic choice for M and it will overwrite the two rightmost symbols of the non-deterministic part of the tape with two blank symbols.
- Before each simulated step of M, M_w will overwrite the two leftmost symbols of the counting part of the tape with two blank symbols.
- If M halts before the counting part of the tape vanishes, M_w halts. Else, M_w starts an infinite loop.

We see that M_w, if it does not go into an infinite loop, finishes the second phase in time $\mathrm{O}(T(|w|)^2)$ using $\mathrm{O}(|w| + q)$ states. Note that to achieve that, the counting part of the tape really has to be computed and not encoded in the states, which takes $\mathrm{O}(T(|w|)^2)$ steps. A possible implementation of this would be to first write $|w|$ in binary ($|w|$ can be encoded in the states), then compute $T(|w|)$ in binary and extend it to unary.

To define the integer c that is used in the first phase, suppose that M_w makes at most $(cT(|w|))^2$ steps in the second phase before starting the infinite loop. Note that c is independent of M and w. If the second phase begins, we know that

$$\begin{aligned} |\tilde{w}| &\geq \operatorname{lcm}\{m, (m+1)\ldots(m+C-2)\} \\ &\geq \frac{m^{C-1}}{(C-2)!} \\ &\geq (cT(|w|))^2, \end{aligned}$$

as in the proof of Lemma 6.2.3, thus M_w makes at most $|\tilde{w}|$ steps in the second phase if and only if it does not go into an infinite loop. This inequality also implies that the non-deterministic part of the tape in the phase two is long enough so that it does not vanish during the simulation of M.

Now if M makes at most $T(|w|)$ steps on all computations on the input w, then M_w runs in time $Cn + 1$. But if there exists a computation ζ on input w such that M makes more than $T(|w|)$ steps on it, then because M is a two-choice machine, there exists a binary input $\tilde{w}$ for M_w such that the non-deterministic part of the tape in the phase two corresponds to the non-deterministic choices of ζ, hence M_w on the input $\tilde{w}$ simulates more than $T(|w|)$ steps of M which means that the counting part of the tape vanishes and thus M_w does not halt on the input $\tilde{w}$. So we have proven that

$$M_w \text{ runs in time } Cn+1 \iff M_w \text{ runs in time } Cn+D \iff M \text{ makes at most } T(|w|) \text{ steps on the input } w.$$

Now let us describe a multi-tape DTM M_{mult} that constructs M_w from (M, w). First we prove that, for some integer κ independent of $C,\ D,\ K$ and k, the DTM M_{mult} can construct, in time $\mathrm{O}(q^\kappa + |w|^2)$, a one-tape DTM M_2 that does the second phase. To see this, let M_T be a one-tape DTM that given a number x in binary, its head never crosses the boundary -1 and it computes $T(x)$ in unary in time $\mathrm{O}(T(x)^2)$. Note that M_T does not depend on the input (M, w) for M_{mult} and thus it can be computed in constant time. Now M_2 can be viewed as a composition of three deterministic Turing machines:

- The first DTM writes down the simulating part of the tape, followed by $|w|$ written in binary. M_{mult} needs $\mathrm{O}(|w|^2)$ time to construct this DTM.
- The second DTM is M_T and M_{mult} needs $\mathrm{O}(1)$ time to construct it.
- The third DTM performs the simulation of M on w and M_{mult} needs $\mathrm{O}(q^\kappa)$ time to construct it, where κ is independent of $C,\ D,\ K$ and k.

Because the composition of Turing machines can be computed in linear time, we can construct M_2 in time $\mathrm{O}(q^\kappa + |w|^2)$.

Because the first phase does not depend on M and we need $\mathrm{O}(m)$ states to do it, M_{mult} can compute the DTM M_1 that does the first phase in time

$$\mathrm{O}(m^2) = \mathrm{O}\left(T(|w|)^{4/(C-1)}\right),$$

as in the proof of Lemma 6.2.3. Since M_w is the composition of M_1 and M_2, M_{mult} can construct M_w in time

$$\mathrm{O}\left(T(|w|)^{4/(C-1)} + q^\kappa + |w|^2\right). \qquad \square$$

Proposition 6.2.7. *For all positive integers C and D, the problem* D-HALT$^1_{Cn+D}$ *cannot be solved by a multi-tape NTM in time* $\mathrm{o}(q^{(C-1)/4})$.

Proof. For $C \leq 9$, the proposition holds (the length of the input is $\Theta(q^2)$), so suppose $C \geq 10$. Let κ be as in Lemma 6.2.6 and let M be the following one-tape NTM:

- On an input w which is a padded code of a one-tape two-choice NTM M', construct a one-tape q_w-state DTM M_w such that

  $$M_w \text{ runs in time } Cn + D \iff M' \text{ makes at most } |w|^{\kappa(C-1)} \text{ steps on the input } w.$$

  The machine M_w can be constructed by a multi-tape DTM in time $\mathrm{O}(|w|^{4\kappa})$ by Lemma 6.2.6, hence it can be constructed in time $\mathrm{O}(|w|^{8\kappa})$ by a one-tape DTM (as in Proposition 3.3.8). It also follows that $q_w = \mathrm{O}(|w|^{2\kappa})$.
- Verify whether M_w runs in time $Cn + D$. If so, start an infinite loop, else halt.

Now we make a standard diagonalization argument to prove the proposition. Suppose that D-HALT$^1_{Cn+D}$ can be solved by a multi-tape NTM in time $\mathrm{o}(q^{(C-1)/4})$. Then it can be solved in time $\mathrm{o}(q^{(C-1)/2})$ by a one-tape NTM (Proposition 3.3.8). Using more states and for a constant factor more time, D-HALT$^1_{Cn+D}$ can be solved in time $\mathrm{o}(q^{(C-1)/2})$ by a one-tape two-choice NTM (Proposition 3.3.16). If M uses this machine to verify whether M_w runs in time $Cn + D$, then considering $q_w = \mathrm{O}(|w|^{2\kappa})$ and $C \geq 10$, M is a one-tape two-choice NTM that makes

$$\mathrm{O}\left(|w|^{8\kappa}\right) + \mathrm{o}\left(|w|^{\kappa(C-1)}\right) = \mathrm{o}\left(|w|^{\kappa(C-1)}\right)$$

steps on any computation on the input w, if it does not enter the infinite loop.

Let w be a padded code of M. If M makes at most $|w|^{\kappa(C-1)}$ steps on the input w, the Turing machine M_w will run in time $Cn + D$ which implies that M will start an infinite loop on some

computation on the input w, which is a contradiction. Hence, M must make more steps on the input w than $|w|^{\kappa(C-1)}$ which implies that M_w does not run in time $Cn + D$ and hence M does not start the infinite loop, thus it makes $\mathrm{o}\left(|w|^{\kappa(C-1)}\right)$ steps. It follows that

$$\mathrm{o}\left(|w|^{\kappa(C-1)}\right) > |w|^{\kappa(C-1)}$$

which is impossible since the padding can be arbitrarily long. □

Corollary 6.2.8. *For all positive integers C and D, the problem* $\textsc{Halt}^1_{Cn+D}$ *cannot be solved by a multi-tape NTM in time* $\mathrm{o}(q^{(C-1)/4})$.

Proof. The result follows by Proposition 6.2.7 because a Turing machine that solves $\textsc{Halt}^1_{Cn+D}$ also solves D-$\textsc{Halt}^1_{Cn+D}$. □

The following lemma is a "deterministic" analog of Lemma 6.2.3.

Lemma 6.2.9. *Let $C \geq 2$ and $D \geq 1$ be integers, let $T(n) = Kn^k + 1$ for some integers $K, k \geq 1$ and let M be a one-tape two-choice q-state NTM that runs in time $T(n)$. Then there exists an*

$$\mathrm{O}\left(T(n)^{4/(C-1)} + n^2\right) \textit{-time}$$

multi-tape DTM that given an input w for M, constructs a one-tape DTM M_w such that

$$M_w \textit{ runs in time } Cn + D \iff M \textit{ rejects } w.$$

Proof. Let M' be a one-tape two-choice $(q+1)$-state NTM that computes just like M only that it starts an infinite loop whenever M would go to the accepting state. It follows that

$$M' \text{ makes at most } T(|w|) \text{ steps on the input } w \iff M \text{ rejects } w.$$

Now we can use Lemma 6.2.6 to construct a DTM M_w such that

$$\begin{aligned} M_w \text{ runs in time } Cn + D &\iff M' \text{ makes at most } T(|w|) \text{ steps on the input } w \\ &\iff M \text{ rejects } w. \end{aligned}$$

Because M and M' are fixed, we can construct M_w in time

$$\mathrm{O}\left(T(|w|)^{4/(C-1)} + |w|^2\right)$$

by Lemma 6.2.6. □

We now combine Corollary 6.2.2 and Lemma 6.2.9 to show that D-$\textsc{Halt}^1_{Cn+D}$ is co-NP-complete.

Proposition 6.2.10. *The problems* D-$\textsc{Halt}^1_{Cn+D}$ *are* co-NP-complete *for all $C \geq 2$ and $D \geq 1$.*

Proof. Corollary 6.2.2 proves that these problems are in co-NP and Lemma 6.2.9 gives a reduction of an arbitrary problem in co-NP to the above ones. □

To prove the last lower bound, we use Lemma 6.2.9 to translate a hard problem to D-$\textsc{Halt}^1_{Cn+D}$, the same way as in Proposition 6.2.5.

Proposition 6.2.11. *For all positive integers C and D, the problem* $\overline{\text{D-}\textsc{Halt}^1_{Cn+D}}$ *cannot be solved by a multi-tape NTM in time* $\mathrm{o}(q^{(C-1)/4})$.

Proof. The proof is the same as the proof of Proposition 6.2.5, only that we use Lemma 6.2.9 instead of Lemma 6.2.3. □

To sum up this section, we have proven Theorem 1.2.1 which states

For all integers $C \geq 2$ and $D \geq 1$, all of the following holds.

(i) The problems HALT^1_{Cn+D} *and* D-HALT^1_{Cn+D} *are* co-NP-complete.
Proposition 6.2.4 and Proposition 6.2.10 prove this.

(ii) The problems HALT^1_{Cn+D} and D-HALT^1_{Cn+D} *cannot* be solved in non-deterministic time $o(q^{(C-1)/4})$.
Proposition 6.2.7 and Corollary 6.2.8 prove this.

(iii) The problems $\overline{\text{HALT}^1_{Cn+D}}$ and $\overline{\text{D-HALT}^1_{Cn+D}}$ *can* be solved in non-deterministic time $O(q^{C+2})$.
Corollary 6.2.2 proves this.

(iv) The problem $\overline{\text{HALT}^1_{Cn+D}}$ *cannot* be solved in non-deterministic time $o(q^{(C-1)/2})$.
Proposition 6.2.5 proves this.

(v) The problem $\overline{\text{D-HALT}^1_{Cn+D}}$ *cannot* be solved in non-deterministic time $o(q^{(C-1)/4})$.
Proposition 6.2.11 proves this. □

6.2.4 Optimality of Our Measuring of the Length of an Input

Let us again have a look at how we proved the lower bound in Proposition 6.2.5. A very similar idea was also used to prove all the other lower bounds in Theorem 1.2.1, so what follows can be applied to any of them.

Let a one-tape non-deterministic Turing machine M solve a problem L in time $T(n)$. Then, for any input w, we can decide whether $w \in L$ by first constructing a one-tape Turing machine M_w that runs in time $Cn + D$ if and only if M rejects w and then solving HALT^1_{Cn+D} for M_w. On the inputs of length n, the Turing machine M_w computed in two phases (see Lemma 6.2.3): in the first phase M_w only measured the input length using at most $(C-1)n+1$ steps to assure that n was large enough, specifically $n = \Omega(T(|w|))$, and then in the second phase it simulated M on w using at most n steps. In our implementation M_w used $O(T(|w|)^{1/(C-1)})$ states for the first phase and *we claim that this is optimal*.

If we want M_w to measure the length $T(|w|)$ in the first phase using at most $(C-1)n+1$ steps, then for each computation on inputs of length $T(|w|)$, it cannot produce the same crossing sequence at two boundaries. By Lemma 5.2.3, M_w has $\Omega(T(|w|)^{1/(C-1)})$ states which implies that our measuring of the length of the input was optimal. What is more, Lemma 5.2.3 is tight and our method for proving lower bounds cannot give much better bounds.

6.2.5 Relativization in Theorem 1.2.1

While the upper bounds in Theorem 1.2.1 relativize, our lower bound proofs give slightly less powerful lower bounds for NOTMs than for NTMs. The following list gives reasons and indicates where one has to be careful when using NOTMs.

- The property of our encoding of one-tape NTMs that the composition of two NTMs can be computed in linear time is not that clear for NOTMs because defining a composition of two NOTMs is quite more technical, as it is discussed in Section 4.3.2. However, in relativized versions of the proofs of our results it is always the case that the first Turing machine in a composition of two oracle Turing machines does not need an oracle, hence it can be treated

as an NTM. In such a case we can compute a composition of an NTM and an NOTM in linear time.

- In several results, for example in Lemma 6.2.3, we did the following: given an NTM M and an input w for M, construct a one-tape NTM $\tilde{M}$ that computes in two phases: in the first phase it makes several passes through the input and in the second phase it simulates M on w. Note that the simulation of M on w can only be done on the left part of the input to be able to make oracle queries, thus $\tilde{M}$ has to make an even number of passes in the first phase. This results in comparable, but slightly weaker lower bounds.

- Note from Section 4.3.3 that if a multi-tape NOTM decides a language in time $T(n)$, then there exist a one-tape NOTM that decides the same language in time $\mathrm{O}(T(n)^3)$. To prove lower bounds, we used the better bound $\mathrm{O}(T(n)^2)$ that holds for NTMs. This results in comparable, but slightly weaker lower bounds for NOTMs.

- In Figure 6.1 we can see the tape of a one-tape NTM prepared for the simulation of another one-tape NTM. A similar preparation could be done also in the case of NOTMs (only on the left part of the input), however we may need more steps to simulate a computation because we have to keep time somewhere on the tape, mark the head position (also on the oracle part of the tape) and later possibly do an oracle query ... This again results in comparable, but slightly weaker lower bounds.

Because our methods from Section 6.2 can be applied also to NOTMs and they give comparably good results, using only such methods cannot give the solution to the P versus NP problem.

6.2.6 An Open Problem

For $D \in \mathbb{N}$, how hard are the problems $\textsc{Halt}^1_{n+D}$ *and* D-$\textsc{Halt}^1_{n+D}$*?*

It is clear that we can solve the problems $\textsc{Halt}^1_{Cn}$ and D-$\textsc{Halt}^1_{Cn}$, for $C \in \mathbb{N}$, in constant time. The answer is always *no*, since any Turing machine makes at least one step on the empty input.

It is also easy to see that we can solve the problems $\textsc{Halt}^1_D$ and D-$\textsc{Halt}^1_D$, for $D \in \mathbb{N}$, in polynomial time. The algorithm would be to simulate a given one-tape Turing machine M on all the inputs up to the length D and accept if and only if the time bound was not violated. Now, if the algorithm rejects, M clearly does not run in time D and if it accepts, then M never reads the $(D+1)$st symbol of an input by Lemma 3.3.1 and hence it was enough to verify the running time on inputs up to the length D.

For $C \geq 2$ and $D \geq 1$, good complexity bounds for $\textsc{Halt}^1_{Cn+D}$ and D-$\textsc{Halt}^1_{Cn+D}$ are given in Theorem 1.2.1. Hence only the bounds for $C = 1$ are missing. For this case we can prove the following proposition.

Proposition 6.2.12. *The problems* $\textsc{Halt}^1_{n+1}$ *and* D-$\textsc{Halt}^1_{n+1}$ *are in* P.

Proof. The main observation is that a one-tape NTM which runs in time $n+1$ never moves its head to the left, except possibly in the last two steps of a computation. To prove this, we suppose the opposite. Let M be a one-tape NTM that runs in time $n+1$ and let w be an input for M such that on some computation on w, M moves its head to the left for the first time in step $t < n = |w|$ and it makes at least two more steps afterwards. As can be seen in Figure 6.2, M makes more than $t+1$ steps on some computation on the input $w(0,t)$ of length t, which is a contradiction.

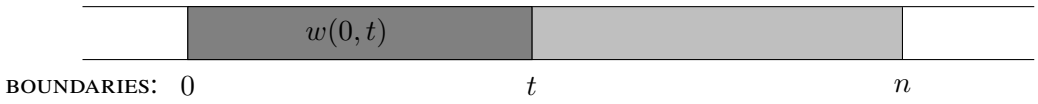


Figure 6.2: Suppose that a Turing machine M on input w of length n moves its head to the left for the first time in step t (the head turns left just before crossing the boundary t) and let M make at least two more steps after this step (we assume some fixed computation). Then M on the input $w(0,t)$ makes at least $t+2$ steps.

Hence, to solve HALT^1_{n+1} and D-HALT^1_{n+1} it is enough to verify, for a given one-tape Turing machine M, whether the head of M never moves to the left, except possibly in the last two steps of a computation. This can be verified in polynomial time. □

Does a similar proof go through for all problems HALT^1_{n+D}?

Slovenski povzetek

Ta povzetek ima enako strukturo kot uvodno poglavje disertacije. Razdeljen je na tri dele. Prvi del je namenjen širšemu krogu bralcev; v njem podamo motivacijo za probleme, ki jih obravnavamo v disertaciji, in opišemo, kako predstavljeni koncepti zrcalijo realnost. Drugi del je namenjen bralcem, ki so seznanjeni z osnovami teorije računske zahtevnosti; v tem delu preletimo vsebino vseh poglavij ter podamo glavne ideje dokazov pomembnejših rezultatov. Tretji del je namenjen poznavalcem teorije računske zahtevnosti, ki jih zanima tudi sorodna literatura.

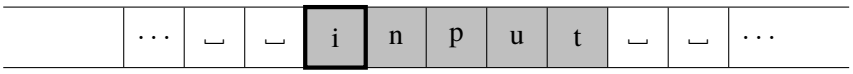
Kazalo

Motivacija

Vse od pojava prvih računalnikov obstaja naravna opredelitev zahtevnosti računskih problemov: problem je težak, če ga z računalnikom ni mogoče hitro rešiti. Ta empirična definicija dobi trde temelje, če namesto računalnika vzamemo neki dobro definirani model računanja. Skozi zgodovino se je izkazalo, da je Turingov stroj ravno pravšnji model. Lahko torej rečemo, da je problem težak, če ga noben Turingov stroj ne more hitro rešiti.

Dobro osnovo za takšno obravnavo zahtevnosti problemov nam predstavljata dve tezi. *Church-Turingova teza* pravi, da je množica funkcij, ki jih lahko računamo s Turingovim strojem, enaka množici *intuitivno izračunljivih funkcij*. Teza ni matematična trditev in je zato ni mogoče formalno dokazati, čeprav so nekateri poskusili storiti prav to [5]. Dober argument v prid tezi je predvsem ta, da je moč s Turingovim strojem simulirati veliko znanih modelov računanja, tudi modele naših računalnikov. Še več, simulacije so efektivne. To pomeni, da Turingov stroj ne naredi bistveno več korakov kot simulirani model računanja. Ta ugotovitev je osnova za *krepko različico Church-Turingove teze*, ki pravi, da so vsi "smiselni" modeli računanja polinomsko ekvivalentni Turingovemu stroju (tj. so primerljivo hitri). Krepka različica teze med teoretiki ni tako splošno sprejeta kot Church-Turingova teza in kvantni računalniki jo domnevno kršijo. Na tem mestu omenimo, da sodobna tehnologija še ne omogoča izgradnje primerno velikih kvantnih računalnikov.

Turingovi stroji lahko izračunajo vse, kar lahko izračunajo teoretični modeli osebnih računalnikov, in to lahko storijo (teoretično) primerljivo hitro. Največja prednost Turingovih strojev pred temi modeli pa je "enostavnost". Da bi pokazali, kako preprosti so, na kratko predstavimo enotračni deterministični Turingov stroj (enotračni DTS) M; formalno definicijo najdemo v poglavju 3.1.6. Fizično M sestavljajo v obe smeri neskončen *trak*, *glava* in *kontrolni mehanizem*, ki omogoča, da je stroj zmeraj v natanko enem izmed končnega števila *stanj*. Nekatera stanja so posebna: eno je *začetno*, nekaj pa jih je *končnih*. Trak je razdeljen na *celice*, v vsaki celici je zapisan neki *simbol* in nad natanko eno izmed celic je glava (glej sliko 1). M zmeraj začne računati v začetnem stanju, pri čemer je na traku zapisan vhod (vsak simbol vhoda v svoji celici, ostale celice pa vsebujejo tako imenovani *prazni simbol* $\sqcup$) in je glava nad prvim simbolom vhoda. V vsakem koraku glava najprej prebere simbol zapisan pod njo na traku, ki skupaj s trenutnim stanjem popolnoma določi naslednji korak stroja M tipa: prepiši simbol pod glavo z drugim simbolom, lahko tudi enakim, premakni glavo za eno celico v levo ali v desno ter zamenjaj stanje. Torej M računa zelo lokalno, saj je vsak naslednji korak določen le s trenutnim stanjem in simbolom pod glavo. Izračun se konča, ko M preide v eno izmed končnih stanj. Če se to nikdar ne zgodi, potem se M nikdar ne ustavi. Rezultat izračuna je lahko stanje na traku ob koncu izračuna ali pa končno stanje, v katerem M zaključi izvajanje. Kot smo že navedli, lahko tak stroj (navkljub enostavnosti) efektivno simulira izračune sodobnih računalnikov.



Slika 1: Trak enotračnega DTS-ja M z vhodom *input*. Preden M začne računati, je v začetnem stanju in njegova glava je nad simbolom *i*.

V disertaciji se ukvarjamo večinoma z *odločitvenimi problemi*, to so taki problemi, ki zahtevajo odgovor *da* ali *ne*. Podrobneje so predstavljeni v poglavju 2.1.4, tukaj bomo predstavili le tri primere.

PRIMERJAVA DOLŽIN	...	Ali je dani niz w oblike $00\ldots011\ldots1$, kjer je število ničel enako številu enic?
HAMILTONOV CIKEL	...	Ali je dani enostavni neusmerjeni graf Hamiltonov[3]?
D-HALT$^1_\varepsilon$	...	Ali se dani enotračni DTS M ustavi na praznem vhodu, tj. vhodu, ki ne vsebuje nobenega simbola?

Najtežji problemi so taki, ki jih ni mogoče rešiti s Turingovimi stroji, in dobro poznano dejstvo je, da je D-HALT$^1_\varepsilon$ tak problem (za dokaz glej poglavje 4.1.1). To dejstvo je zanimivo že samo po sebi, za širšo javnost pa je najbrž bolj zanimiva njegova posledica, da ne obstaja računalniški program, ki bi rešil problem:

Za dani program v programskem jeziku Java, ki ne sprejme nobenega vhoda, ali bi se program kdaj ustavil, če bi ga pognali?

Torej je preverjanje pravilnosti programske kode naloga, ki je ne moremo povsem avtomatizirati.

Naravno je odločitvene probleme razvrstiti v razrede glede na to, kako hitro jih lahko rešimo s Turingovimi stroji. Tako dobimo hierarhijo različnih razredov računske zahtevnosti, ki jo podrobneje opišemo v poglavju 4.2.2. Najbolj znani razred računske zahtevnosti je razred P, ki vsebuje tiste odločitvene probleme, ki so rešljivi z enotračnimi Turingovimi stroji v polinomskem času. Če povemo drugače, je odločitveni problem v razredu P natanko tedaj, ko obstaja polinom p in enotračni DTS, ki reši problem in za vsak n naredi največ $p(n)$ korakov na vhodih dolžine n.

Zelo znan razred odločitvenih problemov je tudi NP, ki vsebuje natanko tiste odločitvene probleme, katerih odgovore *da* lahko preverimo v polinomskem času z enotračnim DTS-jem, ki ob vhodu sprejme še kratek niz, ki mu pravimo *certifikat* (nekakšen namig). Razred je natančno definiran v poglavju 3.4; tukaj ga bomo raje predstavili na primeru. Problem HAMILTONOV CIKEL je v NP, saj za vsak graf, ki ima Hamiltonov cikel, obstaja certifikat, da je to res: zaporedje vozlišč, ki tvori Hamiltonov cikel. Če ob vhodnem grafu dobimo še neko zaporedje vozlišč (kot certifikat), lahko v polinomskem času preverimo, ali je to zaporedje Hamiltonov cikel ali ne. Po drugi strani pa niso znani kratki certifikati, ki bi pomagali pri reševanju komplementa problema HAMILTONOV CIKEL, ki se glasi:

Ali je res, da dani enostavni neusmerjeni graf G ni Hamiltonov?

Ta problem je v razredu $\mathsf{co\text{-}NP}$, ki vsebuje natanko komplemente odločitvenih problemov iz razreda NP (odgovori *da* in *ne* so zamenjani). Obstaja mnogo znanih problemov, ki so v razredih NP ali $\mathsf{co\text{-}NP}$, ni pa znano, ali so tudi v razredu P. Eden takih je tudi problem HAMILTONOV CIKEL [12]. Medtem ko očitno velja $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{co\text{-}NP}$, pa je vprašanje $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ že desetletja osrednje vprašanje v teoriji računske zahtevnosti, ki je motiviralo številne znane rezultate na tem področju. To vprašanje predstavlja enega izmed problemov, poznanih pod imenom Millennium Prize Problems, in njegova rešitev je vredna milijon ameriških dolarjev [23]. Problem $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ se je pojavil tudi v naslovu knjige Richarda J. Liptona [22] in objavljene so bile analize o tem, kaj teoretiki menijo o njem [13]. Obstaja še veliko drugih naravnih razredov odločitvenih problemov, za katere ni znano, v kakšni relaciji so s P, NP in med seboj. Naj omenimo le dve sorodni odprti vprašanji, $\mathsf{NP} \stackrel{?}{=} \mathsf{co\text{-}NP}$ in $\mathsf{P} \stackrel{?}{=} \mathsf{NP} \cap \mathsf{co\text{-}NP}$.

[3] Graf je enostaven, če nima vzporednih povezav in zank. Graf je Hamiltonov, če obstaja cikel, ki vsebuje vsa njegova vozlišča.

Ker je veliko naravnih razredov odločitvenih problemov definiranih s pomočjo Turingovih strojev, je temeljito poznavanje tega modela računanja raziskovalcu na področju računske zahtevnosti lahko v veliko korist. Ena glavnih lastnosti Turingovih strojev je časovna zahtevnost. Glavni avtorjevi rezultati [10, 11] v času doktorskega študija govorijo o tem, kako preveriti in ali je sploh mogoče algoritmično preveriti časovno zahtevnost danega Turingovega stroja. Ti rezultati so predstavljeni v poglavju 6.

Pregled disertacije

Osrednji rezultati v disertaciji so v poglavjih 5 in 6, ostala poglavja služijo predvsem za predstavitev ozadja. V poglavju 6 predstavimo rezultate o preverjanju časovne zahtevnosti Turingovih strojev. Medtem ko so rezultati v primeru večtračnih Turingovih strojev relativno enostavni, pa za analizo časovne zahtevnosti enotračnih Turingovih strojev potrebujemo več različnih konceptov. Eden glavnih so prekrižna zaporedja, o katerih je govora v poglavju 5. Večina rezultatov v poglavjih 5 in 6 je avtorjevih [10, 11] in so podrobneje predstavljeni spodaj.

Preletimo najprej ostala poglavja. Poglavje 1 je uvodno in je v grobem angleška verzija tega slovenskega povzetka. V poglavju 2 predstavimo osnovno notacijo, definiramo regularne jezike, končne avtomate ter regularne izraze. Dokažemo tudi, da regularni izrazi in končni avtomati opišejo natanko regularne jezike. V poglavju 3 definiramo več različnih modelov Turingovih strojev: enotračne in večtračne, deterministične in nedeterministične. Definiramo tudi časovno pogojene razrede odločitvenih problemov, med drugimi razreda P in NP. V pomembnem in zelo tehničnem podpoglavju 3.3 analiziramo, kako zaostritve različnih parametrov Turingovih strojev vplivajo na časovno zahtevnost razpoznavanja jezikov. Parametri, ki jih obravnavamo, so velikost tračne abecede, število trakov in uporaba nedeterminizma.

V poglavju 4 dokažemo neodločljivost zaustavitvenega problema ter neodločljivost problema D-HALT$^1_\varepsilon$. Prav tako dokažemo izreka o deterministični in nedeterministični časovni hierarhiji. Definiramo tudi Turingove stroje z orakljem in sicer tako, da obstajajo tudi enotračni Turingovi stroji z orakljem in da se tehnike dokazovanja, ki jih uporabljamo v poglavjih 5 and 6, enostavno prenesejo iz "navadnih" Turingovih strojev nanje. Tehnikam, ki jih lahko tako "prenesemo", pravimo, da *relativizirajo*. Na koncu poglavja 4 pokažemo, da le s tehnikami, ki relativizirajo, ne moremo rešiti problema $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

Poglavje 6

To poglavje je zadnje in nosi enak naslov kot disertacija: Preverjanje časovne zahtevnosti Turingovih strojev. Vsi rezultati, ki jih bomo v tem povzetku poglavja predstavili, veljajo tako za deterministične Turingove stroje (DTS) kot za nedeterministične Turingove stroje (NTS), razen če napišemo drugače.

Za funkcijo $T : \mathbb{N} \to \mathbb{R}_{>0}$ obstajata vsaj dva naravna tipa problemov preverjanja časovne zahtevnosti, podane s pomočjo funkcije $T(n)$:

- Ali je dani Turingov stroj časovne zahtevnosti $\mathrm{O}(T(n))$?
- Ali je dani Turingov stroj časovne zahtevnosti $T(n)$, tj. ali za vsak n dani Turingov stroj napravi največ $T(n)$ korakov na vsakem izračunu na vhodih dolžine n?

Teoretikom je že dolgo znano, da ne obstaja algoritem, ki bi preveril, ali je dani Turingov stroj časovne zahtevnosti $\mathrm{O}(1)$. Torej je prvi problem neodločljiv za vse uporabne funkcije T. V iz-

reku 6.1.6 najdemo posplošitev tega rezultata. Po drugi strani pa je drugi problem odločljiv za vsako konstantno funkcijo $T(n) = C$. Namreč, da bi preverili, ali je dani Turingov stroj časovne zahtevnosti C, ga moramo le simulirati na vhodih do dolžine C (za argumente glej dokaz leme 6.1.1). Izkaže se celo, da lahko natančno karakteriziramo funkcije T, za katere je drugi problem odločljiv. Izrek 6.1.3 nam namreč pove, da je drugi problem odločljiv natanko tedaj, ko imamo izrojeni primer $T(n_0) < n_0 + 1$ za neki $n_0 \in \mathbb{N}$. Meja časovne zahtevnosti $n+1$ je posebna zato, ker je minimalna taka, ki omogoča večtračnemu Turingovemu stroju, da meri čas svojega izvajanja in hkrati simulira drug Turingov stroj. Čas izvajanja lahko meri na vhodnem traku s tem, da glavo pomika v desno, dokler ne prebere praznega simbola, na preostalih trakovih pa simulira drug Turingov stroj.

Ta strategija se podre, če se omejimo na enotračne Turingove stroje. Na slednjih moramo merjenje časa ter simulacijo opraviti na istem traku in izkaže se, da za to v splošnem potrebujemo $\Omega(n \log n)$ časa. Da je $\Omega(n \log n)$ dovolj, je razvidno iz dokaza izreka 6.1.5, ki pravi:

> *Naj za funkcijo $T : \mathbb{N} \to \mathbb{R}_{>0}$ velja $T(n) = \Omega(n \log n)$ in naj za vsak $n \in \mathbb{N}$ velja $T(n) \geq n + 1$. Potem ni odločljivo, ali je dani enotračni Turingov stroj časovne zahtevnosti $T(n)$.*

Po drugi strani pa nam izrek 6.1.10 pove, da je meja $\Theta(n \log n)$ tesna:

> *Naj bo $T : \mathbb{N} \to \mathbb{R}_{>0}$ "lepa" funkcija, za katero velja $T(n) = \mathrm{o}(n \log n)$. Potem je odločljivo, ali je dani enotračni Turingov stroj časovne zahtevnosti $T(n)$.*

Meje časovne zahtevnosti reda $\Theta(n \log n)$ so zanimive še zaradi nečesa. So minimalne take, ki omogočajo enotračnim Turingovim strojem, da sprejmejo neregularen jezik. Vsak enotračni Turingov stroj časovne zahtevnosti $\mathrm{o}(n \log n)$ namreč odloči neki regularni jezik (trditev 5.1.7). Po drugi strani pa obstaja neregularni jezik, ki ga sprejme neki enotračni Turingov stroj v času $\mathrm{O}(n \log n)$ (trditev 5.1.9).

Zanimivo je, da je vsak enotračni Turingov stroj časovne zahtevnosti $\mathrm{o}(n \log n)$ tudi linearne časovne zahtevnosti (posledica 5.1.6). Torej je linearna časovna zahtevnost najnaravnejša algoritmično preverljiva časovna zahtevnost enotračnih Turingovih strojev. To dejstvo je motivacija za drugi del zadnjega poglavja, v katerem analiziramo računsko zahtevnost naslednjih problemov, parametriziranih s $C, D \in \mathbb{N}$. Problem $\textsc{Halt}^1_{Cn+D}$ je sledeč:

Ali je dani enotračni NTS časovne zahtevnosti $Cn + D$?

Problem D-$\textsc{Halt}^1_{Cn+D}$ je sledeč:

Ali je dani enotračni DTS časovne zahtevnosti $Cn + D$?

Za lažjo analizo teh problemov fiksirajmo vhodno abecedo Σ, ki naj vsebuje vsaj dva simbola, in tračno abecedo $\Gamma \supset \Sigma$. Posledično za večino klasičnih kodiranj enotračnih Turingovih strojev velja, da je dolžina kode Turingovega stroja s q stanji $\mathrm{O}(q^2)$. Da analizo še olajšamo, vzemimo tako kodiranje, da bo dolžina kode enotračnega Turingovega stroja s q stanji $\Theta(q^2)$. Primer takega kodiranja je podan v poglavju 6.2.1. Ob teh predpostavkah lahko računsko zahtevnost problemov $\textsc{Halt}^1_{Cn+D}$ in D-$\textsc{Halt}^1_{Cn+D}$ izražamo s parametrom q, ki do konstantnega multiplikativnega faktorja določa dolžino vhoda. Računsko zahtevnost teh problemov nam zelo natančno opredeli naslednji izrek (izrek 1.2.1).

Za poljubni naravni števili $C \geq 2$ in $D \geq 1$ veljajo vse naslednje točke.

(i) Problema $\textsc{Halt}^1_{Cn+D}$ *in* D-$\textsc{Halt}^1_{Cn+D}$ *sta* co-NP-polna.

(ii) Problema $\textsc{Halt}^1_{Cn+D}$ *in* D-$\textsc{Halt}^1_{Cn+D}$ nista rešljiva *v času* $\mathrm{o}(q^{(C-1)/4})$ *z večtračnimi NTS-ji.*

(iii) Komplementa problemov $\textsc{Halt}^1_{Cn+D}$ *in* D-$\textsc{Halt}^1_{Cn+D}$ sta rešljiva *v času* $\mathrm{O}(q^{C+2})$ *z večtračnimi NTS-ji.*

(iv) Komplement problema $\textsc{Halt}^1_{Cn+D}$ ni rešljiv *v času* $\mathrm{o}(q^{(C-1)/2})$ *z večtračnimi NTS-ji.*

(v) Komplement problema D-$\textsc{Halt}^1_{Cn+D}$ ni rešljiv *v času* $\mathrm{o}(q^{(C-1)/4})$ *z večtračnimi NTS-ji.*

Če povzamemo, sta problema $\textsc{Halt}^1_{Cn+D}$ in D-$\textsc{Halt}^1_{Cn+D}$ co-NP-polna, njuna konedeterministična časovna zahtevnost je navzgor omejena z $\mathrm{O}(q^{C+2})$, navzdol pa z $\Omega(q^{0.25C-1})$, s čimer je navzdol omejena tudi njuna nedeterministična časovna zahtevnost.

Zgornja meja računske zahtevnosti v izreku 1.2.1 je dokazana s pomočjo prekrižnih zaporedij, predstavljenih v poglavju 5. Več o tem, kako jo dokažemo, je napisano spodaj v pregledu omenjenega poglavja. Spodnje meje računske zahtevnosti dokažemo s pomočjo polinomskih prevedb računsko zahtevnih odločitvenih problemov na probleme $\textsc{Halt}^1_{Cn+D}$ in D-$\textsc{Halt}^1_{Cn+D}$. Računsko zahtevne probleme dobimo z diagonalizacijo.

Opišimo še glavno idejo pri prevedbah. Recimo, da enotračni NTS M reši računsko zahteven odločitveni problem L. Potem lahko L rešimo tudi tako, da na vhodu w najprej skonstruiramo enotračni Turingov stroj M_w, ki je časovne zahtevnosti $Cn + D$ natanko tedaj, ko M zavrne w, in nato rešimo komplement problema $\textsc{Halt}^1_{Cn+D}$ za M_w. Če uspemo M_w skonstruirati z malo koraki, potem nam računska zahtevnost problema L zagotavlja spodnjo mejo računske zahtevnosti za komplement problema $\textsc{Halt}^1_{Cn+D}$. Glavna naloga stroja M_w je simulacija M na vhodu w, ki pa jo izvede le na dovolj dolgih (lastnih) vhodih, da ne krši časovne zahtevnosti $Cn + D$. To stori na naslednji način. Najprej s $(C-1)n+1$ koraki preveri, ali je vhod dovolj dolg, nato pa v naslednjih (največ) n korakih simulira M na w. Če M sprejme, M_w požene neskončno zanko in se več ne ustavi, sicer se ustavi. Izkaže se, da je za M_w ključno, da efektivno izmeri vhod z malo koraki in ob uporabi malega števila različnih stanj. Slednje je pomembno predvsem zato, ker manjši opis stroja M_w pomeni boljšo spodnjo mejo računske zahtevnosti za komplement problema $\textsc{Halt}^1_{Cn+D}$. Podrobnosti najdemo v poglavju 6.2.3. V poglavju 6.2.4 pokažemo, da je naš način merjenja dolžine vhoda stroja M_w optimalen, torej z našimi metodami ne moremo dokazati bistveno boljših spodnjih mej računske zahtevnosti.

Omenimo še poglavje 6.2.5, v katerem pokažemo, da tehnike, uporabljene pri dokazovanju izreka 1.2.1, relativizirajo, torej izključno z njimi ne moremo rešiti problema $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

Poglavje 5

V tem poglavju definiramo prekrižna zaporedja in glavne rezultate v zvezi z njimi. Definirana so le za enotračne Turingove stroje. *Prekrižno zaporedje, generirano z enotračnim Turingovim strojem M na ločnici i (glej sliko 2) po t korakih izračuna ζ na vhodu w*, je zaporedje stanj, v katerih M prečka ločnico i, če upoštevamo le prvih t korakov izračuna ζ na vhodu w. Pri tem predpostavljamo, da M v vsakem koraku najprej preide v naslednje stanje in šele nato premakne glavo. To zaporedje vsebuje vse informacije, ki jih M v prvih t korakih izračuna ζ prenese iz leve strani ločnice i na desno in obratno.

Slika 2: Oštevilčenje ločnic med celicami traku enotračnega Turingovega stroja. Osenčeni del je potencialni vhod dolžine 4.

Osrednja tehnika pri obravnavi prekrižnih zaporedij je *tehnika rezanja in lepljenja*, s katero dokažemo glavni rezultat v poglavju 5, izrek o kompaktnosti. Preden ga zapišemo v splošnem, si poglejmo posledico, po kateri je dobil ime.

> *Naj bosta C in D poljubni naravni števili in naj bo M enotračni NTS s q stanji. Potem je M časovne zahtevnosti $Cn + D$ natanko tedaj, ko M za vsak $n \leq \mathrm{O}(q^{2C})$ naredi največ $Cn + D$ korakov na vsakem izračunu na vhodih velikosti n.*

Z drugimi besedami, problem $\textsc{Halt}^1_{Cn+D}$ lahko rešimo tako, da za vhodni NTS M preverimo le število korakov, ki jih M naredi na vhodih velikosti največ $\mathrm{O}(q^{2C})$. Konstanta pod notacijo veliki O je polinomska v C in D (glej posledico 5.2.5). Čeprav nam ta posledica poda način za reševanje problema $\textsc{Halt}^1_{Cn+D}$, pa potrebujemo močnejši rezultat (izrek o kompaktnosti), da dokažemo izrek 1.2.1.

Podajmo sedaj dve oznaki, ki nastopata v izreku o kompaktnosti. Za enotračni NTS M, niz w in prekrižno zaporedje $\mathcal{C}$ intuitivno opišimo število $t_M(w, \mathcal{C})$. To je največje število korakov, ki jih lahko naredi M na strnjenem *delu* w nekega namišljenega vhoda, če upoštevamo le tiste izračune, ki na levem in desnem krajišču niza w proizvedejo enako prekrižno zaporedje $\mathcal{C}$. Če tak izračun ne obstaja, definiramo $t_M(w, \mathcal{C}) = -1$. Število $t_M(w, \mathcal{C})$ lahko enostavno izračunamo s simulacijo M na *delu* w, kot je razvidno iz njegove bolj formalne definicije v poglavju 5.2.1.

Za enotračni NTS M in naravno število n označimo s $\mathcal{S}_n(M)$ množico vseh začetkov prekrižnih zaporedij, ki jih lahko M ustvari na vhodih velikosti n na ločnicah $1, 2 \ldots n$. Izrek o kompaktnosti (izrek 5.2.1) je sledeč.

> *Naj bo M enotračni NTS s q stanji in naj bosta C in D naravni števili. Označimo $\ell = D + 8q^C$, $r = D + 12q^C$ in $\mathcal{S} = \bigcup_{n=1}^{\ell} \mathcal{S}_n(M)$. Potem velja:*
>
> *M je časovne zahtevnosti $Cn + D$ natanko tedaj, ko*
>
> *a) za vsak niz w dolžine največ ℓ in za vsak izračun ζ Turingovega stroja M na vhodu w velja $|\zeta| \leq C|w| + D$ ter*
>
> *b) za vsako prekrižno zaporedje $\mathcal{C} \in \mathcal{S}$ in za vsak niz w dolžine največ r, za katerega je $t_M(w, \mathcal{C}) \geq 0$, velja $t_M(w, \mathcal{C}) \leq C|w|$.*

Spomnimo se posledice, ki smo jo omenili pred izrekom, in ki nam pove, da je za preverjanje časovne zahtevnosti $Cn + D$ danega NTS-ja dovolj NTS simulirati na vhodih velikosti $\mathrm{O}(q^{2C})$. Opazimo, da v izreku o kompaktnosti nastopajo le nizi dolžine $\mathrm{O}(q^C)$. To je ključno v dokazu zgornje meje časovne zahtevnosti $\mathrm{O}(q^{C+2})$ za reševanje komplementa problema $\textsc{Halt}^1_{Cn+D}$ z večtračnimi NTS-ji (izrek 1.2.1).

Glavna tehnika pri dokazovanju izreka o kompaktnosti je tehnika rezanja in lepljenja. Najprej pokažemo, da poljuben NTS časovne zahtevnosti $Cn + D$ na dovolj velikih vhodih zmeraj na

nekaj ločnicah generira enaka prekrižna zaporedja. Če fiksiramo neki izračun na dovolj velikem vhodu, potem lahko ta vhod razrežemo na mestih, kjer se pojavi enako prekrižno zaporedje, in obravnavamo vsak del posebej. Dokažemo, da je dovolj obravnavati le (konstantno) kratke dele vhodov.

V poglavju 5.1.2 dokažemo standardni rezultat o enotračnih Turingovih strojih časovne zahtevnosti $\mathrm{o}(n \log n)$: taki Turingovi stroji generirajo le končno mnogo različnih prekrižnih zaporedij in odločijo le regularne jezike. Še več, vsak enotračni Turingov stroj časovne zahtevnosti $\mathrm{o}(n \log n)$ je tudi linearne časovne zahtevnosti. V poglavju 5.2.3 opišemo algoritem, ki sprejme naravni števili C in D ter enotračni NTM M in v primeru, da je M časovne zahtevnosti $Cn+D$, vrne ekvivalenten končni avtomat.

V poglavju 5.1.3 s pomočjo prekrižnih zaporedij dokažemo dve preprosti, dobro znani spodnji meji časovne zahtevnosti za reševanje problemov z enotračnimi NTS-ji. Prva je reda $\Theta(n \log n)$ za problem Primerjava dolžin, ki je očitno rešljiv v linearnem času z večtračnim DTS-jem. Druga spodnja meja še bolj poudari razliko med efektivnostjo enotračnih in večtračnih Turingovih strojev. Naj bo Palindrom naslednji odločitveni problem:

> Ali je dani niz palindrom, tj. ali se dani niz prebere od leve proti desni enako kot od desne proti levi?

Medtem ko lahko večtračni DTS reši problem Palindrom v linearnem času, pa za vsako funkcijo $T(n) = \mathrm{o}(n^2)$ enotračni NTS potrebuje več kot $\mathrm{O}(T(n))$ korakov. Kot zanimivost pokažemo tudi, da obstaja enotračni NTS časovne zahtevnosti $\mathrm{O}(n \log n)$, ki reši komplement problema Palindrom, s čimer pokažemo, da nedeterministična računska zahtevnost problema ne sovpada nujno z nedeterministično računsko zahtevnostjo njegovega komplementa.

Sorodna in uporabljena literatura

Poglavja 2, 3 in 4 vsebujejo standardno snov iz področja računske zahtevnosti in so v večini pokrita v knjigah Arore in Baraka [2] ter Sipserja [28]. Nekaj dokazov v teh poglavij je povsem avtorjevih, nekaj jih podrobneje sledi literaturi. Izreki in trditve so oblikovani tako, da ustrezajo kontekstu, v katerega so postavljeni. Dodatna literatura, ki je bila uporabljena, je na ustreznih mestih tudi navedena.

Poglavji 5 in 6 slonita na avtorjevih delih [10, 11]. Medtem ko je še veliko druge literature o prekrižnih zaporedjih (poglavje 5), pa je dodatno literaturo za poglavje 6 težje najti. A nekaj je vseeno obstaja. V sedemdesetih letih prejšnjega stoletja je Hájek [19] dokazal, da ne obstaja algoritem, ki bi za dani večtračni DTS povedal, ali je časovne zahtevnosti $n + 1$. Približno v istem obdobju je Hartmanis objavil monografijo [16], v kateri se v poglavju 6 ukvarja z vprašanjem: katere izjave o računski zahtevnosti je mogoče dokazati? V tem delu med drugim primerja razred jezikov, ki jih razpoznavajo Turingovi stroji časovne zahtevnosti $T(n)$, z razredom jezikov, ki jih razpoznavajo Turingovi stroji, katerih časovno zahtevnosto $T(n)$ lahko dokažemo. Naj omenimo še publikacijo Adachija, Iwate and Kasaija [1] iz leta 1984, v kateri predstavijo dobre deterministične spodnje meje časovne zahtevnosti reševanja problemov, ki so P-polni. Struktura tega rezultata je primerljiva strukturi izreka 1.2.1.

Študij prekrižnih zaporedij sega v šestdeseta leta prejšnjega stoletja, z začetniki Hartmanisom [15], Henniejem [17] in Trakhtenbrotom [30]. Leta 1968 je Hartmanis [15] dokazal, da enotračni DTS-ji časovne zahtevnosti $\mathrm{o}(n \log n)$ sprejmejo natanko regularne jezike. Ob tem je omenil, da je do istega rezultata neodvisno prišel tudi Trakhtenbrot [30, v ruščini]. V dokazu je Hartmanis

kot delni rezultat uporabil dejstvo, da enotračni DTS-ji časovne zahtevnosti $\mathrm{o}(n \log n)$ generirajo le prekrižna zaporedja dolžine $\mathrm{O}(1)$, nato pa je uporabil Henniejev rezultat [17], ki pravi, da taki Turingovi stroji razpoznavajo le regularne jezike. Kasneje (v osemdesetih letih prejšnjega stoletja) je Kobayashi [20] podal drugačen dokaz istega rezultata, a za razliko od Hartmanisovega pristopa, je njegov dokaz podal način, kako izračunati zgornjo mejo za dolžino prekrižnih zaporedij. Nedavno so Tadaki, Yamakami in Lin [29] pokazali, da tudi enotračni NTS-ji časovne zahtevnosti $\mathrm{o}(n \log n)$ generirajo le prekrižna zaporedja dolžine $\mathrm{O}(1)$, iz česar sledi, da sprejmejo le regularne jezike. Sledili so Kobayashijevemu dokazu in s tem implicitno podali način, kako izračunati zgornjo mejo za dolžino prekrižnih zaporedij, kar je ključno v dokazu izreka 6.1.10. Le-ta pravi, da za "lepe" funkcije $T(n) = \mathrm{o}(n \log n)$ obstaja algoritem, ki za dani enotračni NTS pove, ali je časovne zahtevnosti $T(n)$. V [26] je Pighizzini dokazal, da so NTM-ji časovne zahtevnosti $\mathrm{o}(n \log n)$ tudi linearne časovne zahtevnosti. Pregled lastnosti različnih tipov enotračnih Turingovih strojev linearne časovne zahtevnosti najdemo v [29].

Bibliography

[1] A. Adachi, S. Iwata, and T. Kasai. Some combinatorial game problems require $\Omega(n^k)$ time. *J. ACM*, 31(2):361–376, 1984.

[2] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[3] T. P. Baker, J. Gill, and R. Solovay. Relativizatons of the $\mathsf{P} = ? \mathsf{NP}$ question. *SIAM J. Comput.*, 4(4):431–442, 1975.

[4] S. Cabello and D. Gajser. Simple PTAS's for families of graphs excluding a minor. *Discrete Appl. Math.*, 189:41–48, 2015.

[5] N. Dershowitz and Y. Gurevich. A natural axiomatization of computability and proof of Church's thesis. *Bull. Symb. Log.*, 14(3):299–350, 2008.

[6] R. Diestel. *Graph Theory, 3rd ed.*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, 2005.

[7] B. Farhi. Nontrivial lower bounds for the least common multiple of some finite sequences of integers. *J. Number Theory*, 125(2):393 – 411, 2007.

[8] L. Fortnow and R. Santhanam. Robust simulations and significant separations. In *Proceedings of the 38th International Colloquim Conference on Automata, Languages and Programming - Volume Part I*, ICALP'11, pages 569–580. Springer-Verlag, 2011.

[9] D. Gajser. The limit of binomial means of a sequence. 2014. Preprint, arXiv:1407.4410.

[10] D. Gajser. Verifying time complexity of turing machines. *Theor. Comput. Sci.*, 600:86 – 97, 2015.

[11] D. Gajser. Verifying whether one-tape Turing machines run in linear time. *ECCC*, TR15-036, 2015.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[13] W. I. Gasarch. Guest column: the second $\mathsf{P} = ? \mathsf{NP}$ poll. *SIGACT News*, 43(2):53–77, 2012.

[14] D. Goldin and P. Wegner. The interactive nature of computing: Refuting the strong Church-Turing thesis. *Minds and Machines*, 18(1):17–38, 2008.

[15] J. Hartmanis. Computational complexity of one-tape Turing machine computations. *J. ACM*, 15(2):325–339, 1968.

[16] J. Hartmanis. *Feasible computations and provable complexity properties*. CBMS-NSF regional conference series in applied mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1978.

[17] F. C. Hennie. One-tape, off-line Turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[18] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *J. ACM*, 13(4):533–546, 1966.

[19] P. Hájek. Arithmetical hierarchy and complexity of computation. *Theor. Comput. Sci.*, 8(2):227–237, 1979.

[20] K. Kobayashi. On the structure of one-tape nondeterministic Turing machine time hierarchy. *Theor. Comput. Sci.*, 40(2-3):175–193, 1985.

[21] D. Kozen. *Theory of Computation*. Texts in Computer Science. Springer, 2006.

[22] R. J. Lipton. *The* $\mathsf{P} = \mathsf{NP}$ *Question and Gödel's Lost Letter*. Springer, 2010.

[23] Millennium prize problems. Retrieved Oct 11, 2015, from `http://www.claymath.org/millennium-problems/p-vs-np-problem`.

[24] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[25] W. J. Paul, N. Pippenger, E. Szemerédi, and W. T. Trotter. On determinism versus nondeterminism and related problems (preliminary version). In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 429–438, 1983.

[26] G. Pighizzini. Nondeterministic one-tape off-line Turing machines and their time complexity. *J. Autom. Lang. Comb.*, 14(1):107–124, 2009.

[27] J. I. Seiferas, M. J. Fischer, and A. R. Meyer. Separating nondeterministic time complexity classes. *J. ACM*, 25(1):146–167, 1978.

[28] M. Sipser. *Introduction to the Theory of Computation, 2nd ed.* PWS Publishing Company, 1997.

[29] K. Tadaki, T. Yamakami, and J. C. H. Lin. Theory of one-tape linear-time Turing machines. *Theor. Comput. Sci.*, 411(1):22–43, 2010.

[30] B. A. Trakhtenbrot. Turing computations with logarithmic delay. *Algebra i Logica 3*, pages 33–48, 1964. In Russian.

[31] S. Žák. A Turing machine time hierarchy. *Theor. Comput. Sci.*, 26(3):327 – 333, 1983.