CORVINUS UNIVERSITY OF BUDAPEST

Attila Tasnádi

COMPUTER SCIENCE

with applications to economics

Lecture Notes

Budapest, August 31, 2026

Contents

Preface **3**

1 Introduction **5**
1.1 Notations . . . 5
1.2 The Θ and O notations . . . 5
1.3 A brief introduction to the Python programming language . . . 6
1.3.1 Interesting features . . . 7
1.3.2 Control statements . . . 7
1.3.3 Function and function call . . . 8
1.4 Exercises . . . 8

2 Computability Theory **9**
2.1 Formal languages . . . 9
2.2 Turing machine (single-tape) . . . 10
2.3 Simulating a Turing machine on a computer . . . 12
2.4 Formal description of Turing machine operation . . . 13
2.5 Basic program constructs on a Turing machine . . . 15
2.6 k-tape Turing machine . . . 16
2.7 Recursiveness and recursive enumerability . . . 18
2.8 Universal Turing machine . . . 19
2.9 Algorithmically undecidable problem . . . 20
2.10 RAM machine . . . 21
2.11 Exercises . . . 22

3 Boolean Logic **25**
3.1 Conjunctive and disjunctive normal forms . . . 27
3.2 Resolution . . . 28
3.3 Boolean functions . . . 29
3.4 Boolean circuits . . . 29
3.5 Exercises . . . 31

4 Computational Complexity **33**
4.1 Nondeterministic Turing Machine . . . 36
4.2 Characterizing NP with the Witness Theorem . . . 38
4.3 NP-Completeness . . . 39
4.4 NP-Complete Problems . . . 41
4.5 Other Notable NP-Complete Problems . . . 43
4.6 Optimization Problems . . . 44
4.7 Handling NP-Complete Problems . . . 44

4.8 Approximation Algorithms . . . 45
4.9 Exercises . . . 47

5 On the boundary of economics and computer science 49
5.1 Algorithmic Mechanism Design . . . 49
5.1.1 Basic concepts of non-cooperative game theory . . . 49
5.1.2 Mechanism Design on a Network . . . 51
5.2 Braess Paradox . . . 53
5.2.1 Braess Paradox Detection . . . 54
5.3 Combinatorial Auctions . . . 55
5.3.1 Winner Determination Problem . . . 57
5.3.2 Communication between Auctioneer and Bidders . . . 58
5.4 Social Choice . . . 59
5.4.1 Some Notable Voting Procedures . . . 59
5.4.2 Desirable Properties of Social Choice Functions . . . 60
5.4.3 Arrow's Impossibility Theorem . . . 61
5.4.4 Gibbard–Satterthwaite Theorem . . . 61
5.4.5 Computational Complexity of Strategic Manipulation . . . 61
5.5 Fair Division Games . . . 62
5.5.1 Fair Division Problem . . . 62
5.5.2 Banach–Knaster Procedure . . . 64
5.5.3 Even–Paz Procedure . . . 65
5.5.4 Selfridge–Conway Procedure . . . 67
5.5.5 Brams–Taylor–Zwicker Procedure . . . 68
5.6 Exercises . . . 68

Bibliography 71

Preface

This note was prepared for first-year master students in Business Informatics at the Corvinus University of Budapest. The subject aims to provide a brief insight into two classical areas of theoretical computer science (computability theory, complexity theory). The qualitative results of the field contribute to classifying the difficulty of problems arising in practical life, and indicate the appropriate approach to solve problems of different types. In terms of the necessary mathematical knowledge, we provide an introduction to mathematical logic, and also take a look at some popular interdisciplinary areas of computer science and economics (e.g., algorithmic mechanism design, combinatorial auctions).

Why is theoretical computer science interesting?

- It is the foundation of many things (algorithms, programming languages, etc.)!
- It examines fundamental questions (solvability of tasks, types of machines, resource requirements, etc.)!
- It deals with problems relevant to practice (route planning, internet modeling, electronic auctions, etc.)!
- It develops formal thinking (programs and systems are formal objects)!
- The results and thought processes are beautiful!

Recommended literature:

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford(2009): *Introduction to Algorithms*, MIT Press.
- Papadimitriou, Christos H. (1994): *Computational Complexity*, Pearson University Press.
- Arora, Sanjev; Barak, Boaz (2009): *Computational Complexity: A Modern Approach*, Cambridge University Press.
- Moore, Christopher; Mertens, Stephan (2017): *The Nature of Computation*, Oxford University Press.
- Rothe, Jörg (2024): *Economics and Computation*, Springer

The author welcomes any comments on this note at the email address `attila.tasnadi@uni-corvinus.hu`.

Finally, I would like to thank Zsuzsanna Horváth for her help in preparing this lecture notes, and also Kristóf Ábele-Nagy, László Csató, Izabella Kuncz, and Dóra Petróczy for their comments on the Hungarian version of this lecture notes.

Budapest, August 31, 2026.

Attila Tasnádi

Chapter 1

Introduction

In this chapter, we introduce the notations used throughout this note and the O and Θ notations used to characterize the running time of algorithms. We illustrate the O and Θ notations on the bubble sort algorithm. Furthermore, the chapter provides a short introduction to the Python programming language, in which we also provide several sample programs.

1.1 Notations

Let $\mathbb{N} = \{0, 1, \ldots\}$ denote the natural numbers, $\mathbb{Z}$ the integers, $\mathbb{Z}_+$ the positive integers, $\mathbb{R}$ the real numbers, $\mathbb{R}_+$ the positive real numbers. The infinity symbol is ∞.

For a finite set A, $|A|$ denotes the number of elements in A. For an infinite set A, $|A| = \infty$. $A \subseteq B$ denotes that A is a subset of B, while $A \subset B$ denotes that A is a proper subset of B. The empty set is denoted by $\emptyset$. The set of all subsets of a set A is denoted by $\mathcal{P}(A)$ and is called the *power set* of A. For example, $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

The symbol := is used in the text to denote definitional equality. Thus, for example, for any $n \in \mathbb{Z}_+$, $\mathbb{Z}_n := \{0, 1, \ldots, n-1\}$ means that henceforth $\mathbb{Z}_n$ denotes the set of integers from 0 to $n-1$. In our programs, := is the assignment operation. For any $i, j \in \mathbb{Z}$, $[i..j] := \{i, i+1, \ldots, j-1, j\}$, where specifically $[i..j] = \emptyset$ if $j < i$. Real intervals are denoted as follows: $[a, b] := \{x \in \mathbb{R} \mid a \leq x \leq b\}$, $[a, b) := \{x \in \mathbb{R} \mid a \leq x < b\}$, $(a, b] := \{x \in \mathbb{R} \mid a < x \leq b\}$ and $(a, b) := \{x \in \mathbb{R} \mid a < x < b\}$, where at the open ends of the interval we also allow the ∞ symbol. Thus, for example, $(-\infty, \infty) = \mathbb{R}$.

The end of proofs of theorems, propositions, and lemmas (auxiliary theorems) is marked by $\Box$.

1.2 The Θ and O notations

We will need a concise characterization of how the operation count of a given program grows as the input size increases. The Θ and O notations to be introduced serve this purpose.

Consider the so-called bubble sort as an introductory example. We wish to sort the elements of an array x of n elements storing integers in increasing order. To do this, we start from the beginning of the array and, based on pairwise comparisons, perform the necessary swaps to move the largest element to the very end of the array. After this, we only need to deal with the first $n-1$ elements, whose largest element we move similarly to the $n-1$th position of the array. We continue the procedure until only the first element remains. The value of the outer loop variable i decreases one by one from n to 2. Then, in the $n-i+1$th iteration of the outer loop body, the ith largest element is moved to the ith position of the array. The operation count of bubble sort could be characterized by the number of executions

of the inner loop body, which swaps two adjacent elements if they are in the wrong order. Note that this value depends only on the number of elements in the array x. Since for a given i the inner loop body is executed $i-1$ times, the sought value is $\sum_{i=2}^{n}(i-1) = \frac{1}{2}(n-1)n$. From this, we can conclude that the operation count of bubble sort increases "in order of magnitude" with the square of the number of elements in the array x. The order of magnitude of the operation count is defined using the so-called asymptotically tight bound.

Definition 1.1. The sequence $f : \mathbb{Z}_+ \to [0, \infty)$ has $g : \mathbb{Z}_+ \to [0, \infty)$ as an *asymptotically tight bound*, if there exist $c_1, c_2 \in \mathbb{R}_+$ and $n_0 \in \mathbb{Z}_+$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. Then we write $f = \Theta(g)$.

Clearly, if the limit $\lim_{n\to\infty} f(n)/g(n) \in \mathbb{R}_+$ exists, then g is an asymptotically tight bound for f. However, the existence of the limit of the sequence $f(n)/g(n)$ is not necessary for g to be an asymptotically tight bound for f.

Returning to bubble sort, if we denote by $T(n)$ how many times the inner loop body of bubble sort is executed for an array of n elements, then $T = \Theta(n^2)$. However, unlike bubble sort, for many procedures we can only estimate the order of magnitude of the operation count from above.

Definition 1.2. The sequence $f : \mathbb{Z}_+ \to [0, \infty)$ has $g : \mathbb{Z}_+ \to [0, \infty)$ as an *asymptotic upper bound*, if there exist $c \in \mathbb{R}_+$ and $n_0 \in \mathbb{Z}_+$ such that $f(n) \leq c g(n)$ for all $n \geq n_0$. Then we write $f = O(g)$.

For bubble sort, for example, we can write $T = O(n^5)$, but also $T = O(n^2 \log_2 n)$ or $T = O(n^2)$.

1.3 A brief introduction to the Python programming language

This note assumes that the reader has already mastered one or two programming languages (such as C++ or C#), so it actually only covers the syntactic peculiarities of the Python language.

The reader may rightly wonder why we use Python specifically. The following advantages speak for using Python.

- It is easy to learn compared to other languages.
- Due to its conciseness, algorithms are more readable, and thus codes require less space.
- Code readability plays a prominent role in Python.
- It supports procedural, object-oriented, and functional programming.
- It is system-independent (Windows, Linux, Mac OS X).
- It is freely available, which also applies to its background libraries (web server, numerical and symbolic computations, simulation, etc.)
- It is widespread in scientific computing. Its background libraries rival professional software, and moreover, they are easier to program using the Python programming language.

It is worth saying a few words about the development and spread of the Python language. The Python programming language was started by Guido van Rossum in the 1980s, and he still determined the development directions of the language until 2020. For this reason, he was referred to as the benevolent dictator of the Python community. The first implementation of

Python began in 1989, and the first version, Python 1.0, was completed by 1994. The currently used version is 3.14.7 (August 15, 2026). Python is widely used, for example, by YouTube, the original BitTorrent client, Google, Yahoo!, CERN, and NASA.

Readers interested in a deeper understanding of Python are recommended Mark Summerfield (2009): *Programming in Python 3*, Pearson, India.

Python can be downloaded from `https://www.python.org/downloads/`. After downloading, for first use we recommend the Python GUI interpreter; a more comfortable development environment is provided by Anaconda `https://www.anaconda.com/products/individual`.

1.3.1 Interesting features

- Dynamically typed language.
- Variables are object references.
- `tuple` is constant, while `list` is mutable. Respectively `a=(1,5,3)`, and `a=[1,5,3]`.
- List operations: `len(lista)`, `lista.append(x)`, `lista.extend(lista2)`, etc.
- Identity check: after `a=[1,2]; b=[1,2]`, `print(a is b)` is `False`, but after `a=[1,2]; b=a`, `print(a is b)` is `True`.
- Membership operation: after `a=[2,1,5,8]`, `5 in a` is `True`. Here `in` performs a linear search.

1.3.2 Control statements

The syntax of branching:

```
if boolean_expression1:
     code_block1
elif boolean_expression2:
     code_block2
else:
     code_block
```

Note that code blocks are preceded by a :. Another interesting feature is that indentation matters! Any number of `elif` can be used, and the `else` is optional.

The `while` loop syntax:

```
while boolean_expression:
     code_block
```

Note that the code block can contain `break` and `continue` statements, and at the end an `else` clause. The optional `else` code block is executed if the loop terminated normally, meaning we did not exit the loop body via a `break` statement or an exception.[1]

The `for` loop syntax:

```
for variable in iterable_object:
     code_block
```

The `for` loop can also contain `break` or `continue` statements, and at the end an `else` clause. The `iterable_object` can be, for example, a list (e.g., `for i in [1,2,3]:`)

[1] We will talk about exception handling later.

1.3.3 Function and function call

The syntax for defining a function:

```
def function_name(parameters):
    code_block
```

The `code_block` can contain a `return value` statement, where the `value` is optional. Example:

```
def hypotenuse(a,b):
    return pow(a**2+b**2,0.5)
```

1.4 Exercises

Exercise 1.1. Write the Python code for the program that solves the two-variable linear Diophantine equation!

Exercise 1.2. Let a graph be given by its adjacency matrix and a coloring of its vertices in a list. Write a Python program to decide whether the list containing the coloring specifies a three-coloring of the given graph!

Chapter 2

Computability Theory

Even in the 1980s, when personal computers were not yet widespread, a shop assistant, when asked what the Commodore VC 20 computer on the shelf — which had a clock speed of 1 MHz and 5 kB of memory — could be used for, gave the surprising answer: for everything. In 1928, David Hilbert, even before the advent of modern computers, posed a more specific question: "can an algorithm be given that, starting from the axioms and using the rules of logic, decides the truth of a (first-order logic) statement." The problem became famous as the "Entscheidungsproblem", for which, surprisingly, the answer is negative.

In this chapter, we provide a mathematical model of the computer, with the help of which we delineate the set of algorithmically solvable problems. The model we describe was formulated by Alan Turing in 1936, with which he pointed out the existence of algorithmically undecidable problems. Remarkably, his published result preceded digital computers. According to von Neumann, the modern computer's conceptual design owes to the universal Turing machine introduced by Turing. Alan Turing was the most decisive figure among the code-breakers at Bletchley Park during World War II (including breaking the German Enigma codes). In the field of artificial intelligence, the Turing test is named after him. Alan Turing is often referred to as the father of computer science and artificial intelligence; in his honor, the ACM (Association for Computing Machinery) has awarded the Turing Award annually since 1966, considered the Nobel Prize of computer science.

2.1 Formal languages

To decide whether a task is algorithmically solvable, first of all, the mathematical formulation of the task is required. Formal languages enable us to specify problems in a way that the simple mathematical models of the computer (which we will discuss later) can process them.

To define a formal language, we need an alphabet, which consists of the possible symbols of our language. Thinking in terms of natural languages, our alphabet could be, for example, the letters and symbols of the Hungarian language, or for digital computers, the set of bits, usually identified with the digits $0, 1$. We denote our alphabet by Σ, which is the finite and non-empty set of our formal language. Σ is also referred to as an alphabet.

By themselves, the symbols[1] of our language do not get us far, but the words and sentences formed from them will be more useful for us. In the following, a *word* is nothing but a finite sequence of symbols, and a sentence is a synonym for word, which does not cause a problem because, considering natural languages, our alphabet may include the empty symbol. Specifically, the "word" consisting of no symbols is the *empty word*, denoted by ε. The set of

[1] Sign and symbol are synonyms.

words that can be formed from our alphabet Σ is denoted by Σ^*.

Example 2.1. For the alphabet $\Sigma := \{0, 1\}$, $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$, where we listed the words systematically ordered by word length.

In the above example, we already used the concept of word length, which can naturally be defined as follows: the *length* of the word $\alpha = a_1 a_2 \dots a_n$ is $|\alpha| = n$ and $|\varepsilon| := 0$. By convention, the first letters of the Arabic alphabet (a, b and c) denote a symbol, while Greek letters denote a word.

We define further operations on words. The *concatenation* joins two words, i.e., we write the two given words one after the other. Formally, if $\alpha = a_1 a_2 \dots a_m$ and $\beta = b_1 b_2 \dots b_n$ are words, then their concatenation is $\alpha\beta := a_1 a_2 \dots a_m b_1 b_2 \dots b_n$. If a word is written repeatedly, then we speak of a *power* of the given word, i.e., $\alpha^0 := \varepsilon$ and $\alpha^i := \alpha^{i-1}\alpha$, where $i \in \mathbb{Z}_+$.

The word α is a *subword* of the word β if α can be found in β, more precisely, α is a subword of β if there exist words γ and δ such that $\beta = \gamma\alpha\delta$. Using the alphabet of the Hungarian language, "talál" is a subword of "megtalálható". A prefix is a special subword that is found at the beginning of another word, i.e., α is a *prefix* of β if there exists a word γ such that $\beta = \alpha\gamma$.

As finite sequences of symbols, we obtain many "meaningless" words. So, in some sense, we need to restrict the set of words. This leads to the concept of a formal language.

Definition 2.1. A *formal language* is $L \subseteq \Sigma^*$.

By definition, the set of possible languages is still large, since any subset of words is a formal language. At this point, we cannot do more than that, since we have not given word formation rules (syntax) and we have not assigned meaning (semantics) to the words. In this sense, the adjective formal emphasizes that we are not talking about a set of meaningful words or sentences in the everyday sense.

For example, over the alphabet $\Sigma = \{0, 1\}$, $L = \{1, 11, 111, \dots\}$ or the empty language $L_\emptyset = \emptyset$ are also formal languages.

2.2 Turing machine (single-tape)

The Turing machine is a very simple computational model that helps to categorize the difficulty of problems well. Surprisingly, despite its simplicity, it is one of the most powerful computational models. Figure 2.1 points out why we say the Turing machine is simple. As can be seen, it consists only of a finite-state control, a read/write head, and an input/output tape. In one step, the read/write head can move at most one position left or right. Nevertheless, the known computational models capturing the intuitive concept of an algorithm (step-by-step execution of operations, branching, loops, recursion, etc.) are not capable of performing tasks that are unsolvable by a Turing machine.

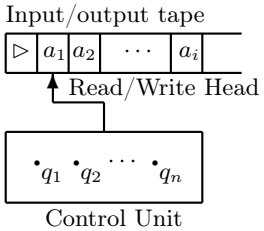


Figure 2.1: Schematic diagram of a Turing machine

The *significance* of the Turing machine is supported by the following properties.

- It is a mathematical model of a computer.
- No more powerful computational model is known.
- Whatever is solvable in C, for example, is also solvable by a Turing machine. Branching, loops, recursion, etc. are solvable.
- It is a very simple model.

Attempts to mathematically define the concept of an algorithm have not led to a computational model more powerful than the Turing machine, but this does not prove the non-existence of a more powerful computational model. The problem is caused by the mathematical indefinability of the concept of an algorithm, so we identify algorithmic solvability with solvability by a Turing machine. However, based on our experience, we accept the *Church thesis* or Church–Turing thesis, which states that there is no computational model more powerful than the Turing machine.

After the long introduction, let's turn to the mathematical definition of the Turing machine.

Definition 2.2. $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a *Turing machine* if

(i) Q is the finite non-empty *set of states* of the control unit;

(ii) Σ is the *input alphabet*;

(iii) Γ is the *tape alphabet*, where

- $\Sigma \subset \Gamma$,
- $Q \cap \Gamma = \emptyset$, furthermore
- $\sqcup \in \Gamma \setminus \Sigma$ contains the blank symbol and
- $\rhd \in \Gamma \setminus \Sigma$ contains the (tape) start symbol in Γ;

(iv) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}$ is the *transition function*, a partial function, for which we assume that $\delta(q, \rhd) = (p, \rhd, \rightarrow)$ holds for every state $q \in Q$ for some state $p \in Q$;

(v) $q_0 \in Q$ is the *start state*;

(vi) $F \subseteq Q$ is the set of *final states* (also called accepting states).

We denote the set of movement directions by D in the following, i.e., $D := \{\leftarrow, -, \rightarrow\}$.

The transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times D$ is essentially the "program" of the *Turing machine* $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, which specifies that if $\delta(q, a) = (p, b, d)$, then M in state q reading symbol $a \in \Gamma$ goes to state p, writes symbol b, and moves its head in direction d. The computation of one value of the transition function corresponds to one step of the Turing machine.

Initially, the tape of M contains a word $x \in \Sigma^*$ and the control unit is in the start state q_0, then it proceeds step by step according to the partial function δ until it reaches an undefined (q, a) state and read symbol pair (then it stops). On certain words, it may happen that the Turing machine does not stop after a finite number of steps. In this case, the Turing machine runs indefinitely, of which an infinite loop is a possible case.

As a recognizer, M *accepts* a word $x \in \Sigma^*$ if it stops in a final state during its processing. The result of the computation is the word found on the tape.

Consider the following Turing machine as an example.

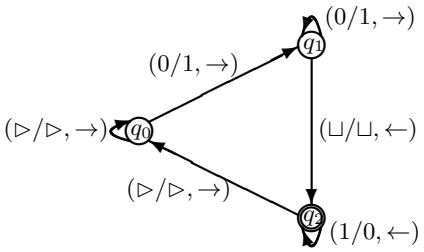


Figure 2.2: Turing machine with transition diagram

Example 2.2. The Turing machine $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \sqcup, \triangleright\}, \delta, q_0, \{q_2\})$ has its transition function given by the transition diagram in Figure 2.2.

In Figure 2.2, the arrows show which state can transition to which, provided that the read/write head reads the symbol before the slash (/) on the arrow labels. Then the symbol after the / is written, and the third symbol shows in which direction the read/write head moves. For example, the Turing machine given by Figure 2.2 in state q_0 reading symbol 0 goes to state q_1, while writing symbol 1 and moving the read/write head to the right.

The transition function can also be given in tabular form. Moreover, this form is the most practical when simulating the Turing machine with any computer program. The transition function shown in Figure 2.2 is given in tabular form in Table 2.1. The row of the table is determined by the current state, the column by the symbol under the read/write head, while the three elements of the corresponding entry give the new state, the symbol to be written, and the movement direction of the read/write head. The absence of an entry at a current state and read symbol indicates that the transition function is undefined.

$Q\backslash\Gamma$	0	1	$\sqcup$	$\triangleright$
q_0	$q_1, 1, \rightarrow$			$q_0, \triangleright, \rightarrow$
q_1	$q_1, 1, \rightarrow$		$q_2, \sqcup, \leftarrow$	
q_2		$q_2, 0, \leftarrow$		$q_0, \triangleright, \rightarrow$

Table 2.1: Turing machine given by a table

2.3 Simulating a Turing machine on a computer

We can easily convince ourselves that a Turing machine can be simulated by a computer program, if we disregard the fact that the memory of real computers cannot be expanded arbitrarily. Without loss of generality, we can assume that for the Turing machine M, $Q = \{1, \ldots, m\}$, $\Gamma = \{1, \ldots, n\}$, $q_0 = 1$ and $F = \{m - t + 1, \ldots, m\}$. For the input/output tape, let's allocate an array ($tape[]$) capable of storing symbols from the tape alphabet. We will need a variable ($head$) that indicates which cell of our tape the head is on. In another variable (q), we store the current state of the control unit. In a three-dimensional array $delta[m][n][3]$, we store the transition function, where the first dimension is the current state, the second is the read tape symbol, and the third is the triple of new state, symbol to write, movement direction $(-1, 0, 1)$.

The outline of the simulator program:

1. Initialization: copy the input word into the *tape* array, the start state 1 into q, and let $head := 1$.

2. Read the tape symbol: $a := tape[head]$.
3. Stop if $delta[q, a]$ is undefined.
4. Compute the value of the transition function: let $p := delta[q, a, 1]$, $b := delta[q, a, 2]$, $d := delta[q, a, 3]$.
5. Execute the transition: $q := p$, $tape[head] := b$, $head := head + d$.
6. Go back to step 2.

A computer is Turing-complete if, disregarding memory limitations, it can simulate any Turing machine. The first Turing-complete computer was Konrad Zuse's Z3 (1941), which was still an electromechanical computer. Its Turing-completeness was proven by Rojas (1998). The first Turing-complete and also planned electronic computer was the ENIAC (1946).

2.4 Formal description of Turing machine operation

The state of a Turing machine can be described by its so-called configuration. It includes the state of the control unit, the word on the tape, and the head position that splits it. If we know the configuration of a Turing machine, then how it got to that configuration is completely irrelevant from the perspective of its further operation.

Definition 2.3. A *configuration* of M is the triple $(q, \alpha a, \beta)$, where $q \in Q$, $a \in \Gamma$ is the symbol under the head, $\alpha \in \Gamma^*$ is the word to the left of the head, and $\beta = \varepsilon$ if only blanks are to the right of the head, otherwise $\beta \in \Gamma^*$ is the word from the head to the last non-blank symbol to the right.

The start configuration and the halting configuration of the Turing machine are distinguished.

Definition 2.4. $(q_0, \triangleright, x)$ is the *start configuration*, where $x \in \Sigma^*$ is the word to be processed. $(q, \alpha a, \beta)$ is a *halting configuration* if $\delta(q, a)$ is undefined.

The transition function establishes a direct connection between two configurations if one step of the Turing machine (one transition function computation) leads from one to the other.

Definition 2.5 (immediate configuration transition)**.** From configuration $(q, \alpha a, \beta)$, another configuration is immediately reachable as follows:

$$(q, \alpha a, \beta) \vdash \begin{cases} (p, \alpha bc, \gamma), & \text{if } \delta(q, a) = (p, b, \rightarrow), \beta = c\gamma; \\ (p, \alpha b\sqcup, \varepsilon), & \text{if } \delta(q, a) = (p, b, \rightarrow), \beta = \varepsilon; \\ (p, \alpha b, \beta), & \text{if } \delta(q, a) = (p, b, -); \\ (p, \alpha, b\beta), & \text{if } \delta(q, a) = (p, b, \leftarrow), \end{cases}$$

where $\vdash$ is the symbol for immediate configuration transition.

Recall that the definition of δ included that $\delta(q, \triangleright) = (p, \triangleright, \rightarrow)$ for any state $q \in Q$ for some state $p \in Q$, i.e., from the beginning of the tape, the machine must move in the direction of the tape according to the definition of the Turing machine's transition function. So when reading the start symbol, the movement direction cannot be $-$ or $\leftarrow$. Furthermore, when moving right, the reached configuration must be described differently depending on whether there is a non-blank symbol to the right of the head or not, because if there is no non-blank symbol to the right of the head, then the head steps onto a blank symbol.

The indirect configuration transition is a sequence of immediate configuration transitions.

Definition 2.6. The configuration $(p, \mu b, \nu)$ is *reachable in* $2 \leq k$ *steps* from configuration $(q, \alpha a, \beta)$, if there exists an intermediate configuration sequence $(r_i, \gamma_i c_i, \tau_i)_{i=1}^{k-1}$ such that

$$(q, \alpha a, \beta) \vdash (r_1, \gamma_1 c_1, \tau_1) \vdash \cdots \vdash (r_{k-1}, \gamma_{k-1} c_{k-1}, \tau_{k-1}) \vdash (p, \mu b, \nu),$$

which we denote by $(q, \alpha a, \beta) \vdash^k (p, \mu b, \nu)$.

Since to reach a configuration from itself, we need to take zero steps, let any configuration be reachable from itself in 0 steps.

After this, we can define the concept of reachability.

Definition 2.7. The configuration $(p, \mu b, \nu)$ is *reachable* from configuration $(q, \alpha a, \beta)$ if there exists a number of steps $k \in \mathbb{N}$ such that $(p, \mu b, \nu)$ is reachable from $(q, \alpha a, \beta)$ in k steps.

We denote that $(p, \mu b, \nu)$ is reachable from $(q, \alpha a, \beta)$ by $(q, \alpha a, \beta) \models (p, \mu b, \nu)$.

With the help of the concept of reachability, the language recognized by a Turing machine can be formally defined.

Definition 2.8. The Turing machine M *accepts* a word $x \in \Sigma^*$ if starting from the start configuration $(q_0, \triangleright, x)$ it halts in a final state.

The *language* L_M *recognized* by the Turing machine M is the set of accepted words in Σ^*. The language recognized by the Turing machine M is sometimes also denoted by $L(M)$. Formally:

$$L_M = \{x \in \Sigma^* \,|\exists (p, \mu b, \nu) : (q_0, \triangleright, x) \models (p, \mu b, \nu) \wedge (p, b) \notin \mathcal{D}_\delta \wedge p \in F\}.$$

where $\mathcal{D}_\delta$ denotes the set of those $Q \times \Gamma$ pairs for which δ is defined.

The Turing machine can also be used for function value computation, or, in other words, for transforming the input word. The result of the computation is found on the input/output tape, where the result is a word from the input alphabet. Since at the end of the computation process, besides input alphabet symbols, only tape alphabet symbols may be on the tape, the result of the computation is identified with the first sequence of input alphabet symbols found after the last start symbol.

Definition 2.9. The *partial function* $f_M : \Sigma^* \to \Sigma^*$ *computed by* M assigns to a word $x \in \Sigma^*$ the word (value) $y \in \Sigma^*$ found on the tape at halting, if $(q_0, \triangleright, x) \models (p, \mu b, \nu)$ for some halting configuration $(p, \mu b, \nu)$ and $\mu b \nu \in (\Gamma \setminus \Sigma)^* \, y \, (\Gamma \setminus (\Sigma \cup \{\triangleright\})) \, (\Gamma \setminus \{\triangleright\})^*$.

After formally defining the language recognized by a Turing machine and the function computed by it, let's decide whether the Turing machine given in Example 2.2 accepts the words 01110 and 000. We document the progress of the computation with configuration transitions. Start with the word 01110:

$$(q_0, \triangleright, 01110) \vdash (q_0, \triangleright 0, 1110) \vdash (q_1, \triangleright 11, 110).$$

Since the Turing machine did not halt in a final state, it rejects the word 01110. Now turn to the processing of the word 000.

$$\begin{aligned}(q_0, \triangleright, 000) &\vdash (q_0, \triangleright 0, 00) \vdash (q_1, \triangleright 10, 0) \vdash (q_1, \triangleright 110, \varepsilon) \vdash (q_1, \triangleright 111 \sqcup, \varepsilon) \\ &\vdash (q_2, \triangleright 111, \varepsilon) \vdash (q_2, \triangleright 11, 0) \vdash (q_2, \triangleright 1, 00) \vdash (q_2, \triangleright, 000) \vdash (q_0, \triangleright 0, 00).\end{aligned}$$

It can be seen that M falls into an infinite loop, so it does not accept the word 000.

Let's look at another example!

Example 2.3. Given $M = (\{q_0, q_1\}, \{1\}, \{1, \sqcup, \rhd\}, \delta, q_0, \{q_1\})$, where

$$\delta(q_0, \rhd) = (q_0, \rhd, \rightarrow),\ \delta(q_0, 1) = (q_0, 1, \rightarrow) \text{ and } \delta(q_0, \sqcup) = (q_1, 1, -).$$

Determine L_M and f_M!

Let's examine the operation of the Turing machine M for example with the word $x = 111$:

$$\begin{aligned}(q_0, \rhd, 111) &\vdash (q_0, \rhd 1, 11) \vdash (q_0, \rhd 11, 1) \vdash (q_0, \rhd 111, \varepsilon)\\ &\vdash (q_0, \rhd 111\sqcup, \varepsilon) \vdash (q_1, \rhd 1111, \varepsilon).\end{aligned}$$

It can be seen that $L_M = \{1\}^*$ and $f_M(x) = x1$. If we consider the input word as a number given in unary encoding, then M increments the input value by one.[2]

2.5 Basic program constructs on a Turing machine

We have already mentioned that any task that can be performed by a traditional computer can be performed by a Turing machine. In the practice exercises, we will see that simple arithmetic, comparison, and copying operations can be performed by Turing machines. If we can also implement the basic program constructs with Turing machines, we can indeed be reassured that the tasks solvable with traditional computers are also solvable with Turing machines. For this, the implementation of sequence, branching, and loop is sufficient, since recursive function calls can always be eliminated.[3]

Let's start with the sequence. Let $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, F_2)$ be two Turing machines such that $Q_1 \cap Q_2 = \emptyset$. Then the *sequence* $(M_1; M_2)$ of the two Turing machines is the Turing machine

$$M = (Q_1 \cup Q_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, q_1, F_2),$$

whose transition function is

$$\delta(q, a) = \begin{cases} \delta_1(q, a), & \text{if } (q, a) \in \mathcal{D}_{\delta_1};\\ (q_2, a, -), & \text{if } (q, a) \notin \mathcal{D}_{\delta_1}, q \in F_1 \text{ and } a \in \Gamma_2;\\ \delta_2(q, a), & \text{if } (q, a) \in \mathcal{D}_{\delta_2};\\ \text{undefined}, & \text{otherwise.}\end{cases}$$

Now let's turn to the implementation of branching. Let the Turing machines $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, F_2)$ be given, whose state sets are pairwise disjoint, and $F = \{q_i, q_n\}$. The Turing machine implementing the *branching* $M' = (M : M_1; \neg M : M_2)$ is the Turing machine $M' = (Q \cup Q_1 \cup Q_2, \Sigma, \Gamma \cup \Gamma_1 \cup \Gamma_2, \delta', q_0, F_1 \cup F_2)$, whose transition function is

$$\delta'(q, a) = \begin{cases} \delta(q, a), & \text{if } (q, a) \in \mathcal{D}_{\delta};\\ (q_1, a, -), & \text{if } (q, a) \notin \mathcal{D}_{\delta}, q = q_i \text{ and } a \in \Gamma_1;\\ (q_2, a, -), & \text{if } (q, a) \notin \mathcal{D}_{\delta}, q = q_n \text{ and } a \in \Gamma_2;\\ \delta_1(q, a), & \text{if } (q, a) \in \mathcal{D}_{\delta_1};\\ \delta_2(q, a), & \text{if } (q, a) \in \mathcal{D}_{\delta_2};\\ \text{undefined}, & \text{otherwise.}\end{cases}$$

Finally, we give the implementation of the loop using two Turing machines. Let the Turing machines $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ and $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, F_1)$ be given, for which $Q \cap Q_1 =$

[2] In unary encoding, the number of ones determines the value of the natural number.
[3] We will not prove this latter statement.

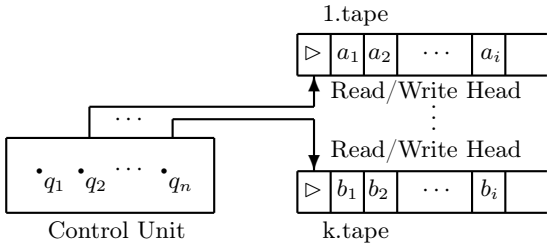
$\emptyset$ and $F = \{q_i, q_n\}$. The Turing machine implementing the *loop* $M' = DO(M : M_1)$ is the $M' = (Q \cup Q_1, \Sigma, \Gamma \cup \Gamma_1, \delta', q_0, \{q_n\})$ Turing machine, whose transition function is

$$\delta'(q,a) = \begin{cases} \delta(q,a), & \text{if } (q,a) \in \mathcal{D}_\delta; \\ (q_1, a, -), & \text{if } (q,a) \notin \mathcal{D}_\delta, q = q_i \text{ and } a \in \Gamma_1; \\ \delta_1(q,a), & \text{if } (q,a) \in \mathcal{D}_{\delta_1}; \\ (q_0, a, -), & \text{if } (q,a) \notin \mathcal{D}_{\delta_1}, q \in F_1 \text{ and } a \in \Gamma; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Corollary 2.1. *Program constructs can be implemented with Turing machines.*

2.6 k-tape Turing machine

The k-tape Turing machine, unlike the Turing machine we have learned, has k tapes, where each tape is equipped with a separate read/write head.



Formally, the k-tape Turing machine can be defined as follows.

Definition 2.10. $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a *k-tape Turing machine* if

(i) Q, Σ, Γ, q_0 and F are respectively the state set of the control unit, the input alphabet, the tape alphabet, the start state and the set of final states, as in the Turing machine; furthermore

(ii) only the *transition function* given by partial function $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, -, \rightarrow\}^k$ is broader than that seen in the Turing machine, since now we have to work on multiple tapes simultaneously.

The k-tape Turing machine is clearly an extension of the (1-tape) Turing machine.

We omit the precise definition of the concepts given for the Turing machine, since these can be given in the same way as for the Turing machine. Here we only draw attention to some differences:

- A *configuration* of the k-tape Turing machine M is $(q, \alpha_1 a_1, \beta_1, \ldots, \alpha_k a_k, \beta_k)$. The configuration transition ($\vdash$) and reachability ($\models$) can be defined similarly to the Turing machine.
- The first tape is the input tape, the kth tape is the output tape, and the other tapes are work tapes. In the two-tape case, or for Turing machines performing only recognition, we do not distinguish a separate output tape.
- L_M and $f_M : \Sigma^* \to \Sigma^*$ can be defined without much difficulty.

According to the Church–Turing thesis, any task that can be performed by a k-tape Turing machine can be performed by a Turing machine. The question is only how much increase in operation count this entails. Before answering this question, we need to define the time and space requirements of the k-tape Turing machine.

Definition 2.11. $T_M(x)$ is the number of configuration transitions needed to decide $x \in \Sigma^*$, or to compute the value $f(x) = y$, for the k-tape Turing machine M performing the task, if M halts on input x; otherwise $T_M(x) = \infty$.

The above definition also gives the operation count[4] of the Turing machine, since the Turing machine is a $k = 1$ tape Turing machine.

Since even for inputs of length n, the number of steps can vary greatly, we perform the following information compression.

Definition 2.12. The *time requirement* of the k-tape Turing machine M on words of length n is

$$T_M(n) = \max_{|x|=n} T_M(x).$$

So we measure the operation count by the worst case, i.e., with the number of steps required to process the word of length n that requires the most steps, which can also be infinite if there is a word of length n on which the Turing machine M does not halt.

Similarly to the time requirement, the space requirement can be defined.

Definition 2.13. The space requirement $S_M(x)$ of the Turing machine M is the number of tape cells used during the processing of $x \in \Sigma^*$. $S_M(x) = \infty$ is possible. Often we only consider the number of cells used on the work tapes, if the input tape is read-only and the output tape is write-only.

Definition 2.14. The *space requirement* of M on words of length n is $S_M(n) = \max_{|x|=n} S_M(x)$.

The following theorem states that switching from a multi-tape Turing machine to a (single-tape) Turing machine comes with at most a quadratic efficiency loss.

Theorem 2.1. *For every k-tape Turing machine M, there exists a Turing machine M' such that $L_M = L_{M'}$, $f_M = f_{M'}$, $T_{M'}(n) = O\left(T_M^2(n)\right)$ and $S'_M(n) \leq \max\{S_M(n), n+1\} + 1$.*

Proof. Let the tapes of M be the tracks of M'.

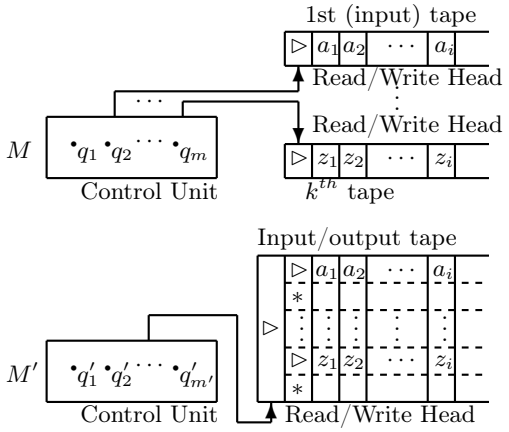


Figure 2.3: Simulation of a k-tape Turing machine with a single-tape one

Let the tape alphabet of M' be $\Gamma' = \Gamma \cup (\Gamma \times \{\sqcup, *\})^k$ and the start configuration

$$(q'_0, \triangleright, (\triangleright, *, \ldots, \triangleright, *)\,(x_1, \sqcup, \ldots, \sqcup, \sqcup) \ldots (x_n, \sqcup, \ldots, \sqcup, \sqcup)).$$

The simulation of the steps of M, i.e., the specification of δ', can be broken down into the following steps.

[4] In the following, we consider time requirement and operation count as synonyms.

1. M' moves past the last $*$ symbol, meanwhile storing the $(\gamma_i)_{i=1}^k$ symbols all elements of Γ under the "heads" in Q' and moves the "heads" to the right. It is advisable to move the heads, more precisely the '*' symbols representing the head positions, to the right so that in the backward movement described in step 3, we only need to move the heads to the left, because otherwise moving the '*' symbol to the right on a track would require unnecessary movement of M''s head to the right in step 3.

2. After step 1, we have stored in the appropriate Q' state the current state of M's control unit and the read symbols under the $*$ symbols. So we have the information needed to compute δ. We store the result of the computation, i.e., the function value of δ, in a Q' state — which contains the new control unit state, the new symbols to be written on the odd-numbered tracks, and the movement directions of the $*$ symbols on the even-numbered tracks.

3. While moving left, M' performs the changes stored in Q' on the tracks.

Let's turn to estimating the operation count of M'. Simulating one step of M requires the most steps when the number of cells written on one of M's tapes becomes as large as possible, which happens when the head on one of M's tapes moves right in every step. Then the number of right moves on the tape in question certainly cannot exceed $T_M(n)$. In step 1, M' takes two more steps because it starts from its own start symbol, and it can move the '*' symbol beyond the last symbol. Step 2 can be done with one configuration transition. Step 3, moving from the end of the tape to the beginning, requires as many steps as moving from the beginning to the end in step 1. Summing up, one step by M can be simulated by M' in at most $2T_M(n)+5$ steps.

Since M takes at most $T_M(n)$ steps in the worst case, the simulation of M, requires at most $2T_M^2(n) + 5T_M(n) = O\left(T_M^2(n)\right)$ steps.

M' writes to exactly $S_M(n)$ different cells if M uses at most one non-input tape. Let's also consider that by default, for multi-tape Turing machines, we do not count the number of cells needed to store the input in the number of cells used. However, for M' we only have one tape, so we must take into account the number of cells needed to store the input. Therefore, $S'_M(n) = \max\{S_M(n), n+1\} + 1$ follows for multi-tape Turing machine M that uses only one non-input tape, where the plus 1s are needed due to the start symbols, and in general $S'_M(n) \leq \max\{S_M(n), n+1\} + 1$. □

2.7 Recursiveness and recursive enumerability

In this subsection, we define the classes of languages that can be recognized or decided by Turing machines.

Definition 2.15. $L \subseteq \Sigma^*$ is *recursively enumerable* if there exists a Turing machine M that recognizes the language L, i.e., $L = L_M$.

Notation 2.1. $\mathcal{L}_{RE}$ is the set of recursively enumerable languages.

Recursive enumerability is a weak requirement, since the rejection of a word could be the result of the recognizing Turing machine running forever. A stronger requirement is if we demand that the rejection of any word occurs in a finite number of steps.

Definition 2.16. The language $L \subseteq \Sigma^*$ is *recursive* if there exists a Turing machine M such that $L = L_M$ and M halts on every word $x \in \Sigma^*$. Then we say the language L is (algorithmically) *decidable*.

Notation 2.2. $\mathcal{L}_R$ is the set of recursive languages.

The problems that can be captured by recursive languages are the algorithmically solvable problems. In contrast, for recursively enumerable but not recursive languages, only a so-called "semi-algorithm" can be given. For recursive languages, we get a yes or no answer to our question[5] in finite time, while for only recursively enumerable languages, we cannot know whether it is still worth waiting for the yes or no answer, because the yes answer might arrive after arbitrarily many steps due to possible infinite operation. For recursive languages, it is also not certain that we can wait for the answer, but at least, if we are sufficiently patient, we will always get an answer to our question. In practice, the time available to us is usually strongly limited; we will deal with such questions in the complexity theory chapter. Nevertheless, there is a sharp, qualitative dividing line between recursive and only recursively enumerable languages.

Let's turn to the corresponding concepts introduced for decision problems for function value computation tasks.

Definition 2.17. The partial function $f : \Sigma^* \to \Sigma^*$ is *partial recursive* if there exists a Turing machine M such that $f = f_M$.

The partial recursive function is akin to recursively enumerable languages in the sense that the associated Turing machine does not halt on every input. In other words, on certain words we cannot know whether it is still worth waiting for the result of the computation process, since precisely on those words the partial recursive function will be undefined, on which the Turing machine computing it does not halt.

More pleasant than partial recursive functions are the recursive functions, since the Turing machines computing them halt on any input and provide a result.

Definition 2.18. The function $f : \Sigma^* \to \Sigma^*$ is *recursive* if there is an always halting Turing machine M such that $f = f_M$.

2.8 Universal Turing machine

Based on the above, the Turing machine seems to be a computer with a burned-in program, and therefore it might appear to be a more modest tool than digital computers in the sense that we do not yet see its programmability.

A possible encoding of a Turing machine: given the k-tape Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ and let $m = |Q|$, $n = |\Gamma|$, furthermore $t = |F|$. Q can be identified with $Q^* = \{1, \ldots, m\}$, Γ with $\Gamma^* = \{m+1, \ldots, m+n\}$ (where $m+1$ codes $\triangleright$ and $m+2$ codes $\sqcup$), $D = \{\leftarrow, -, \rightarrow\}$ with $D^* = \{m+n+1, m+n+2, m+n+3\}$, q_0 with 1 and F with $F^* = \{m-t+1, \ldots, m\}$. The δ function can be given pointwise as a sequence $q, a_1, \ldots, a_k, p, b_1, \ldots, b_k, d_1, \ldots, d_k$, where $q, a_1, \ldots, a_k, p, b_1, \ldots, b_k, d_1, \ldots, d_k$ are appropriately encoded. The numbers $1, 2, \ldots, m+n+3$ can be encoded as binary sequences of length $\lceil \log_2(m+n+3) \rceil$. So M can be given as the sequence $\#k$, $\#m$, $\#n$, $\#t$ and $\#\delta$, where $\#$ is the symbol for binary encoding. At the very beginning the word length of the Turing machine can be encoded unary followed by a 0 separating the actual encoding.

A universal Turing machine U is capable of simulating any Turing machine. For this, only the code $\omega = \#M$ of the given Turing machine M and the code $\xi = \#x$ of the given input word x need to be given on the input tape of U. The value 0 can be used to separate ω and ξ, but for clarity we write $\omega; \xi$. Without loss of generality, it can be assumed that $\Sigma = \{0, 1\}$.

[5] Our question is whether $x \in L$.

Then $x = \xi$. So we expect $U(\omega, x) = \#M(x)$ (sloppily $U(M, x) = M(x)$) to hold for every M and every $x \in \Sigma^*$.

Theorem 2.2. *A 3-tape Turing machine can be constructed such that $U(M, x) = M(x)$ for every M and $x \in \Sigma^*$.*

Proof. We can assume without loss of generality that M is single-tape. The first tape of U contains $\omega; x$, the second plays the role of M's tape, and the third contains the state of M.

(1.) Check for the existence of a machine with code ω.

(2.) From ω, it reads and then writes q_0 on the third tape, x on the second, after which the read heads move to the beginning of the tapes.

(3.) Step by step, simulate M. For this, read the symbol a under the second tape's read head and the state q on the third tape. Then on the first tape, find the transition (p, b, d) corresponding to (q, a).

Write b on the second tape and move the second tape's head in direction d. Write state p on the third tape.

(4.) U should halt exactly when (q, a) is not found on the first tape. If halting occurs in a final state of M, U should also enter a final state. The second tape is identical to the tape content of M at termination. □

Remark 2.1. The universal Turing machine corresponds to an interpreter. So the Turing machine is programmable.

2.9 Algorithmically undecidable problem

Let us denote the Turing machine corresponding to code ω by M_ω.

Definition 2.19. $L_d = \{\omega \in \{0,1\}^* \mid \exists M_\omega \text{ and } \omega \notin L_{M_\omega}\}$ is the *diagonal language*.

Theorem 2.3. $L_d \notin \mathcal{L}_{RE}$.

Proof. Assume that $L_d \in \mathcal{L}_{RE}$, i.e., there exists a Turing machine M that recognizes the language L_d ($L_d = L_M$). Let $\omega \in \{0,1\}^*$ denote the code of this M. Since $M = M_\omega$, we have $L_M = L_{M_\omega}$. Two cases are possible:

(i) $\omega \in L_d$. Then by the definition of L_d, $\omega \notin L_{M_\omega}$. But since $L_d = L_M = L_{M_\omega}$, we arrive at a contradiction.

(ii) $\omega \notin L_d$. Then $\omega \in L_{M_\omega}$, and $L_d = L_M = L_{M_\omega}$ again leads to a contradiction.

□

For the diagonal language, not even a semi-algorithm can be given, which is rather negative. For the following two languages, the situation is somewhat better, since they can be recognized by a semi-algorithm, but not by an algorithm, so they are algorithmically undecidable.

Definition 2.20. $L_u = \{\omega; x \in \{0,1\}^* \mid \exists M_\omega \text{ and } x \in L_{M_\omega}\}$ is the *universal language*.

Theorem 2.4. $L_u \in \mathcal{L}_{RE} \setminus \mathcal{L}_R$.

Proof. Since L_u is the language recognized by the universal Turing machine, $L_u \in \mathcal{L}_{RE}$.

Assume that L_u is recursive. Then there is a Turing machine M that always halts such that $L_u = L_M$. Define a Turing machine M' whose input is a potential encoding ω. If ω is not an encoding of a Turing machine, i.e., there is no corresponding M_ω Turing machine, then M' halts in a rejecting state. If ω is a valid encoding, i.e., M_ω exists, then the computation continues with M on input $\omega;\omega$; and M' accepts ω exactly if M rejects $\omega;\omega$. So

$$\omega \in L_{M'} \Leftrightarrow \exists M_\omega \text{ and } \omega;\omega \notin L_M = L_u \Leftrightarrow \exists M_\omega \text{ and } \omega \notin L_{M_\omega}.$$

M' always halts, since the question of the existence of M_ω is algorithmically decidable and M always halts. So

$$\omega \in L_{M'} \iff \exists M_\omega \text{ and } \omega \notin L_{M_\omega},$$

i.e., $L_{M'} = L_d$. Since $L_d \notin \mathcal{L}_{RE}$, we arrive at a contradiction, because L_d cannot be recognized by a Turing machine. □

The halting problem seeks the answer to the question of whether a Turing machine halts on a given input.

Definition 2.21. $L_h = \{\omega;x \in \{0,1\}^* \mid \exists M_\omega \text{ and } T_{M_\omega}(x) < \infty\}$.

Theorem 2.5. $L_h \in \mathcal{L}_{RE} \setminus \mathcal{L}_R$.

Proof. Modify the universal Turing machine U so that all its states are accepting states, from which it follows that $L_h \in \mathcal{L}_{RE}$.

Assume that L_h is recursive. Then there is a Turing machine M that always halts such that $L_h = L_M$. Let M' be the following branch: `if` $M(\omega;x)$ `then` $U(\omega;x)$ `else` $\downarrow$.

Note that $L_{M'} = L_U$, which is impossible, since M' halts on every input, while $L_U \notin \mathcal{L}_R$ does not. □

2.10 RAM machine

A construction closer to real computers than the Turing machine is the RAM machine, which consists of the following components:

1. potentially infinite input tape and output tape;
2. directly addressable potentially infinite internal memory, whose zero-indexed location is the accumulator, and
3. program memory and program counter.

We store integers on the tapes and in the memory. The RAM machine has the following instruction set:

Arithmetic	Data Movement	Control
ADD *op*	LOAD *op*	JUMP *label*
SUB *op*	STORE *op*	JGTZ *label*
	READ *op*	JZERO *label*
	WRITE *op*	HALT

The RAM machine is programmable with a simple assembly language, we just need to specify the possible values of *op*:

- $= i$ the integer value i,

- i the value of the ith memory cell and
- $*i$ the value of the memory cell addressed by the ith memory cell.

The *label* is the new value of the program counter.

It is natural to measure the operation count of the RAM machine by the number of instructions executed.

Definition 2.22. The *uniform cost* is the number of instructions executed.

Since the RAM machine can work with arbitrarily large integers, for a more realistic measurement of the operation count, we must take into account the number of bits needed for computation, or the length of numbers.

Definition 2.23. The *logarithmic cost* sums the instructions weighted by the data lengths involved in the computation.

We state without proof the theorem capturing the relationship between the operation count of the Turing machine and the RAM machine.

Theorem 2.6. *A Turing machine M can be simulated by a RAM machine with $O(T_M(n)\log_2 T_M(n))$ logarithmic and $O(T_M(n))$ uniform cost. A RAM machine with $t(n)$ logarithmic cost can be simulated by a Turing machine M in time $T_M(n) = O(t^2(n))$.*

2.11 Exercises

Exercise 2.1. Construct a Turing machine that decides the language

$L = \{\alpha = a_1 \dots a_n \in \{0,1,2,3\}^* \mid 2 \leq n \text{ and there exists } i \in \{1,\dots,n-1\} \text{ such that } a_i = a_n\}$

Exercise 2.2. Construct a Turing machine that accepts the language $L = \{\alpha \in \{0,1\}^* \mid \alpha = \alpha^{-1}\}$! ($L$ is the language of palindromes). Determine the maximum number of steps of the machine as a function of word length!

Exercise 2.3. Construct a Turing machine that decides the divisibility by three of the number encoded by the word $x \in \Sigma^* = \{0,1,2,3,4,5,6,7,8,9\}^*$ placed on the input tape in decimal representation!

Exercise 2.4. Construct a Turing machine that increments by one the value of a number encoded in binary!

Exercise 2.5. Prove the following statements!

1. If in the definition of the Turing machine we do not allow the head to stay in place, it does not change the power of the computational model (i.e., $D = \{\rightarrow, \leftarrow\}$).
2. If we allow potentially infinite tape(s) in both directions, that also does not change the power of the computational model.

Exercise 2.6. Construct a 2-tape Turing machine that accepts the language $L = \{\alpha \in \{0,1\}^* \mid \alpha = \alpha^{-1}\}$! Determine the maximum number of steps of the machine as a function of word length!

Exercise 2.7. Construct a two-tape Turing machine that decides the language

$$L = \{\alpha \in \{0,1,\#\}^* \mid \alpha = \beta\#\gamma,\ \beta,\gamma \in \{0,1\}^* \text{ and } \beta < \gamma\}$$

where $\beta < \gamma$ denotes that the number represented by β in binary is less than the number represented by γ in binary!

Exercise 2.8. Construct a two-tape Turing machine that gives the number of 0 symbols found in the word $x \in \Sigma^* = \{0,1,2,3\}^*$ placed on the input tape (in some representation) on the second tape!

Exercise 2.9. Construct a multi-tape Turing machine that gives the product of the numbers $x \in \Sigma^* = \{1\}^*$ and $y \in \Sigma^* = \{1\}^*$ given in unary encoding on the output tape in unary encoding! The input tape has x and y separated by a blank.

Exercise 2.10. Construct a multi-tape Turing machine that gives the sum of the numbers $x \in \Sigma^* = \{0,1\}^*$ and $y \in \Sigma^* = \{0,1\}^*$ given in binary encoding on the output tape in binary encoding! The input tape has x and y separated by a blank.

Exercise 2.11. Write a Python class that simulates a Turing machine! The class should be capable of solving decision and function value computation tasks. If needed, print the configurations step by step. Test our class with the Turing machines given in the above exercises!

Chapter 3

Boolean Logic

Boolean logic, also known as propositional calculus, is suitable for formulating very simple logical statements. It is important for us in order to discuss a central complexity theory problem and to introduce first-order predicate calculus. We have already used Boolean logic in our inferences; the novelty may only be the precise specification of its language and meaning.

The symbols of Boolean logic are

- $X = \{x_1, x_2, \ldots\}$ the countably infinite set of *Boolean variables*, which can then take the logical true (denoted $\uparrow$) or logical false (denoted $\downarrow$) values;[1] and
- the Boolean operations $\neg$, $\vee$ and $\wedge$.

In the following, we specify how to build syntactically correct so-called Boolean formulas using our introduced symbols.

Definition 3.1. Boolean formulas can be formed as follows:

(i) A Boolean variable x_i by itself is a Boolean formula,

(ii) $\varphi = \neg\varphi_1$ is a Boolean formula if φ_1 is a Boolean formula,

(iii) $\varphi = (\varphi_1 \vee \varphi_2)$ is a Boolean formula if φ_1 and φ_2 are already Boolean formulas, and

(iv) $\varphi = (\varphi_1 \wedge \varphi_2)$ is a Boolean formula if φ_1 and φ_2 are already Boolean formulas.

Boolean variables and their negations play a distinguished role, which are called *literals*. So x_i and $\neg x_i$ are literals.

Statements about Boolean formulas, or introduced concepts, can be proven or introduced by structural induction on the syntax. For this, first consider the formal specification of the set of Boolean variables occurring in Boolean formulas. Let $X(\varphi)$ denote the set of variables occurring in formula φ. Formally:

(i) If $\varphi = x_i$, then $X(\varphi) = \{x_i\}$.

(ii) If $\varphi = \neg\varphi_1$, then $X(\varphi) = X(\varphi_1)$.

(iii) If $\varphi = (\varphi_1 \vee \varphi_2)$, then $X(\varphi) = X(\varphi_1) \cup X(\varphi_2)$.

(iv) If $\varphi = (\varphi_1 \wedge \varphi_2)$, then $X(\varphi) = X(\varphi_1) \cup X(\varphi_2)$.

[1] The mention of the true and false values that Boolean variables can take facilitates the understanding of the described concepts; it does not play a role in the specification of the syntax, only in the specification of the semantics.

In order to assign meaning to a Boolean formula, we first need to assign meaning or binary truth values to the variables occurring in it. This leads us to the following concept.

Definition 3.2. An *interpretation* is a function $I : X' \to \{\uparrow, \downarrow\}$, where $X' \subseteq X$. We say that the interpretation $I : X' \to \{\uparrow, \downarrow\}$ is an *applicable* interpretation for φ if $X(\varphi) \subseteq X'$.

After introducing the concept of interpretation, we can assign meaning to our Boolean formulas. The semantics of the language of Boolean logic is given by induction on its syntax.

Definition 3.3. An *interpretation* $I : X' \to \{\uparrow, \downarrow\}$ applicable to formula φ *satisfies* the formula φ, denoted $I \models \varphi$, in the following cases:

(i) If $\varphi = x_i$, then $I \models \varphi$ if $I(x_i) = \uparrow$;

(ii) If $\varphi = \neg\varphi_1$, then $I \models \varphi$ if $I \not\models \varphi_1$;

(iii) If $\varphi = (\varphi_1 \vee \varphi_2)$, then $I \models \varphi$ if $I \models \varphi_1$ or $I \models \varphi_2$;

(iv) If $\varphi = (\varphi_1 \wedge \varphi_2)$, then $I \models \varphi$ if $I \models \varphi_1$ and $I \models \varphi_2$.

Let's look at an example of the interpretation of a Boolean formula.

Example 3.1. $\varphi = ((x_1 \wedge x_2) \vee (x_1 \wedge x_3))$ and $I(x_1) = \uparrow$, $I(x_2) = \uparrow$ and $I(x_3) = \downarrow$, then $I \models \varphi$.

Implication and equivalence are introduced as follows:

Notation 3.1. $\varphi_1 \Rightarrow \varphi_2$ is an abbreviation for $\neg\varphi_1 \vee \varphi_2$ (implication). $\varphi_1 \Leftrightarrow \varphi_2$ is an abbreviation for $\varphi_1 \Rightarrow \varphi_2$ and $\varphi_2 \Rightarrow \varphi_1$ (equivalence).

Two Boolean formulas are identical if they are obtained in the same way according to the syntax of Boolean formulas, or in other words, the two formulas read left-to-right character by character are the same. Weaker than that is the concept of logical equivalence.

Definition 3.4. φ_1 and φ_2 are *equivalent* ($\varphi_1 \equiv \varphi_2$) if for every interpretation I applicable to both, $I \models (\varphi_1 \Leftrightarrow \varphi_2)$.

Based on our definitions, the following logical equivalences are easily provable.

Proposition 3.1. *Given formulas φ_1, φ_2 and φ_3.*

(i) $(\varphi_1 \vee \varphi_2) \equiv (\varphi_2 \vee \varphi_1)$ *and* $(\varphi_1 \wedge \varphi_2) \equiv (\varphi_2 \wedge \varphi_1)$.

(ii) $((\varphi_1 \vee \varphi_2) \vee \varphi_3) \equiv (\varphi_1 \vee (\varphi_2 \vee \varphi_3))$ *and* $((\varphi_1 \wedge \varphi_2) \wedge \varphi_3) \equiv (\varphi_1 \wedge (\varphi_2 \wedge \varphi_3))$.

(iii) $((\varphi_1 \vee \varphi_2) \wedge \varphi_3) \equiv ((\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3))$ *and* $((\varphi_1 \wedge \varphi_2) \vee \varphi_3) \equiv ((\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3))$.

(iv) $(\varphi_1 \vee \varphi_1) \equiv \varphi_1$ *and* $(\varphi_1 \wedge \varphi_1) \equiv \varphi_1$.

(v) $\neg(\varphi_1 \vee \varphi_2) \equiv (\neg\varphi_1 \wedge \neg\varphi_2)$ *and* $\neg(\varphi_1 \wedge \varphi_2) \equiv (\neg\varphi_1 \vee \neg\varphi_2)$.

(vi) $(\neg\neg\varphi_1) \equiv \varphi_1$.

We will employ the following notations.

Notation 3.2. $\wedge_{i=1}^{n}\varphi_i := \varphi_1 \wedge \ldots \wedge \varphi_n$ and $\vee_{i=1}^{n}\varphi_i := \varphi_1 \vee \ldots \vee \varphi_n$.

3.1 Conjunctive and disjunctive normal forms

The following two types of Boolean formulas play a distinguished role.

Definition 3.5. A Boolean formula φ is given in *conjunctive normal form* (CNF) if

$$\varphi = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} L_{i,j} = \wedge_{i=1}^{n} C_i,$$

where C_i are clauses and L_{ij} are literals.

Definition 3.6. A Boolean formula φ is given in *disjunctive normal form* (DNF) if

$$\varphi = \vee_{i=1}^{n} \wedge_{j=1}^{m_i} L_{i,j} = \vee_{i=1}^{n} D_i,$$

where L_{ij} are literals.

Conjunctive and disjunctive normal forms are useful not only because of their simpler structure, but also because for any Boolean formula there exists a logically equivalent Boolean formula of such special form.

Theorem 3.1. *For every Boolean formula φ, there exist equivalent Boolean formulas in CNF and DNF.*

Proof. The proof is by induction on the syntax of Boolean formulas.

(i) If $\varphi = x_i$ is a Boolean formula that is merely a variable, then the formula is already given in CNF and also in DNF.

(ii) If the formula is obtained as the negation of another formula, then apply De Morgan's laws. Let $\varphi = \neg\varphi_1$ and $\varphi_1 = \vee_{i=1}^{n} \wedge_{j=1}^{m_i} L_{i,j}$ given in DNF. Then

$$\neg\varphi_1 = \neg(\vee_{i=1}^{n} \wedge_{j=1}^{m_i} L_{i,j}) \equiv \wedge_{i=1}^{n} (\neg \wedge_{j=1}^{m_i} L_{i,j}) \equiv \wedge_{i=1}^{n} \vee_{j=1}^{m_i} \neg L_{i,j}.$$

Similarly, taking the CNF of φ_1, we get that φ can be written in DNF.

(iii) If a formula is the disjunction of two other formulas, then the statement is obtained immediately or by applying the distributive laws. Let $\varphi = (\varphi_1 \vee \varphi_2)$. Then the DNF is trivial, since φ_1 and φ_2 are in DNF. To write φ in CNF, take the CNFs of φ_1 and φ_2. Then

$$\varphi_1 \vee \varphi_2 \equiv \left(\wedge_{i=1}^{m} C_i^{(1)}\right) \vee \left(\wedge_{j=1}^{n} C_j^{(2)}\right) \equiv \left(\wedge_{i=1}^{m} \left(C_i^{(1)} \vee \left(\wedge_{j=1}^{n} C_j^{(2)}\right)\right)\right) \equiv \wedge_{i=1,j=1}^{m,n} \left(C_i^{(1)} \vee C_j^{(2)}\right).$$

(iv) The case $\varphi = (\varphi_1 \wedge \varphi_2)$ is proven similarly to (iii): the CNF is obtained as the conjunction of the two CNFs, while for the DNF, the distributive laws must be applied similarly to the above. □

Formulas that are satisfied by some, or all, interpretations play an important role.

Definition 3.7. A Boolean formula φ is *satisfiable* if there exists an interpretation I applicable to φ such that $I \models \varphi$.

Definition 3.8. A Boolean formula φ is *valid* if for every interpretation I applicable to φ, $I \models \varphi$ holds. Then we can write $\models \varphi$.

Example 3.2. Is the formula $((x_1 \vee \neg x_2) \wedge \neg x_1)$ satisfiable? Is it valid?

A Boolean formula can be written as a word over the symbol set $\Sigma = \{x, 0, 1, (,), \neg, \vee, \wedge\}$. For example, the variable x_5 should be written as $x101$. The length of a Boolean formula $\varphi \in \{x, 0, 1, (,), \neg, \vee, \wedge\}^*$ is the length of its representation.

A famous, central problem in complexity theory is the question of the satisfiability of Boolean formulas.

Definition 3.9. The *SAT* (SATISFIABILITY) problem is to decide whether a Boolean formula given in CNF is satisfiable.

The satisfiability of a Boolean formula can be decided in at least exponential time in the number of variables by the brute force method.

3.2 Resolution

Using resolution, we can directly prove the unsatisfiability of a Boolean formula in CNF form. During the procedure, the set of clauses is continuously expanded until a contradiction is reached. If we find a complementary literal pair in two different clauses, i.e., x_i in one and $\neg x_i$ in the other, then the disjunction of the remaining parts of the clauses gives us a new clause.

Example 3.3. Prove that $\varphi = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_4) \wedge x_1 \wedge x_4$ is unsatisfiable.

In φ, the first and second clause pair can be resolved since the first contains x_2 and the second contains $\neg x_2$, resulting in the sixth clause $\neg x_1 \vee x_3$. The newly obtained clause and the third clause of φ can similarly be resolved using the complementary literal pair x_3 and $\neg x_3$, thus obtaining the seventh clause $\neg x_1 \vee \neg x_4$. From this and x_1, the eighth clause $\neg x_4$ follows, which contradicts the fifth clause x_4. The obtained contradiction proves that φ is unsatisfiable.

We prove that the above method is correct. First, we prove two lemmas.

Lemma 3.1. *If* $\Psi_1 \implies \Psi_2$ *is valid, then* $\Psi_1 \iff \Psi_1 \wedge \Psi_2$ *is also valid.*

Proof. Due to the validity of $\Psi_1 \implies \Psi_2$, only the following three cases are possible:

$I \models \Psi_1$	$I \models \Psi_2$	$I \models (\Psi_1 \implies \Psi_2)$	$I \models (\Psi_1 \iff \Psi_1 \wedge \Psi_2)$
↑	↑	↑	↑
↓	↑	↑	↑
↓	↓	↑	↑

□

Lemma 3.2. *The implication* $(x_i \vee \varphi_1) \wedge (\neg x_i \vee \varphi_2) \implies (\varphi_1 \vee \varphi_2)$ *is valid.*

Proof. The precondition requires that either $I \models x_i$ or $I \models \neg x_i$ and since both formulas of the conjunction must be satisfied $I \models (\varphi_1 \vee \varphi_2)$ follows. □

Now we can turn to our theorem, which proves that using two complementary literal pairs, the clause obtained as the disjunction of the remaining parts of the two clauses, when added to a Boolean formula, yields a Boolean formula that is logically equivalent.

Theorem 3.2. $(x_i \vee \varphi_1) \wedge (\neg x_i \vee \varphi_2) \iff (x_i \vee \varphi_1) \wedge (\neg x_i \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_2)$.

Proof. Let $\Psi_1 = (x_i \vee \varphi_1) \wedge (\neg x_i \vee \varphi_2)$ and $\Psi_2 = (\varphi_1 \vee \varphi_2)$, then applying Lemmas 3.1 and 3.2 yields the theorem. □

3.3 Boolean functions

If an interpretation satisfies a Boolean formula, then this can be thought of as the formula being true given the truth values of the variables. In this way, a logical function can be defined by means of a Boolean formula.

Definition 3.10. An n-variable *Boolean function* is a function $f : \{\uparrow,\downarrow\}^n \to \{\uparrow,\downarrow\}$.

Example 3.4. $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are binary Boolean functions. $\neg$ is a unary Boolean function.

As we have already hinted, any Boolean formula φ can be considered as an $n = |X(\varphi)|$ variable Boolean function f_φ. For simplicity, we can assume that $X(\varphi) = \{x_1, \ldots, x_n\}$. Then for an interpretation $I : \{x_1, \ldots, x_n\} \to \{\uparrow,\downarrow\}$, let $f_\varphi(I(x_1), \ldots, I(x_n)) = \uparrow$ exactly when $I \models \varphi$.

We will show that conversely, for any Boolean function, an equivalent Boolean formula can be constructed.

Proposition 3.2. *Any Boolean function $f : \{\uparrow,\downarrow\}^n \to \{\uparrow,\downarrow\}$ can be expressed by a φ_f Boolean formula using the variables $x_1, \ldots, x_n$.*

Proof. Let $F = \{(t_1, \ldots, t_n) \in \{\uparrow,\downarrow\}^n \mid f(t_1, \ldots, t_n) = \uparrow\}$. Assume that $F \neq \emptyset$. For each $\mathbf{t} = (t_1, \ldots, t_n) \in F$, assign the

$$D_{\mathbf{t}} = (\wedge_{t_i=\uparrow} x_i) \wedge (\wedge_{t_i=\downarrow} \neg x_i)$$

formula. Finally, let $\varphi_f = \vee_{\mathbf{t}\in F} D_{\mathbf{t}}$ (DNF).

We show that for an interpretation $I : \{x_1, \ldots, x_n\} \to \{\uparrow,\downarrow\}$, $f(I(x_1), \ldots, I(x_n)) = \uparrow$ holds exactly if $I \models \varphi_f$. If $f(I(x_1), \ldots, I(x_n)) = \uparrow$, then $(I(x_1), \ldots, I(x_n)) \in F$ and $D_{(I(x_1),\ldots,I(x_n))} = (\wedge_{I(x_i)=\uparrow} x_i) \wedge (\wedge_{I(x_i)=\downarrow} \neg x_i)$ is a term of φ_f. Therefore $I \models D_{(I(x_1),\ldots,I(x_n))} = (\wedge_{I(x_i)=\uparrow} x_i) \wedge (\wedge_{I(x_i)=\downarrow} \neg x_i)$, from which it follows that $I \models \varphi_f$.

Conversely, if $I \models \varphi_f$, then there is a term of φ_f for which $I \models D_{(I(x_1),\ldots,I(x_n))} = (\wedge_{I(x_i)=\uparrow} x_i) \wedge (\wedge_{I(x_i)=\downarrow} \neg x_i)$. Then $(I(x_1), \ldots, I(x_n)) \in F$, so $f(I(x_1), \ldots, I(x_n)) = \uparrow$.

Finally, if $F = \emptyset$, then let $\varphi_f = (x_1 \wedge \neg x_1)$. □

Since storing Boolean functions is space-consuming (the space requirement is exponential in the number of variables), we might manage better with the associated Boolean formula. However, it can be proven that a Boolean formula determined for a given Boolean function is "generally long".

3.4 Boolean circuits

Boolean formulas can be evaluated more easily with the help of Boolean circuits. As an example, we give the Boolean circuit corresponding to the Boolean formula $\varphi = (((x_1 \wedge x_2) \vee \neg(x_1 \wedge x_3)) \vee ((x_1 \vee x_2) \vee x_4))$ in Figure 3.1.

Let's turn to the formal definition of a Boolean circuit.

Definition 3.11. A directed graph $C = (V, E)$ is a Boolean circuit if

- $V = \{1, \ldots, n\}$ is the set of *gates*;
- C is directed acyclic (so it can be assumed that if $(i, j) \in E$ then $i < j$);
- $s : V \to \{\uparrow,\downarrow,\neg,\vee,\wedge, x_1, \ldots, x_n\}$ is the *type* of the gates, where

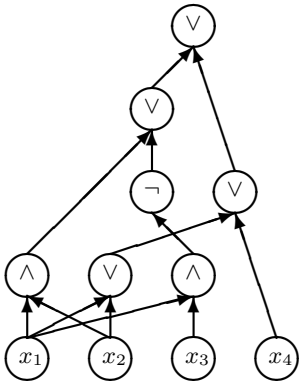


Figure 3.1: Boolean circuit.

  - indegree$(i) = 0$ if $s(i) \in \{\uparrow, \downarrow, x_1, \ldots, x_n\}$ (these are the *input* gates),
  - indegree$(i) = 1$ if $s(i) = \neg$ and
  - indegree$(i) = 2$ if $s(i) \in \{\vee, \wedge\}$;
- outdegree$(n) = 0$ (n is the *output* gate).

Let $X(C)$ denote the set of variables occurring in C.

Definition 3.12. An interpretation $I : X' \to \{\uparrow, \downarrow\}$ is *applicable* to C if $X(C) \subseteq X'$.

We give the rules for evaluating a Boolean circuit.

Definition 3.13. The truth value $T : V \to \{\uparrow, \downarrow\}$ of the gates in $C = (V, E)$ under an interpretation $I : X' \to \{\uparrow, \downarrow\}$ applicable to C is defined as follows:

- $T(i) = s(i)$ if $s(i) \in \{\uparrow, \downarrow\}$;
- $T(i) = I(s(i))$ if $s(i) \in X$;
- $T(i) = \neg T(j)$ if $s(i) = \neg$ and $(j, i) \in E$;
- $T(i) = T(j) \vee T(k)$ if $s(i) = \vee$, $(j, i) \in E$, $(k, i) \in E$ and $j \neq k$;
- $T(i) = T(j) \wedge T(k)$ if $s(i) = \wedge$, $(j, i) \in E$, $(k, i) \in E$ and $j \neq k$.

Definition 3.14. The truth value of the circuit C is $T(n)$.

Remark 3.1. For a Boolean formula φ, an equivalent Boolean circuit C_φ can be constructed, and conversely, for a Boolean circuit C, an equivalent Boolean formula φ_C can be assigned. It can be proven that $T(C_\varphi) = \uparrow$ exactly when $I \models \varphi$.

The following theorem points out that there are Boolean functions that can only be computed by Boolean circuits of large size.

Theorem 3.3. *For any integer $n \geq 2$, there exists an $f : \{\uparrow, \downarrow\}^n \to \{\uparrow, \downarrow\}$ that can only be computed by a Boolean circuit with more than $\frac{2^n}{2n}$ gates.*

Proof. Assume that $m = \frac{2^n}{2n}$ gates are sufficient to compute any n-variable Boolean function. Estimate from above the number of Boolean circuits with at most m gates. A gate can have at most $n + 5$ types, and the number of possible input edge combinations per gate is less

than m^2. So per gate, there are less than $(n+5)m^2$ possibilities. If we allow any possibility per gate (which also brings in non-Boolean circuits), then we get that there can be at most $((n+5)m^2)^m$ m-gate Boolean circuits.

Take into account that there are 2^{2^n} different Boolean functions $f : \{\uparrow,\downarrow\}^n \to \{\uparrow,\downarrow\}$. But then we will not have enough Boolean circuits, since

$$\left((n+5)\,m^2\right)^m = \left((n+5)\,\frac{2^{2n}}{4n^2}\right)^{\frac{2^n}{2n}} < 2^{2^n} \iff n+5 < 4n^2,$$

which holds for $n \geq 2$. So we have reached a contradiction. □

It is noteworthy that no family of Boolean functions has been found yet that requires essentially an exponential number of gates.

3.5 Exercises

Exercise 3.1. Prove that

$$((x_1 \vee x_2) \wedge (x_3 \vee x_4)) \equiv ((x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4))!$$

Exercise 3.2. Prove that

$$((x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)) \equiv ((x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3))!$$

Exercise 3.3. Prove that $(\neg \wedge_{i=1}^n \varphi_i) \equiv (\vee_{i=1}^n \neg\varphi_i)$!

Exercise 3.4. Prove that $((\wedge_{i=1}^n \varphi_i) \vee \varphi) \equiv (\wedge_{i=1}^n (\varphi_i \vee \varphi))$!

Exercise 3.5. Determine the CNF of the Boolean formula $\neg((x_1 \Rightarrow x_2) \wedge ((x_2 \Rightarrow x_3) \wedge \neg(x_4 \wedge x_3)) \Rightarrow (x_1 \Rightarrow \neg x_4))$!

Exercise 3.6. Is

$$\varphi = ((x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3))$$

satisfiable? Is $\neg\varphi$ valid?

Exercise 3.7. How many unary and binary Boolean functions are there? How many n-variable Boolean functions are there?

Exercise 3.8. Consider the Boolean function f given by the following table!

x_1	x_2	x_3	$f(x_1,x_2,x_3)$
↑	↑	↑	↑
↑	↑	↓	↓
↑	↓	↑	↓
↑	↓	↓	↓
↓	↑	↑	↓
↓	↑	↓	↑
↓	↓	↑	↓
↓	↓	↓	↑

Determine a Boolean formula φ in disjunctive normal form that expresses the Boolean function f!

Exercise 3.9. Consider the Boolean function f given by the following table!

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$
↑	↑	↑	↑	↓
↑	↑	↑	↓	↓
↑	↑	↓	↑	↑
↑	↑	↓	↓	↓
↑	↓	↑	↑	↓
↑	↓	↑	↓	↑
↑	↓	↓	↑	↓
↑	↓	↓	↓	↑
↓	↑	↑	↑	↓
↓	↑	↑	↓	↑
↓	↑	↓	↑	↑
↓	↑	↓	↓	↓
↓	↓	↑	↑	↑
↓	↓	↑	↓	↑
↓	↓	↓	↑	↑
↓	↓	↓	↓	↓

Determine a Boolean formula φ in disjunctive normal form that expresses the Boolean function f!

Exercise 3.10. Draw the Boolean circuits that evaluate the following Boolean formulas!

(i) $\varphi = (\neg(x_1 \wedge \neg x_3) \vee ((\neg x_2 \vee x_4) \wedge \neg(x_2 \wedge \neg x_3)))$

(ii) $\varphi = (((x_1 \wedge x_2) \wedge x_3) \vee \neg(x_2 \wedge \neg x_4) \vee x_2)$

(iii) $\varphi = ((((\neg x_1 \wedge x_2) \vee (x_3 \vee \neg x_4)) \wedge (x_2 \wedge \neg x_4)) \vee \uparrow)$

Chapter 4

Computational Complexity

In this chapter, we will bound the time and space requirements needed to solve a problem, i.e., we will classify problems into time and space complexity classes.

To define complexity classes, we use the Turing machine as the computational model. The processing time required by a Turing machine is measured by the number of configuration transitions needed to solve the problem. The worst-case number of steps required by a Turing machine for inputs of a given length is bounded by a strictly monotonically increasing non-negative sequence as follows.

Definition 4.1. A Turing machine M is *time-bounded* by $t(n)$ if $T_M(n) \leq t(n)$ for all $n \in \mathbb{N}$.

If among inputs of length n there are non-"redundant" ones, then at least one input of length n must be read. Therefore, it can be assumed that in the worst case, processing inputs of length n requires at least n steps. Hence, we can make the following natural assumption.

Assumption 4.1. The time bound t satisfies $t(n) \geq n$ for any $n \in \mathbb{N}$.

So far, we have defined the time bound of a single Turing machine. We obtain a time complexity class of languages by taking the class of languages recognizable by Turing machines that are time-bounded by t.

Definition 4.2. The complexity class corresponding to $t : \mathbb{N} \to \mathbb{N}$ is

$$\mathrm{TIME}(t(n)) = \{L \subseteq \Sigma^* \mid \exists M : L = L_M \text{ and } T_M(n) = O(t(n))\}.$$

From the definition of the TIME($t(n)$) complexity class, we see that it consists of the languages, which are on one hand, recognizable by some Turing machine M, and on the other hand, the recognition can be performed in time asymptotically at least t, i.e., T_M has t as an asymptotic upper bound. Due to the asymptotic nature of the definition, we are only interested in the existence of an index n_0 such that for inputs not shorter than that ($n \geq n_0$), the recognition can be performed in at least $ct(n)$ steps for some suitably chosen positive constant c. Our statements about complexity classes are therefore asymptotic in nature, and for brevity, in our proofs we will not always repeat that inequalities like $T_M(n) \leq ct(n)$ only need to hold from some suitable index onwards.

Due to time boundedness, any language in TIME($t(n)$) is algorithmically decidable.

Proposition 4.1. *If $L \in$ TIME($t(n)$), then $L \in \mathcal{L}_R$.*

Sometimes for convenience, we assume that a Turing machine M' deciding a language L requires exactly $ct(n)$ steps — then clearly $c \in \mathbb{N}$ — which for a language $L \in$ TIME($t(n)$)

can be achieved by slowing down the $t(n)$ time-bounded Turing machine M that decides it with unnecessary steps. The slowdown can also be achieved by "parallel combination" of the Turing machine M and a Turing machine N that computes the sequence $ct(n)$, in which M and N take their steps simultaneously, and if M finishes earlier, we still wait for N's steps, even though the result of the computation is determined by machine M. Thus, the following simple statement holds.

Proposition 4.2. *If $L \in TIME(t(n))$, then there exists a Turing machine M recognizing L for which there exists a positive integer c such that $T_M(n) = ct(n)$ holds from some suitable index onwards.*

In complexity theory, problems solvable in polynomial time play a distinguished role.

Definition 4.3. $\mathrm{P} = \cup_{k=1}^{\infty} \mathrm{TIME}(n^k)$.

The decision of languages in P can also be time-consuming, but experience suggests that if a higher-degree polynomial algorithm is found for a problem, then the degree can be "sufficiently" reduced. A polynomial algorithm is asymptotically more efficient than an exponential one, although it is conceivable that only from a very large index does the polynomial algorithm behave more favorably than an exponential one. Again, based on our experience, we can say that for most practical problems the index is sufficiently small, and therefore the polynomial algorithm runs faster. Of course, there are exceptional problems, but complexity classes based on the asymptotic behavior of algorithms provide a concise, elegant, mathematically manageable, relevant qualitative characterization of problems.

We have already encountered a language decidable in polynomial time.

Example 4.1. The language of palindromes is in P.

After introducing time complexity, we turn to space complexity.

Definition 4.4. An M Turing machine is *space-bounded* by $s(n)$ if $S_M(n) \leq s(n)$ for all $n \in \mathbb{N}$.

Similar to Assumption 4.1, though somewhat stronger, is the following assumption.

Assumption 4.2. $s(n) \geq \log_2 n$.

Assumption 4.2 holds if we need to store the input length during processing, for example, for addressing purposes.

A space complexity class is defined similarly to the time complexity class.

Definition 4.5. The space complexity class corresponding to $s : \mathbb{N} \to \mathbb{N}$ is

$$\mathrm{SPACE}(s(n)) = \{L \subseteq \Sigma^* \mid \exists M : L = L_M \text{ and } S_M(n) = O(s(n))\}.$$

In particular, $\mathrm{L} = \mathrm{SPACE}(\log_2 n)$ is the logarithmic space complexity class.

Similar to decision problems, time and space complexity classes can be defined for determining the value of a function, denoted by $\mathrm{FTIME}(t(n))$, $\mathrm{FSPACE}(s(n))$, and FP, respectively.

The following theorem establishes a connection between space and time complexity.

Theorem 4.1 (Space-Time Theorem)**.** *Let Assumption 4.2 hold. Then if $L \in SPACE(s(n))$, then there exists a constant $c \in \mathbb{N}$ greater than 1 such that $L \in TIME(c^{s(n)})$.*

Proof. If $L \in \mathrm{SPACE}(s(n))$, then by definition there exists a k-tape Turing machine M such that $L = L_M$, and by inserting unnecessary steps, it can also be achieved that $S_M(n) = c_1 s(n)$, where $c_1 \in \mathbb{N}$ and $c_1 \geq 1$. Note that due to the space boundedness of M, M does not halt if and only if M enters an infinite loop. We give an N Turing machine that is $O(c^{s(n)})$ time-bounded and for which $L_N = L_M$.

N contains two duplicates of M, M_1 and M_2, and two counter tapes from which it can be read which step the simulating M_1 and M_2 are at, respectively. The "code" of N:

```
l := 0; M1 to starting configuration;
while M1 does not terminate repeat
    j := 0; M2 to starting configuration;
    while j < l repeat
        if configurations of M1 and M2 are identical then
            N halts in rejecting state;
        j := j + 1; M2 takes one step;
    end while
    l := l + 1; M1 takes one step;
end while
```

If we did not exit the inner loop, then upon termination, N accepts and rejects according to M_1. l denotes the number of steps taken by M_1 and j the number of steps taken by M_2, whose values are found on the two counter tapes. N simulates M while monitoring for possible configuration repetitions, thereby filtering out the infinite loops of M. We show that N halts, i.e., it cannot proceed indefinitely without configuration repetition. We estimate the number of configurations of M from above as follows:

1. the number of control unit states is $|Q|$;
2. on the work tapes and the output tape, we use at most $S(n)$ cells, and in each cell we can write $|\Gamma|$ different symbols independently, giving $|\Gamma|^{S(n)}$ possibilities;
3. on the input tape, the read head can be at $n+1$ positions, while on the other tapes at most at $S(n)$ positions.

Summing up, the number of possible configurations cannot exceed $t' := |Q||\Gamma|^{S(n)}(n+1)(S(n))^{k-1}$. Thus, within t' steps, it will certainly be revealed whether M enters an infinite loop or not.

By Assumption 4.2 and $S_M(n) = c_1 s(n)$, we have $S(n) \geq s(n) \geq \log_2 n$.

$$\begin{aligned} t' &\leq |Q|(S(n))^{k-1}|\Gamma|^{S(n)}2n \leq 2|Q|2^{\log_2(S(n))^{k-1}}|\Gamma|^{S(n)}2^{S(n)} \\ &\leq 2|Q|2^{(k-1)S(n)}2^{S(n)}|\Gamma|^{S(n)} = 2|Q|\left(2^k|\Gamma|\right)^{S(n)} = c_0 c_2^{S(n)}. \end{aligned}$$

Let $t := c_0 c_2^{S(n)}$. So within t' steps, a step repetition must occur if M has not halted by then. Based on the double loop in the code of N,

$$T_N(n) = O(t^2) = O\left(\left(c_2^{S(n)}\right)^2\right) = O\left(\left(c_2^2\right)^{c_1 s(n)}\right).$$

Choosing $c := c_2^{2c_1}$ yields the theorem. □

The following analogous theorem can be proved.

Theorem 4.2. *Under Assumption 4.2 and if $f \in FSPACE(s(n))$, then there exists a $c \in \mathbb{N}$ with $c > 1$ such that $f \in FTIME(c^{s(n)})$.*

Besides the P time complexity class, the following complexity classes play an important role.

Definition 4.6. $\text{EXPTIME} = \cup_{k=1}^{\infty} \text{TIME}(2^{n^k})$.

Definition 4.7. $\text{PSPACE} = \cup_{k=1}^{\infty} \text{SPACE}(n^k)$.

The following theorem establishes a connection between the three complexity classes introduced so far.

Theorem 4.3. *$P \subseteq PSPACE \subseteq EXPTIME$.*

Proof. If $L \in \text{P}$, then there is a Turing machine M such that $L = L_M$ and M is cn^k time-bounded, and hence also cn^k space-bounded, since at each step at most one new cell can be written. Thus $\text{P} \subseteq \text{PSPACE}$.

If $L \in \text{PSPACE}$, then by the Space-Time Theorem, there exists a $c \in \mathbb{N}$ with $c > 1$ such that $L \in \text{TIME}(c^{n^k}) \subseteq \text{TIME}(2^{cn^k}) \subseteq \text{TIME}(2^{n^{k+1}}) \subseteq \text{EXPTIME}$. □

For the above theorem, the stronger statement $\text{P} \subset \text{PSPACE} \subset \text{EXPTIME}$ can also be proved, i.e., these complexity classes are proper subsets of each other.

The complexity classes P, PSPACE, and EXPTIME we have encountered are *robust* in the sense that the set of problems belonging to a given complexity class does not change regardless of the computational model used (Turing machine, k-tape Turing machine, RAM machine, Python program, C program, etc.).

4.1 Nondeterministic Turing Machine

In this subsection, we discuss an artificial computational model whose significance lies in the fact that it can "efficiently" solve many "hard" problems that occur in practice. Thereby, these problems become well-characterized and their difficulty can be captured.

Compared to the deterministic Turing machine, the only addition is that multiple possible transitions are allowed.

Definition 4.8. An $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a *nondeterministic Turing machine* (NTM) if

(i) Q, Σ, Γ, q_0, and F have the same meaning as in the definition of the Turing machine;

(ii) $\delta(q, a) \subseteq Q \times \Gamma \times D$ for every $(q, a) \in Q \times \Gamma$.

The transition function can also be given as a relation, since $\delta \subseteq Q \times \Gamma \times D \times Q \times \Gamma$.

The concept of configuration for a nondeterministic Turing machine coincides with that of the deterministic Turing machine. However, the transition function δ allows that from one configuration we may reach several configurations. The processing of a given input word by a nondeterministic Turing machine can be illustrated by a (possibly infinite) rooted tree of configuration transitions. A path from the root, i.e., the starting configuration, corresponds to a possible sequence of choices given by the transition function.

Definition 4.9. A possible sequence of configuration transitions starting from the initial configuration is a *computation path*.

When simulating a nondeterministic Turing machine on a computer, from one configuration with multiple possibilities, we can only choose one at a time, whereas the nondeterministic machine is capable of processing all possible possibilities at a given level of the binary tree simultaneously. This shows that the nondeterministic Turing machine is a model of non-existent computers, since the number of possible transitions per level grows exponentially as we move away from the root.

We have not yet said when a nondeterministic Turing machine accepts a word.

Definition 4.10. The M nondeterministic Turing machine *accepts* the word $x \in \Sigma^*$ if there is at least one computation path ending in an accepting state. Let L_M denote the set of words accepted by M.

As an example, consider the language of graphs that are 3-colorable.[1] The 3-color language consists of those graphs that are 3-colorable. We represent the graph $G = (V, E)$ starting from its adjacency matrix as follows: G can be represented by a bit string of length n^2, where a 1 indicates the adjacency of two vertices.

Example 4.2. Construct a multi-tape nondeterministic Turing machine that decides the 3-color problem, thereby also proving that 3-color $\in$ NP.

The number of steps $T_M(x)$ required by a nondeterministic Turing machine to process a given input x is measured by the length of the longest computation path from the root. Consequently, if there is an infinite computation path, then $T_M(x) = \infty$. After this, the time boundedness of a nondeterministic Turing machine can be defined similarly to deterministic Turing machines.

Definition 4.11. An NTM M is *time-bounded* by $t(n)$ if $T_M(n) \leq t(n)$ for all $n \in \mathbb{N}$, where now $T_M(n)$ is the length of the longest computation path for words of length n.

Analogously to the deterministic case, the corresponding complexity classes can be defined.

Definition 4.12. The (nondeterministic) complexity class corresponding to $t : \mathbb{N} \to \mathbb{N}$ is

$$\mathrm{NTIME}(t(n)) = \{L \subseteq \Sigma^* \mid \exists M \text{ NTM} : L = L_M \text{ and } T_M(n) = O(t(n))\}.$$

The class of languages decidable in polynomial time by a nondeterministic Turing machine is NP.

Definition 4.13. $\mathrm{NP} = \cup_{k=1}^{\infty} \mathrm{NTIME}(n^k)$.

Since deterministic Turing machines are special nondeterministic Turing machines and it can be verified that the step count defined for nondeterministic Turing machines, as well as the concept of word acceptance, coincide with the corresponding concepts defined for them in the deterministic case, the following theorem is obvious.

Proposition 4.3. *P⊆NP.*

Whether the inclusion stated in the above proposition is proper or not is the most significant unanswered question in computer science. The answer has been sought in vain for over fifty years by the best mathematicians and computer scientists. The question is among the millennium prize problems.

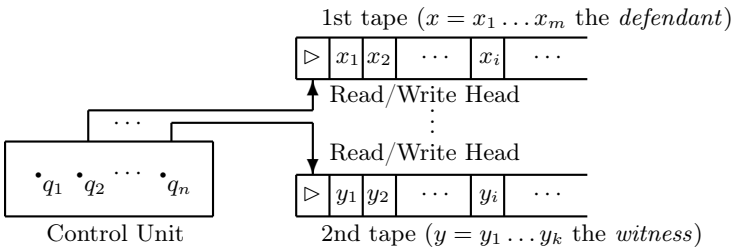


Figure 4.1: Judge Turing Machine.

4.2 Characterizing NP with the Witness Theorem

To verify whether a language is in NP, the following *judge* two-tape Turing machine will be useful.

The two-tape Turing machine shown in Figure 4.1 is special in that y is directly on the second tape. This does not pose a restriction, since a standard two-tape TG with input $x\#y$, by copying y to the 2nd tape, deleting the # separator, and moving the heads to the beginning of the tapes, yields the above type of Turing machine.

The two tapes serve a didactic purpose: in deciding $x \in L$, the witness y can help, potentially reducing our computational burden. Determining whether a problem is in NP by constructing a suitable deciding NTM is a difficult task. Therefore, the following theorem provides significant help in determining whether a problem is in NP.

Theorem 4.4 (NP characterization via witnesses)**.** *The language $L \subseteq \Sigma^*$ is in NP if and only if there exists a positive constant c, and a language $L_1 \in P$ decidable in polynomial time such that*

$$(x \in L \iff \exists y \in \Sigma^* : |y| \leq |x|^c \wedge (x, y) \in L_1)\,.$$

Proof. First, let $L \in$NP. Then there exists a nondeterministic Turing machine N that is n^{c_1} time-bounded and for which $L = L_N$. Let $n = |x|$ and

$$d = \max\{|\delta_N(q, a)| \mid q \in Q_N, a \in \Gamma\},$$

i.e., d is the maximum number of possible continuation options. The witness $y = y_1 y_2 \dots y_m$ encodes the directions of progression step by step along the accepting computation path of the NTM N for $x \in L$ ($y_i \in \{1, \dots, d\}$). Consequently, $m \leq n^{c_1}$. Encoding in binary, $|y| \leq n^{c_1} \lceil \log_2(d+1) \rceil \leq n^c$. In one pass, the Turing machine M checks from the witness tape which step of N to execute, then executes it. This requires a constant number of steps depending on N. Furthermore, M halts exactly if the simulated step of N would result in termination, or if it encounters an invalid witness (i.e., y_i specifies an invalid continuation direction). Finally, M halts in an accepting state if the simulated step of N would lead to acceptance by N.

Clearly, if $x \in L$, there is a good short witness, i.e., $(x, y) \in L_1 := L_M$ and $|y| \leq n^c$, so $L_1 \in$P. If $x \notin L$, then no good witness can be found, since then M simulating N would halt in a rejecting state for any witness y.

It should also be noted that M certainly performs the verification in linear time in the input length (x, y), since each symbol of y corresponds to exactly one step.

Conversely, assume that

$$\exists c > 0 : \exists L_1 \in P : (x \in L \iff \exists y \in \Sigma^* : |y| \leq |x|^c \wedge (x, y) \in L_1).$$

[1]Recall that a graph is k-colorable if it is possible to assign one of $k \in \mathbb{N}$ available colors to each vertex such that adjacent vertices receive different colors.

Since $L_1 \in P$, there is a polynomial-time Turing machine M such that $L_1 = L_M$. We outline the construction of the corresponding NTM N. We may assume that $L_1 \subseteq \{0,1\}^*$. Therefore, when M reads a bit from the witness tape, two corresponding continuation directions must be associated in N via its transition function. For a given $x \in L$, there exists a witness y such that $|y| \leq n^c$ and $(x,y) \in L_1$. The number of such short witnesses y is at most 2^{n^c}. The configuration transitions of the equivalent NTM N to be constructed from a given $x \in L$ can be represented in a binary tree of height at most $O(n^c)$, and hence $L \in$NP. □

Example 4.3. Verify using the NP characterization via witnesses that 3-color is in NP.

A suitable witness y of length $2n$ assigns colors to the n vertices as follows: 01 encodes red, 10 yellow, and 11 green. The given y is sufficiently short, since $2n \leq n^2$ for $n \geq 2$.

Using a RAM machine, traversing the adjacency matrix, we can verify the correctness of the coloring stored in y in polynomial time. Recall that by a previous theorem, this means that the verification can also be performed by a Turing machine in polynomial time. Thus, by the NP characterization via witnesses, 3-color$\in$NP.

Example 4.4. The language H consists of those graphs that contain a Hamiltonian cycle, i.e., starting from a vertex, moving to adjacent vertices step by step such that we do not step on already visited vertices, traversing all vertices and returning to the starting vertex. Show using the NP characterization via witnesses that $H \in$NP.

Let the graph $G = (\{1,\ldots,n\}, E)$ be given by its adjacency matrix represented by n^2 bits. A suitable witness y stores the n vertices to be visited in order in $n\lceil(\log_2 n + 1)\rceil$ bits. Clearly, y is a short witness and whether the path given by the witness y is a Hamiltonian cycle can be verified by a RAM machine in polynomial time.

4.3 NP-Completeness

Definition 4.14. $f : \Sigma^* \to \Sigma^*$ is a *Karp reduction* of $L_1 \subseteq \Sigma^*$ to $L_2 \subseteq \Sigma^*$ if

(i) $\forall x \in \Sigma^* : x \in L_1 \iff f(x) \in L_2$ and

(ii) $f \in$FP.

Notation 4.1. $L_1 \prec L_2$.

Proposition 4.4. *If $L_1 \prec L_2$ and $L_2 \in P$, then $L_1 \in P$.*

Proof. Let $x \in L_1$ and $n = |x|$. The computation of $f(x)$ requires at most $c_1 n^k$ steps due to $f \in$FP, so $|f(x)| \leq c_1 n^k$. Since $L_2 \in$P, $f(x) \in L_2$ can be decided in at most $c_2|f(x)|^l$ steps. Thus, $x \in L_1$ can be decided in $c_2 c_1^l n^{kl}$ steps, i.e., $L_1 \in$P. □

Proposition 4.5. *If $L_1 \prec L_2$ and $L_2 \in$NP, then $L_1 \in$NP.*

Definition 4.15. $L \in \Sigma^*$ is NP-complete if

(i) $L \in$NP and

(ii) for any $L' \in$NP, there exists a Karp reduction $L' \prec L$.

Theorem 4.5 (Cook–Levin Theorem)**.** *SAT is NP-complete.*

Proof. SAT∈NP follows from the NP characterization via witnesses, since a witness for an n-variable Boolean formula is an interpretation, whose length is linear in n.

For any $L \in$NP, we must give a Karp reduction $L \prec$SAT, i.e., we need an $f_L : \Sigma^* \to \Sigma^*$ such that $\forall x \in L \iff f_L(x) \in$ SAT and $f_L \in$FP.

We capture $L \in$NP using the NP characterization via witnesses, i.e.,

$$\exists d > 0, L_1 \in \mathrm{P} : x \in L \Leftrightarrow (\exists y \in \Sigma^* : |y| \leq |x|^d, (x, y) \in L_1).$$

So there is a polynomial-bounded Turing machine M such that $L_M = L_1$. The defendant-witness pair is given on the input tape of $M = (Q, \Sigma, \Gamma, \delta, q_0, \{q_1\})$ in the form $x\#y$. Let $Q = \{q_0, q_1, \ldots, q_r\}$, $\Gamma = \{s_0, s_1, \ldots, s_v\}$, (where $s_0 = \sqcup$, $s_1 = \rhd$ and $s_v = \#$) and $n = |x|$. Then M makes at most n^c steps on the word of length $|x\#y| \leq n^{1+d} + 1$, since M is polynomial-bounded. Moreover, we may assume it makes exactly n^c steps. Therefore, we can document the operation of M in a table with $n^c + 1$ rows and n^c columns, where the $i = 0, \ldots, n^c$ row contains the tape information of machine M after the i-th step. We construct the suitable Boolean formula φ documenting the computation table, whose Boolean variables are

$$\begin{aligned}
Q[i,k] &= \begin{cases} \uparrow, \text{ if } M \text{ is in state } q_k \text{ after step } i, \\ \downarrow, \text{ otherwise;} \end{cases} \\
F[i,j] &= \begin{cases} \uparrow, \text{ if the head is on cell } j \text{ after step } i, \\ \downarrow, \text{ otherwise;} \end{cases} \\
S[i,j,k] &= \begin{cases} \uparrow, \text{ if cell } j \text{ contains } s_k \text{ after step } i, \\ \downarrow, \text{ otherwise.} \end{cases}
\end{aligned}$$

For the word $x \in \Sigma^*$, the Boolean formula φ corresponding to the documented computation sequence of M must describe that (1) the control unit assumes exactly one state, (2) the head is on one cell, (3) any tape cell contains exactly one tape symbol, (4) initially the input is on the tape, (5) the input is accepted in state q_1, and (6) the steps occur according to the transition function δ.

(1) $\varphi_i^1 = (\vee_{k=0}^{r} Q[i,k]) \wedge (\wedge_{0 \leq k < k' \leq r} \neg(Q[i,k] \wedge Q[i,k']))$, where $i = 0, \ldots, n^c$. The number of literals in φ_i^1 equals $(r+1) + 2\binom{r+1}{2} = (r+1)^2$. Let $\varphi^1 = \wedge_{i=0}^{n^c} \varphi_i^1$, which contains $(r+1)^2(n^c+1) = O(n^c)$ literals.

(2) $\varphi_i^2 = \left(\vee_{j=1}^{n^c} F[i,j]\right) \wedge \left(\wedge_{1 \leq j < j' \leq n^c} \neg(F[i,j] \wedge F[i,j'])\right)$, where $i = 0, \ldots, n^c$. The number of literals in φ_i^2 equals $n^c + 2\binom{n^c}{2} = n^{2c}$. Let $\varphi^2 = \wedge_{i=0}^{n^c} \varphi_i^2$, which contains $n^{2c}(n^c+1) = O(n^{3c})$ literals.

(3) $\varphi_{i,j}^3 = (\vee_{k=0}^{v} S[i,j,k]) \wedge (\wedge_{0 \leq k < k' \leq v} \neg(S[i,j,k] \wedge S[i,j,k']))$, where $i = 0, \ldots, n^c$ and $j = 1, \ldots, n^c$. The number of literals in $\varphi_{i,j}^3$ equals $(v+1) + 2\binom{v+1}{2} = (v+1)^2$. Let $\varphi^3 = \wedge_{i=0}^{n^c} \wedge_{j=1}^{n^c} \varphi_{i,j}^3$, which contains $(v+1)^2 n^c (n^c+1) = O(n^{2c})$ literals.

(4) $\varphi^4 = Q[0,0] \wedge F[0,1] \wedge S[0,1,1] \wedge (\wedge_{i=1}^{n} S[0, 1+i, k_i])$, where $x = s_{k_1} s_{k_2} \ldots s_{k_n}$ and the number of literals is $O(n^c)$.

(5) $\varphi^5 = Q[n^c, 1]$, which consists of one literal.

(6) Depending on whether we are dealing with a cell under the head in the i-th row or not, two cases must be distinguished:

a. $\varphi^6_{i,j} = (\neg F[i,j] \Rightarrow (\wedge^v_{k=0}(S[i,j,k] \Leftrightarrow S[i+1,j,k])))$, where $i = 0,\ldots,n^c$ and $j = 1,\ldots,n^c$. Let $\varphi^6 = \wedge^{n^c}_{i=0} \wedge^{n^c}_{j=1} \varphi^6_{i,j}$. It can be verified that the number of literals in φ^6 is $O(n^{2c})$.

b. $\varphi^7_{i,j,k,l} = ((Q[i,k] \wedge F[i,j] \wedge S[i,j,l]) \Rightarrow (Q[i+1,k'] \wedge F[i+1,j'] \wedge S[i+1,j,l']))$, where $i = 0,\ldots,n^c-1$, $j = 1,\ldots,n^c$, $k = 0,\ldots,r$, $l = 0,\ldots,v$, $\delta(q_k,s_l) = (q_{k'},s_{l'},d)$ and

$$j' = \begin{cases} j+1, & \text{if } d = \rightarrow; \\ j, & \text{if } d = - \text{ and} \\ j-1, & \text{if } d = \leftarrow . \end{cases}$$

Note that if $\delta(q_k,s_l)$ is undefined and $i < n^c$, then we take it as if $\delta(q_k,s_l) = (q_k,s_l,-)$. Let $\varphi^7 = \wedge^{n^c-1}_{i=0} \wedge^{n^c}_{j=1} \wedge^r_{k=0} \wedge^v_{l=0} \varphi^7_{i,j,k,l}$. It can be verified that the number of literals in φ^7 is $O(n^{2c})$.

Finally, let $\varphi = \wedge^7_{m=1}\varphi^m$, whose number of literals is polynomial in n. Based on this, it easily follows that the computation of the function f_L that produces φ for a given L can be performed in polynomial time.

The function f_L indeed fulfills its task, since $x \in L \iff \exists y : (x,y) \in L_M \iff \exists I : I \models \varphi \iff \varphi \in \text{SAT}$. □

Theorem 4.6. *If $L_1 \in$NP-complete, $L_2 \in$NP and $L_1 \prec L_2$, then $L_2 \in$NP-complete.*

Proof. Since $L_1 \in$NP-complete, $\forall L' \in$NP:$\exists \prec$: $L' \prec L_1$. Let $f_{L'} : \Sigma^* \to \Sigma^*$ be the mapping in the reduction and $g : \Sigma^* \to \Sigma^*$ be the mapping corresponding to the reduction $L_1 \prec L_2$. Then $f = g \circ f_{L'}$ provides the mapping for $L' \prec L_2$. □

Steps for NP-completeness proofs:

(i) Prove $L \in$NP (usually with the NP characterization via witnesses),

(ii) choose a suitable $L' \in$NP-complete language,

(iii) construct an f such that $x \in L' \iff f(x) \in L$ for all $x \in \Sigma^*$ and

(iv) prove $f \in$FP.

4.4 NP-Complete Problems

Definition 4.16. A 3-CNF is a special Boolean formula given in CNF form where each clause contains at most three literals.

Definition 4.17. The 3-SAT language is the set of satisfiable Boolean formulas given in 3-CNF form.

Theorem 4.7. *3-SAT is NP-complete.*

Proof. 3-SAT is in NP, since a satisfying assignment is a suitable witness.

SAT is the only NP-complete language we have encountered so far. Therefore, only the reduction of SAT to 3-SAT can be considered.

We give the mapping f that assigns a 3-CNF form Boolean formula to Boolean formulas.

For the formula φ, we construct an equivalent Boolean circuit $G = (V,E)$ (in the known way). The Boolean variables in φ are $x_1,\ldots,x_n$ and $V = \{1,\ldots,m\}$. For each vertex of the Boolean circuit, we assign a Boolean variable z_i, which is true for an interpretation I if and only if the value computed by gate i during the evaluation of the circuit is true. So for any $i \in V$:

(i) $\varphi_i := (z_i \Leftrightarrow s(i))$, if $s(i) \in \{\uparrow, \downarrow, x_1, \ldots, x_n\}$;

(ii) $\varphi_i := (z_i \Leftrightarrow \neg z_j)$, if $s(i) = \neg$ and $(j, i) \in E$;

(iii) $\varphi_i := (z_i \Leftrightarrow z_j \vee z_k)$, if $s(i) = \vee$, $(j, i) \in E$, $(k, i) \in E$ and $j \neq k$;

(iv) $\varphi_i := (z_i \Leftrightarrow z_j \wedge z_k)$, if $s(i) = \wedge$, $(j, i) \in E$, $(k, i) \in E$ and $j \neq k$.

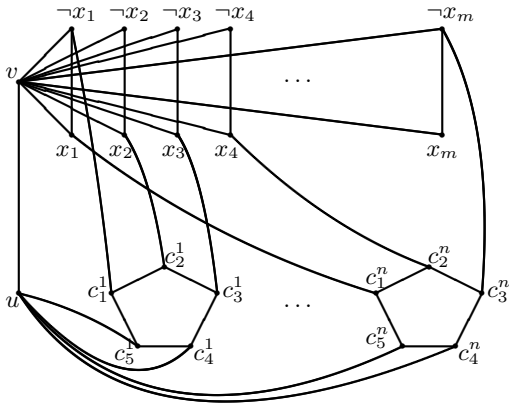
The φ_i's indicate to us that the evaluation is performed correctly, according to the circuit. Therefore, $\varphi \equiv (\wedge_{i=1}^{m} \varphi_i) \wedge z_m$. Note that z_m must be included in the conjunction since z_m is the result of the computation. The obtained expression is not yet in 3-CNF form. For this, the individual φ_i's, which already contain at most three variables, can be written as disjunctions of three literals using the equivalence $(p \Leftrightarrow q) \equiv ((\neg p \vee q) \wedge (\neg q \vee p))$, thus obtaining ψ_i's. So $\varphi \equiv (\wedge_{i=1}^{m} \psi_i) \wedge z_m$ is now in 3-CNF form. It remains to show that $f \in$FP, i.e., the construction of $(\wedge_{i=1}^{m} \psi_i) \wedge z_m$ from φ can be done in polynomial time. The latter follows from the fact that (V, E) can be built from φ in polynomial time and the maximum size of ψ_i is independent of n. □

Theorem 4.8. 3-*color is NP-complete.*

Proof. We have already seen that 3-color is in NP. We will show that 3-SAT$\prec$ 3-color.

For $\psi \in$ 3-SAT, we construct a graph $G = (V, E) \in$ 3-color, i.e., we give f. Let ψ have m variables and n clauses, i.e., $\psi = C_1 \wedge \ldots \wedge C_n$. We may assume that the C_i's consist of three literals (e.g., $x_1 \vee \neg x_2 \equiv x_1 \vee x_1 \vee \neg x_2$). The vertices of the graph assigned to ψ consist of the variables of ψ $(x_1, \ldots, x_m)$ and their negations $(\neg x_1, \ldots, \neg x_m)$, furthermore, five vertices per clause (C_i has $c_i^1, c_i^2, c_i^3, c_i^4, c_i^5$), and finally the vertices u and v. Let v be connected to $u, x_1, \ldots, x_m, \neg x_1, \ldots, \neg x_m$, u be connected to $v, c_1^4, c_1^5, \ldots, c_n^4, c_n^5$, furthermore, let the vertices $c_i^1, c_i^2, c_i^3, c_i^4, c_i^5$ form cycles of length five in G, and finally, let the vertices c_i^1, c_i^2, c_i^3 be connected to the three literals in clause C_i $(i = 1, \ldots, n)$. The construction is illustrated in Figure 4.2:

Figure 4.2: 3-*color*∈*NP-complete.*



In the figure, $C_1 = \neg x_1 \vee x_2 \vee x_3$ and $C_n = x_1 \vee x_4 \vee \neg x_m$.

We show that if $G \in$ 3-color, then $\psi \in$ 3-SAT. Take a 3-coloring of G. Let v be red and u yellow. Then the colors of x_i and $\neg x_i$ are different, and only yellow or green $(i = 1, \ldots, m)$. Among the literals, let the green ones be true. This gives us an interpretation I. Clearly, $I \models \psi$ if and only if every C_i $(i = 1, \ldots, n)$ has a green literal. Assume that $\exists i$ such that C_i has all false literals. Then the vertices c_i^1, c_i^2, c_i^3 cannot be yellow. But c_i^4, c_i^5 also cannot be yellow,

since u is yellow. This leads to a contradiction, since an odd-length cycle cannot be colored with two colors. So every clause has a true literal, and therefore $I \models \psi$, i.e., $\psi \in$ 3-SAT.

Conversely, we need to show that if $\psi \in$ 3-SAT, then $G \in$ 3-color. Take an interpretation I satisfying ψ. Color the vertices corresponding to true literals green, and those corresponding to false ones yellow. Also color v red and u yellow. After this, it can be shown by case analysis, that the vertices of the pentagons corresponding to the clauses can be colored with three colors. Only the polynomial computability of the given mapping f remains. But this is almost obvious, since the size of the graph is linear in n and m, and a coloring directly provides the corresponding interpretation. □

Definition 4.18. In a graph $G = (V, E)$, a set $X \subseteq V$ is *independent* if the vertices in X are all non-adjacent.

Definition 4.19. The INDSET problem decides whether a graph $G = (V, E)$ contains an independent set of size at least k.

Theorem 4.9. *INDSET is NP-complete.*

Proof. By the NP characterization via witnesses, INDSET is in NP, since a set of vertices is a suitable short witness.

We give a Karp reduction 3-color$\prec$INDSET. We seek a mapping that assigns INDSET problems to 3-color problems such that $G \in$ 3-color if and only if $(G_1, k) \in$INDSET.

From G, take three copies $G^{(1)}$, $G^{(2)}$, and $G^{(3)}$ of G, and connect the copies of the vertices of G accordingly, to obtain G_1. Let $k = |V(G)|$.

We show that if $G \in$ 3-color, then $G_1 \in$INDSET. Take a coloring of G with colors 1, 2, and 3. The set $H \subseteq V(G_1)$ is obtained by taking for each $v \in V(G)$ the copy of v in $G^{(i)}$ corresponding to the color $i \in \{1, 2, 3\}$ of v in G. From the coloring, H is a k-element independent set.

Conversely, if $G_1 \in$INDSET, then we show that $G \in$ 3-color. Take a k-element independent set H in G_1, which can contain exactly one vertex from the three copies of each vertex. Color the vertex $v \in H$ with color $i \in \{1, 2, 3\}$ if $v \in G^{(i)}$. This colors G with 3 colors, because if $u, v \in V(G)$ have the same color, then they are in the same copy of G, and since they are in H, they are not adjacent. □

Definition 4.20. A complete subgraph $G' \subseteq G$ of $G = (V, E)$ is a *clique*.

Definition 4.21. The MAXCLIQUE problem decides whether G contains a clique of size at least k.

Corollary 4.1. *MAXCLIQUE is NP-complete.*

Proof. Clearly MAXCLIQUE$\in$NP. $X \subseteq V(G)$ is independent in G if and only if X is a clique in $\overline{G}$. With $f : (G, k) \to (\overline{G}, k)$, we get INDSET$\prec$MAXCLIQUE. □

4.5 Other Notable NP-Complete Problems

Definition 4.22. Given $\mathcal{R} \subseteq X \times Y \times Z$, where $q = |X| = |Y| = |Z|$. A *perfect matching* H of $\mathcal{R}$ is a subset $H \subseteq \mathcal{R}$ with $|H| = q$ such that all elements of X, Y, and Z appear in the elements of H. The 3DM problem is to decide whether there exists a perfect matching in $\mathcal{R}$.

Definition 4.23. An instance of the X3C problem is a pair $(X, \mathcal{C})$, where the set X has $3q$ elements and $\mathcal{C} = \{C_1, C_2, \ldots, C_r\}$ is a family of 3-element subsets of X. The X3C problem is to decide whether X can be covered by pairwise disjoint sets from $\mathcal{C}$ (i.e., $\left(\exists C_{i_1}, \ldots, C_{i_q} \in \mathcal{C} : \cup_{n=1}^{q} C_{i_n} = X\right)$ and $(\forall n, m \in \{1, \ldots q\} : n \neq m \Rightarrow C_{i_n} \cap C_{i_m} = \emptyset)$).

Definition 4.24. The HAMILTON problem is the problem of detecting Hamiltonian cycles.

Definition 4.25. Given a graph $G = (V, E)$ with costs $c : E \to \mathbb{Z}_+$. The TRAVELING SALESMAN problem seeks Hamiltonian cycles in G with cost at most k.

Definition 4.26. Given m items with weights $s_1, \ldots, s_m > 0$, values $v_1, \ldots, v_m > 0$, and a knapsack with allowed total weight $b > 0$. The KNAPSACK problem seeks a set $I \subseteq \{1, \ldots, m\}$ of items such that $\sum_{i \in I} s_i \leq b$ and $\sum_{i \in I} v_i$ is maximized.

Theorem 4.10. *3DM, X3C, HAMILTON, TRAVELING SALESMAN, and KNAPSACK are NP-complete.*

4.6 Optimization Problems

Example 4.5. Let MAXINDSET be the problem of finding the size of the maximum independent set. We consider that MAXINDSET and INDSET are equally "hard".

Solution: If we have an algorithm that solves MAXINDSET, then after determining the optimum value k^*, a simple comparison $k \leq k^*$ decides whether $(G, k) \in$INDSET.

Take an algorithm that decides INDSET and a graph $G = (V, E)$. Let $n = |V|$. First, use our algorithm to decide whether $(G, \lceil n/2 \rceil) \in$INDSET. If yes, then check $(G, \lceil 3n/4 \rceil) \in$INDSET; if no, then check $(G, \lceil n/4 \rceil) \in$INDSET. Continuing the procedure with a logarithmic search, we obtain the optimum value. So after $\lceil \log_2 n \rceil$ calls to INDSET, we get the optimum value.

It is not surprising that MAXINDSET is no easier than INDSET. But now we also see that compared to INDSET, the operational cost of MAXINDSET does not even increase by a linear factor.

Remark 4.1. If the set of possible optimum values is finite, then with the reasoning in the example, we can establish a connection between the optimization problem and the corresponding decision problems.

4.7 Handling NP-Complete Problems

- Establishing NP-completeness is only the first step.
- For small-sized problems, exponential algorithms may still be useful.
- Approximability of optimization problems.
- Randomized algorithms (e.g., see RSA key generation).
- Search strategies using heuristics.
- "Soft computing" models:
  - genetic algorithms,
  - neural networks,
  - fuzzy logic, etc.

4.8 Approximation Algorithms

Let x be an instance of our optimization problem. Denote $\mathrm{OPT}(x) > 0$ the optimum and $c(M(x)) > 0$ the value obtained by the approximation algorithm M.

Definition 4.27. We say that M is an ε-approximation algorithm ($\varepsilon > 0$) if

$$\frac{|c(M(x)) - \mathrm{OPT}(x)|}{\max\{\mathrm{OPT}(x), c(M(x))\}} \leq \varepsilon,$$

for every instance x of the problem. ε is the upper bound of the relative error of M.

Definition 4.28. The *approximation threshold* of a given optimization problem is the greatest lower bound of those ε for which there exists a polynomial-time ε-approximation algorithm.

Consider the VERTEX COVER$\in$NP-complete problem. Recall that $X \subseteq V$ is a *vertex cover* of the graph $G = (V, E)$ if every edge of G has at least one endpoint in X. So an instance of the problem is given by G and we seek $\tau(G)$, the size of the minimum vertex cover.

We determine the relative error bound of the following approximation algorithm:

$X := \emptyset$;
`while` $|E(G)| > 0$ `repeat`
    $uv :\in E(G)$; $X := X \cup \{u, v\}$; $G := G - uv$;
`end while`

$G := G - uv$ deletes the edges incident to u or v.

X is a vertex cover. Since the removed edges form a matching in G, the minimum vertex cover must contain at least one endpoint of each removed edge. So $\frac{1}{2}|X| \leq \tau(G)$, i.e., $\frac{1}{2}$ is an upper bound for the approximation threshold.

We actually do not know a better approximation algorithm than this. If we allow ε to depend on the input size $n = |\#x|$, then $\varepsilon(n) < \frac{1}{2}$ and $\varepsilon(n) \to \frac{1}{2}$ holds for the best known approximation algorithm. Dinur and Safra (2005) proved that the relative error bound is at least 0.3606. This was improved in a series of papers between 2016 and 2018 by Dinur, Khot, Kindler, Minzer and Safra to 0.414. It is believed that the exact value is 1/2. So the VERTEX COVER problem is not "well" approximable.

Remark 4.2. For bipartite graphs, $\tau(G)$ can be computed in polynomial time.

Theorem 4.11. *If P$\neq$NP, then the approximation threshold of the TRAVELING SALESMAN problem is* 1.

Proof. Assume that there is a polynomial ε-approximation algorithm M ($\varepsilon < 1$), with the help of which we give a polynomial-time algorithm for HAMILTON.

Take an instance $G = (V, E)$ of HAMILTON. Let $n = |V|$. Assign to it the instance of TRAVELING SALESMAN* $((V, E'), d)$, where $(V, E') = K_n$, $d : E' \to \mathbb{R}_+$, $d(e) = 1$ if $e \in E$, and $d(e) = \lceil n/(1-\varepsilon) \rceil$ if $e \notin E$.

Two cases are possible for TRAVELING SALESMAN*: (i) we get a tour of cost n, which is equivalent to $G \in$HAMILTON; or (ii) we get a tour of cost greater than $n/(1-\varepsilon)$, which is equivalent to $G \notin$HAMILTON.

Applying our polynomial ε-approximation algorithm to the TRAVELING SALESMAN* instance $((V, E'), d)$, we can decide $G \in$HAMILTON in polynomial time, because the difference

between the results of the two cases would result in a relative error greater than or equal to ε, since if $G \in$HAMILTON is falsely identified, the relative error is greater than

$$\frac{\frac{n}{1-\varepsilon} - n}{\frac{n}{1-\varepsilon}} = \frac{1-(1-\varepsilon)}{1} = \varepsilon,$$

which contradicts the NP-completeness of HAMILTON. □

Remark 4.3. If the distance function d satisfies the triangle inequality, i.e., $d(uv) \leq d(uw) + d(wv)$ for all $uv, uw, wv \in E$, then a polynomial $\frac{1}{3}$-approximation algorithm can be given.

Theorem 4.12. *The approximation threshold of the KNAPSACK problem is* 0.

Proof. An instance of KNAPSACK is $x = (s_1, \ldots, s_n, b, v_1, \ldots, v_n) \in \mathbb{Z}_+^{2n+1}$ and we want $\max\{\sum_{i\in I} v_i \mid I \subseteq \{1, \ldots, n\}, \sum_{i\in I} s_i \leq b\}$. Let $V := \max\{v_1, \ldots, v_n\}$ and

$$W(i, v) := \min\{\sum_{j\in J} s_j \mid J \subseteq \{1, \ldots, i\}, \sum_{j\in J} s_j = v\}$$

for every $i = 0, 1, \ldots, n$ and $v = 0, 1, \ldots, nV$. To compute W recursively, let $W(0, 0) := 0$ and $W(0, v) := \infty$ for all $v = 1, \ldots, nV$, then

$$W(i+1, v) := \begin{cases} \min\{W(i, v), W(i, v - v_{i+1}) + s_{i+1}\}, & \text{if } v \geq v_{i+1}, \\ W(i, v), & \text{if } v < v_{i+1}. \end{cases}$$

Since v takes n values and v can take $nV + 1$ values in total, the operation count is $O(n^2V)$. Finally, determine the largest weight satisfying

$$v := \max\{v' \mid v' = 0, 1, \ldots, nV \text{ and } W(n, v') \leq s\},$$

thus defining the algorithm M. The problem is that the value of V is large, specifically exponential in the number of digits of V.

From the values in instance x, drop the last k bits, i.e., $v_i' := 2^k \lfloor v_i/2^k \rfloor$, thus obtaining the approximate instance $x' = (s_1, \ldots, s_n, b, v_1', \ldots, v_n')$. The operational cost of solving instance x' is $O(n^2V/2^k)$. Let I be the optimal set of items for x and I' for x'. Then for the optimal solutions

$$\sum_{i\in I} v_i \geq \sum_{i\in I'} v_i \geq \sum_{i\in I'} v_i' \geq \sum_{i\in I} v_i' \geq \sum_{i\in I} (v_i - 2^k) \geq \left(\sum_{i\in I} v_i\right) - n2^k.$$

Considering that $V \leq$OPT(x) (since we may assume $s_i \leq b$ for all $i = 1, \ldots, n$),

$$\frac{\text{OPT}(x) - c(M(x))}{\text{OPT}(x)} \leq \frac{n2^k}{V} = \varepsilon.$$

For a given relative error bound $\varepsilon > 0$, let

$$k := \max\left\{\left\lceil \log_2 \frac{\varepsilon V}{n} \right\rceil, 0\right\}.$$

Then the operational cost of the ε-approximation algorithm M is

$$O\left(\frac{n^2V}{2^k}\right) = O\left(\frac{n^3}{\varepsilon}\right),$$

so it is polynomial time. Since $\varepsilon > 0$ can be chosen arbitrarily small, the approximation threshold is 0. □

4.9 Exercises

Exercise 4.1. Prove that deciding whether a positive integer is composite is an NP problem! The input size is measured by the number of digits needed to represent the given positive number.

Exercise 4.2. A set $X \subseteq V$ is a *vertex cover* of the graph $G = (V, E)$ if every edge of G has at least one endpoint in X. Prove that the problem of the existence of a vertex cover of size at most $k \in \mathbb{N}$ in $G = (V, E)$ is in NP!

Exercise 4.3. Given integers $a_1, a_2, \ldots, a_n$, decide whether there exists an $S \subseteq N := \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = \sum_{i \in N \setminus S} a_i$. Prove that the problem defined this way is in NP!

Exercise 4.4. Given a directed graph $G = (V, E)$ and four distinct vertices $s_1, s_2, t_1, t_2 \in V$, decide whether there exist disjoint directed paths in G from s_1 to t_1 and from s_2 to t_2, where two directed paths are disjoint if they have no common vertices. Prove that the problem defined this way is in NP!

Exercise 4.5. Given a graph $G = (V, E)$ and two distinct vertices $s, t \in V$, decide whether there exist two edge-disjoint paths from s to t in G, where two paths are edge-disjoint if they have no common edges. Prove that the problem defined this way is in NP!

Exercise 4.6. Let $\mathcal{R} \subseteq W \times X \times Y \times Z$ be given, where $q = |W| = |X| = |Y| = |Z|$. A *perfect matching* N of $\mathcal{R}$ is a subset $N \subseteq \mathcal{R}$ with $|N| = q$ such that all elements of W, X, Y, and Z appear in the elements of N. The 4DM problem is to decide whether there exists a perfect matching in $\mathcal{R}$. Prove the NP-completeness of 4DM using the known NP-completeness of 3DM!

Exercise 4.7. A pair $(X, \mathcal{C})$ is given, where the set X has $4q$ elements and $\mathcal{C} = \{C_1, C_2, \ldots, C_r\}$ is a family of 4-element subsets of X. The X4C problem is to decide whether X can be covered by pairwise disjoint sets from $\mathcal{C}$! Prove that X4C is NP-complete, using the NP-completeness of X3C!

Exercise 4.8. A set X and a family $\mathcal{C}$ of its subsets, and a positive integer K are given. Decide whether there exists a set $\mathcal{C}' \subseteq \mathcal{C}$ of at least K elements such that

$$\bigcup_{S \in \mathcal{C}'} S \subseteq X \text{ and } S \cap S' = \emptyset \text{ for any } S, S' \in \mathcal{C}', S \neq S'.$$

Prove the NP-completeness of the given problem using one of the known NP-complete problems!

Exercise 4.9. A set S and a family $\mathcal{C}$ of its subsets, and a positive integer K are given. Decide whether there is a set $\mathcal{C}' \subseteq \mathcal{C}$ of at most K elements that covers the set S, i.e.,

$$S \subseteq \bigcup_{S' \in \mathcal{C}'} S'.$$

Prove the NP-completeness of the given problem using one of the known NP-complete problems!

Exercise 4.10. A set S and a family $\mathcal{C}$ of its subsets, and a positive integer K are given. Decide whether there is a set $S' \subseteq S$ of at most K elements that contains at least one element of every set in $\mathcal{C}$. Prove the NP-completeness of the problem defined this way using the NP-completeness of the vertex cover problem!

Exercise 4.11. The K3PARTITION problem decides for a simple graph $G = (V, E)$ with $3q$ vertices whether it can be partitioned into K_3's, i.e., whether there exist $G_1 = (V_1, E_1), \ldots, G_q = (V_q, E_q)$ complete three-vertex subgraphs of G such that $V = V_1 \cup V_2 \cup \ldots \cup V_q$. Prove the NP-completeness of K3PARTITION using one of the known NP-complete problems!

Exercise 4.12. What does it tell us if the approximation threshold of a minimization problem is $\frac{1}{3}$?

Exercise 4.13. Let $V = \{v_1, \ldots, v_n\}$ be the vertex set of graph $G = (V, E)$ and let the numbers $1, \ldots, n$ denote the available colors. Color the vertices $v_1, \ldots, v_n$ one after the other such that the currently colored vertex v_i is colored with the smallest number different from the colors of its already colored neighbors. Determine the lower bound of the relative error of the approximation!

Exercise 4.14. In a simple graph $G = (V, E)$, we search for the longest path as follows:

1. Choose a vertex with the highest degree.
2. As long as possible, proceed via neighbors with the highest degree, ensuring that no cycle is formed.
3. Starting again from the vertex chosen in step 1, proceed via the second highest degree neighbor, then via neighbors with the highest degree, ensuring that no cycle is formed.

Determine the lower bound of the relative error of the approximation!

Exercise 4.15. We approximate the MAXINDSET problem with the following algorithm.

$X := \emptyset$;
`while` $|V(G)| > 0$ `repeat`
    $u :\in \arg\max\{\deg(v) \mid v \in V(G)\}$;
    $X := X \cup \{u\}$;
    $G := G - (\{u\} \cup N(\{u\}))$;
`end while`

Interpretation: the graph G is the input of the program, $G := G - (\{u\} \cup N(\{u\}))$ means deleting u, the vertices adjacent to u, and the edges incident to them from G, and X gives the determined independent set.
Determine the lower bound of the relative error of the approximation!

Chapter 5

On the boundary of economics and computer science

5.1 Algorithmic Mechanism Design

In this chapter, through a cost-sharing problem defined on a network, we point out the opportunities in mechanism design. In the problem studied, we wish to deliver a packet from a sender to a recipient through a network where the actual costs of the individual connections are only known to the respective service providers. Our goal is to determine the minimum cost route in the presence of self-interested service providers, an example which appeared in the mechanism design context in the paper by Nisan and Ronen (2001).

We start the subsection with a brief introduction to non-cooperative game theory, and then present one of the fundamental results of Nisan and Ronen (2001).

5.1.1 Basic concepts of non-cooperative game theory

Assume that in a game, the participants make their decisions separately from each other and know the rules of the game. Such a game structure is called a strategic game.

Definition 5.1. The triple $(N, (S_i)_{i\in N}, (u_i)_{i\in N})$ is a *strategic game*, where

- N is a finite non-empty set of players;
- S_i is the strategy set of player $i \in N$, which can be an arbitrary non-empty set;
- $u_i : \times_{i\in N} S_i \to \mathbb{R}$ is the utility function of player $i \in N$.

We will need some notation. For a vector $x = (x_1, \ldots, x_n)$, denote by x_{-i} the vector $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ without the i-th component. Then (x_i, x_{-i}), or (x_{-i}, x_i), denotes the original vector x. If $X = \times_{i=1}^{n} X_i$, then $X_{-i} = X_1 \times \cdots \times X_{i-1} \times X_{i+1} \times \ldots X_n$.

As an example, consider the following game: we want to model a penalty kick. The goalkeeper has three options: stay in the center, dive left, or dive right. The shooter can kick the ball to the center, left, or right. $N = \{\text{goalkeeper}, \text{shooter}\}$, $S_{\text{goalkeeper}} = \{left, center, right\}$, $S_{\text{shooter}} = \{left, center, right\}$,

$$u_{\text{goalkeeper}}(s) = \begin{cases} 1, & \text{if } s_{\text{goalkeeper}} = s_{\text{shooter}} \\ 0, & \text{otherwise.} \end{cases}$$

and

$$u_{\text{shooter}}(s) = \begin{cases} 1, & \text{if } s_{\text{goalkeeper}} \neq s_{\text{shooter}} \\ 0, & \text{otherwise.} \end{cases}$$

is a simplified formulation of the penalty kick as a strategic game.

With the strategic game, we describe the decision situation of the participants, but we also need to say what the "outcome" of the strategic game is. For this, different types of equilibrium concepts will serve our purposes. An equilibrium of a system – very generally and broadly speaking – refers to states that can be considered constant or stable in some respect. In games, we can consider the strategy sets of the players as the state space. We assume that the players behave rationally, i.e., they wish to maximize their utility. It is not at all obvious which states we should consider equilibria, since the players wish to maximize their utility simultaneously.

The simplest equilibrium concept is the dominant strategy equilibrium. Its main advantage is that if a dominant strategy equilibrium exists, no player needs to concern themselves with the reasoning and chosen strategy of the other players.

Definition 5.2. The strategy profile $s^* \in S$ of the strategic game $G = (\{1,\dots,n\},(S_i)_{i=1}^n,(u_i)_{i=1}^n)$ is a *dominant strategy equilibrium* of G if

$$\forall i \in \{1,\dots,n\} : \forall s_i' \in S_i : \forall s_{-i} \in S_{-i} : u_i(s_i^*, s_{-i}) \geq u_i(s_i', s_{-i}),$$

where $S = \times_{i=1}^n S_i$.

As an example, consider the famous prisoner's dilemma, given in Table 5.1. According to the story, two accomplices are interrogated separately, and depending on whether they confess or not, they receive the prison sentences in years shown in Table 5.1, where the row header shows player A's decision and the column header player B's decision. In each entry of the table, the first number is player A's prison sentence, while the second is player B's. Overall, they would be best off if neither confessed against the other, since then together they would only serve 2 years for their lesser provable crimes. Since they cannot coordinate their decisions, they are better off confessing regardless of the other's decision. In other words: confessing to the crime is a dominant strategy for both of them. According to this reasoning, the only dominant equilibrium of the game is (confess, confess), which is the equilibrium outcome of the game.

Table 5.1: Prisoner's Dilemma

A\B	not confess	confess
not confess	-1, -1	-10, 0
confess	0, -10	-8, -8

Since a dominant strategy equilibrium often does not exist, it becomes necessary to introduce another equilibrium concept, the Nash equilibrium. Its essence is to call a point an equilibrium if no player has an incentive to deviate from it alone.

Definition 5.3. The strategy profile $s^* \in S$ of the strategic game $G = (\{1,\dots,n\},(S_i)_{i=1}^n,(u_i)_{i=1}^n)$ is a *Nash equilibrium*, if

$$\forall i \in \{1,\dots,n\} : \forall s_i \in S_i : u_i(s_i^*, s_{-i}^*) \geq u_i(s_i, s_{-i}^*).$$

Not every game has a Nash equilibrium. A good example is the penalty kick game mentioned earlier. For the interested reader, we recommend the electronic notes of Forgó et al. (2005).

5.1.2 Mechanism Design on a Network

We represent the network by the directed graph $G = (V, E)$, where the connections (i.e., edges) in E between users in V are owned by the players (service providers). The direct costs of forwarding a packet are given by the function $c : E \to \mathbb{R}_+$. An example of such a network with costs is shown in Figure 5.1. We assume that packet sending can be performed even if any edge fails.

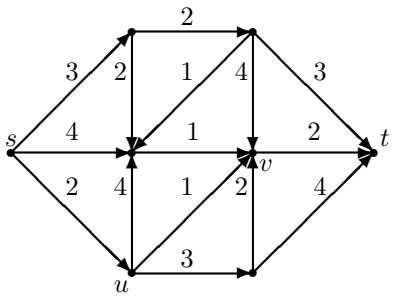


Figure 5.1: Network with Private Costs

Our task is to deliver a packet from vertex $s \in V$ to vertex $t \in V$ in the network with the least possible cost. In the network shown in Figure 5.1, the minimum cost route is the path s, u, v, t, with a total cost of 5 units. The shortest path between two points can be quickly obtained, for example, by Edsger Dijkstra's algorithm (see, e.g., Rónyai et al., 1999). The problem is that the costs of the edges are private information of the providers of the connections, and therefore we can only run Dijkstra's algorithm on the reported, not necessarily true, costs. Nevertheless, we want to achieve that the participants (the owners of the individual edges or connections, henceforth players) voluntarily report their true costs through a suitably constructed game (henceforth mechanism), and thereby the packet indeed travels along the least cost route from s to t. The players are thus the owners of the edges, and their strategy set is the set of possible costs of the edge they own, in this case the set of nonnegative real numbers. Based on the reported costs, the mechanism determines the minimum cost route, which delivers the packet along the shortest path. We wish to define the utility functions of the mechanism such that the shipment from s to t travels along the true minimum total cost route. Clearly, it is not goal-oriented if a player on the shortest route receives their reported costs, since a player, knowing only their own costs, might feel incentivized to report a higher cost than the actual one to increase their profit. In the network of Figure 5.1, assuming the other participants report their actual costs shown in the figure, the (s, u) player could, for example, raise their cost to 3, thereby remaining on the minimum cost route while realizing a profit.

We design a game defined over a network, for now forgetting under which equilibrium concept (e.g., dominant or Nash equilibrium) we wish to determine the outcome of the game. Assume, therefore, that the designer wishes to ensure that the packet always travels along the true – but unknown to them – minimum cost route from the sender to the recipient. If there happen to be multiple minimum cost routes, the packet should travel on one of them. To achieve the designer's goals, they use transfer payments, i.e., depending on the reported costs, they may give or take away different amounts from the players to achieve their goal. For example, if the edge (s, u) in Figure 5.1 were to fail – which can also be represented by taking the cost of edge (s, u) to be infinitely large – then the route s, z, v, t would become the minimum cost route, with a total cost of 7, which is 2 more than the minimum total cost of the original network. So (s, u), in addition to its true cost of 2, could claim at most an additional 2 units, and thus report a total of 4 units while still remaining on a minimum cost route. This 4 units is called the marginal contribution of the (s, u) player, with which the

designer rewards the presence of the (s, u) player.

In general, the marginal contribution $MC_{\tilde{c}}(e)$ of player $e = (u, v) \in E$ is determined based on the reported costs $\tilde{c} : E \to \mathbb{R}_+$ – i.e., the strategy profile of the players – by calculating the cost $\widetilde{C}_{|\tilde{c}(e)=\infty}$ of the minimum total cost route when player e is removed from the network based on the reported costs of the others, then calculating the cost $\widetilde{C}_{|\tilde{c}(e)=0}$ of the minimum total cost route when the cost of player e is zero, again based on the reported costs of the others. This gives us the maximum amount that player e can demand, provided they wish to remain on a minimum total cost route, if this is at all possible given the reported costs $\tilde{c} : E \to \mathbb{R}_+$. Finally, the marginal contribution is the difference of the two values:

$$MC_{\tilde{c}}(e) = \widetilde{C}_{|\tilde{c}(e)=\infty} - \widetilde{C}_{|\tilde{c}(e)=0}.$$

In other words, $MC_{\tilde{c}}(e)$ is the maximum amount by which e can reduce the total cost. Summarizing, in the (N, S, u) game we construct, $N = E$, $S_i = \mathbb{R}_+$ and

$$u_e(\tilde{c}) = \begin{cases} \widetilde{C}_{|\tilde{c}(e)=\infty} - \widetilde{C}_{|\tilde{c}(e)=0} - c_e, & \text{if } e \text{ forwards, and} \\ 0, & \text{otherwise,} \end{cases}$$

where $c_e = c(e)$ is the actual cost of player e.

For players not on the minimum total cost route, their marginal contributions are taken as zero. To achieve the designer's goal, the individual players are given their marginal contribution calculated based on the reported $\tilde{c} : E \to \mathbb{R}_+$ costs. Note that under the mechanism devised by the designer, the players' (transfer) incomes do not depend on their own costs. The profit of a player on the minimum total cost route is the difference between the amount received as a transfer payment and the cost incurred by forwarding the packet. The profits of players not involved in packet forwarding are all zero. Assume that the players report their reported costs to the designer independently, in a simultaneous manner.

Assuming that the reported costs in Figure 5.1 are also the true costs, s, u, v, t is the shortest path, $u_{su} = (7 - 3) - 2 = 2$, $u_{uv} = (7 - 4) - 1 = 2$ and $u_{vt} = (8 - 3) - 2 = 3$. In addition to the incurred cost of $2 + 1 + 2 = 5$, the operator of the mechanism or the sender must pay an additional $2 + 2 + 3 = 7$ units.

Theorem 5.1. *In the unique dominant equilibrium of the above game, $s_e = c_e$ for all $e \in E$, i.e., everyone reports their true cost.*

Proof. Note that for any player $e \in E$, u_e payoff only depends on s_e insofar as it influences whether e is on the minimum total cost route. Moreover, the amount of the payoff is independent of s_e, provided that e remains on the minimum total cost route.

$s_e > c_e$ is meaningless, as this could cause e to fall off a profitable route without increasing their payoff. Consider the case $s_e < c_e$. (i) If with c_e they are on the minimum total cost route, then with s_e they will also be on it, without changing their payoff. (ii) If with c_e they were not on the minimum total cost route and with s_e they are, then the received $\widetilde{C}_{|\tilde{c}(e)=\infty} - \widetilde{C}_{|\tilde{c}(e)=0}$ transfer amount will be less than c_e. To see this, let $s'_e \in [s_e, c_e)$ be the cost at which e would be on a route with the same total cost as the minimum total cost route with c_e, i.e., with s'_e they would be on a minimum total cost route. But with s'_e, e's transfer would be exactly s'_e if the packet traveled along e. Clearly, with $s_e < s'_e$, they would still receive s'_e as transfer. Since $s'_e < c_e$, e would be at a loss. (iii) If with both c_e and s_e, e is not on the minimum total cost route, their profit is zero either way.

In summary, if $s_e \neq c_e$, e's profit cannot be greater than with $s_e = c_e$. Since for any e there exists a path avoiding e, if $s_e \neq c_e$, there exists some s_{-e} for which choosing $s_e = c_e$ is more profitable. □

The mechanism constructed for the problem is dominant for reporting true costs, which is very favorable for the internet, since due to the large and also variable size of the network, it cannot be expected that the participants know each other's costs. However, an unpleasant property of the mechanism is the need for transfers.

5.2 Braess Paradox

In a transportation network, the individual goal of the participants is to get from the starting point to the destination as quickly as possible. The socially optimal solution, in contrast, aims to minimize the average travel time of the participants. Consider the transportation network in Figure 5.2, which shows travel times on its edges depending on the proportion of traffic passing through.

Figure 5.2: Braess Paradox

For simplicity, we assume that due to the large number of participants, one participant's change of route has no effect on the time costs of the individual routes. The corresponding mathematical model has a continuum of participants, which approximates the case of very many participants. Since we talk about the proportion of participants using certain routes, we normalize the total traffic to 1.

If everyone optimizes individually in the left network of Figure 5.2, then in "equilibrium" half of the participants travel on each of the two possible routes from s to t, since then the travel time along both routes is one and a half hours, and it is not worth switching from one route to the other. No other equilibrium can exist, because then on one route one could get from s to t faster, and then any person using the slower route would have an individual incentive to switch to the other route. Still considering the left network, let α denote the fraction of traffic using the upper route. Then a fraction $1-\alpha$ of the traffic uses the lower route. The average time to get from s to t is

$$C(\alpha) = \alpha(1+\alpha) + (1-\alpha)(1+1-\alpha) = 2\alpha^2 - 2\alpha + 2,$$

where $\alpha \in [0,1]$. The solution of the conditional extremum problem minimizing the average travel time is $\alpha = 1/2$. So the individual equilibrium solution and the socially optimal solution coincide for the left network.

Examine the right network of Figure 5.2, which is an expansion of the left road network with a short and wide highway. Let α be the fraction of traffic using the upper route and β the fraction also using the new road section (henceforth the middle route). Then a fraction $1-\alpha-\beta$ of the participants uses the lower route. Our conditions are: $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta \leq 1$. The costs of the upper, middle, and lower routes are $\alpha+\beta+1$, $\alpha+\beta+1-\alpha = 1+\beta$ and $1+1-\alpha = 2-\alpha$, respectively. If $\alpha > 0$, then the upper route is always slower than the middle one. So in equilibrium $\alpha = 0$. Considering this, if $\beta < 1$, the middle route would be faster than the lower one, so in equilibrium $\beta = 1$. The equilibrium route time cost is 2 hours, which is surprising because expanding the left road network with one road section increased the participants' travel time by one third, i.e., network expansion can worsen the situation. This

phenomenon is called the *Braess paradox*. For completeness, determine the social optimum for the right road network:

$$C(\alpha,\beta)=\alpha(\alpha+\beta+1)+\beta(1+\beta)+(1-\alpha-\beta)(2-\alpha)=2\alpha^2+\beta^2+2\alpha\beta-2\alpha-\beta+2,$$

where $\alpha \geq 0$, $\beta \geq 0$ and $\alpha+\beta \leq 1$. The solution of the conditional extremum problem is $\alpha = 1/2$ and $\beta = 0$, which results in an average travel time of one and a half hours. It can be stated that for the right network, the route choice based on the participants' individual decisions increases the value of the coordinated socially optimal solution by one third. This is also said that the price of anarchy is 4/3.

5.2.1 Braess Paradox Detection

Let us define our problem formally.

Definition 5.4. $G=(V,E)$ is a directed graph with *source* $s \in V(G)$, *sink* $t \in V(G)$ $(s \neq t)$ and *cost function* $c : E \to \mathbb{R}_+^{\mathbb{R}_+}$, where the edge cost $c_e(x) = a_e x + b_e$ depends on the traffic x passing through edge $e \in E$ $(a_e, b_e \geq 0)$. (G, s, t, c) is a *network*.

Let $\mathcal{U} \neq \emptyset$ be the set of paths from s to t, $f : E \to \mathbb{R}_+$ the traffic on the edges and $f : \mathcal{U} \to \mathbb{R}_+$ the traffic along the paths, where $f_e = \sum_{e \in U} f_U$. Let the traffic be normalized to 1, i.e., $\sum_{U \in \mathcal{U}} f_U = 1$.

The cost of a path is $c_U(f) = \sum_{e \in U} c_e(f_e)$. The cost of traffic f is $C(f) = \sum_{e \in E} c_e(f_e) f_e$ or alternatively $C(f) = \sum_{U \in \mathcal{U}} c_U(f) f_U$.

Now we formally give the equilibrium concept already used in the previous subsection.

Definition 5.5. In the network (G, s, t, c), traffic f is a *Nash equilibrium* (or Wardrop equilibrium) if for any $U^* \in \mathcal{U}$ with $f_{U^*} > 0$ and any path $U \in \mathcal{U}$

$$c_{U^*}(f) \leq c_U(f).$$

Without proof, we state the claims regarding the existence and approximability of the equilibrium.[1]

Theorem 5.2. *The network (G, s, t, c) has a Nash equilibrium, moreover, all Nash equilibria have the same cost and if $a_e > 0$, they generate the same traffic on the edges. The Nash equilibrium can be approximated to any desired accuracy in polynomial time.*

Let $d(v)$ denote the length of the shortest path from s to $v \in V$ under traffic f, where the edge lengths are given by $c_e(f_e)$, and let $d(G, s, t, c) = C(f^*)$, where f^* is a Nash equilibrium.

The BPD (Braess Paradox Detection) problem seeks in a network (G, s, t, c) a subnetwork (H, s, t, c) that minimizes $d(H, s, t, c)$.

Theorem 5.3. *If P≠NP, then the approximation threshold of BPD is at least* 1/4*, i.e., at most* 4/3 *times the minimum can be achieved in polynomial time.*

Proof. We show that $4/3 - \varepsilon$ times the minimum cannot be achieved in polynomial time. We reason indirectly and use the NP-completeness of the 2DDP problem.

The input of the 2DDP problem is a directed graph $G = (V, E)$ and four distinct vertices s_1, s_2, t_1, t_2 of it. The question is whether there exist disjoint directed paths in G from s_1 to t_1 and from s_2 to t_2, where two directed paths are disjoint if they have no common vertices.

Take an instance $I = (G, s_1, t_1, s_2, t_2)$ of 2DDP, from which we obtain an instance $I' = (G', s, t, c)$ of our BPD problem as follows:

[1] For details, see Roughgarden (2005).

- to G' we add the two new vertices s, t and extend the graph G with the edges (s, s_1), (s, s_2), (t_1, t), (t_2, t),
- the costs of edges in G are 0, $c(s, s_2) = c(t_1, t) = x$ and $c(s, s_1) = c(t_2, t) = 1$.

G' can be constructed from G in polynomial time.

It suffices to show that

1. if $I \in 2DDP$, then G' has a subgraph H such that $d(H, s, t, c) = 3/2$, and
2. if $I \notin 2DDP$, then for any subgraph H $(s, t \in V(H))$ of G', $d(H, s, t, c) \geq 2$.

Proof of 1: Let U_1 and U_2 be disjoint paths from s_1 to t_1 and from s_2 to t_2, respectively. Choose the subgraph H by removing from G' all edges not on U_1 and U_2. Then sending $1/2$ traffic along each path gives $d(H, s, t, c) = 3/2$.

Proof of 2: If there is no path from s to t in H, then $d(H, s, t, c) = \infty$. So we may assume there is an s-t path in H. (i) If there is an s-t path in H that goes through s_2 and t_1, then in a Nash equilibrium traffic, the full traffic passes through edges ss_2 and t_1t, whose cost is 2. By earlier statements, any other Nash equilibrium traffic has cost 2. (ii) If there is no s-t path in H that goes through s_2 and t_1, then either there exists a path from s_i to t_i for some $i \in \{1, 2\}$, or every s-t path goes through s_1 and t_2. Note that there cannot simultaneously be paths from s_1 to t_1 and from s_2 to t_2, because then they would have to have a common vertex, which would imply the existence of an s-t path going through s_2 and t_1. Whichever of the three subcases holds, by routing the full traffic on the appropriate path, we obtain a Nash equilibrium with cost 2. □

Remark 5.1. It can also be proved that 4/3 times the minimum can be achieved in polynomial time.

Call a network (G, s, t, c) *paradox-free*, if for every subnetwork (H, s, t, c), $d(G, s, t, c) \leq d(H, s, t, c)$,

Remark 5.2. Deciding whether a network is paradox-free is NP-hard.

5.3 Combinatorial Auctions

An auction, briefly, is the sale or purchase of goods through bidding. Perhaps the most famous historical example is the auction of the Roman Empire in 193 AD. After the murder of Emperor Pertinax, the Praetorian Guard put the entire Roman Empire up for auction. Didius Julianus outbid Titus Flavius Sulpicianus with an offer of 25,000 sesterces per guard member versus 20,000 sesterces per member. Curse of the winners: Didius Julianus was emperor for less than three months. Practical applications include frequency auctions, public procurement, privatization, etc.

Combinatorial auctions are auctions in which bids can be made for multiple items simultaneously.

Example 5.1. The set of items is a left shoe and a right shoe (its pair). Possible bids: $v(\emptyset) = 0$, $v(\{\text{left shoe}\}) = 10$, $v(\{\text{right shoe}\}) = 10$ and $v(\{\text{left shoe}, \text{right shoe}\}) = 20000$, where v is the valuation function.

Better-known applications of combinatorial auctions:

- privatization of London bus routes,

- outsourcing transportation tasks of production or commercial companies,
- auctioning frequency licenses,
- auctioning computer resources, etc.

To discuss combinatorial auctions, we will need some notation. Let $N = \{1, \dots, n\}$ be the set of bidding players, $M = \{1, \dots, m\}$ the set of items to be auctioned, and $v_i : \mathcal{P}(M) \to \mathbb{R}_+$ the valuation of player $i \in N$. The utility of $i \in N$ if they receive the set $S \subset M$ of items is $u_i(S, p) = v_i(S) - p$, where $v_i(S)$ is i's true (only known to themselves) valuation and p is the amount paid for S. Let

$$\mathcal{O} = \{(S_1, \dots, S_n) \in \mathcal{P}^n(M) \mid \forall i \neq j \in N : S_i \cap S_j = \emptyset\}$$

be the set of *outcomes*, where S_i is the set of items received by player $i \in N$.

Definition 5.6. The outcome $S^* = (S_1, \dots, S_n) \in \mathcal{O}$ is *efficient* if

$$S^* \in \arg\max \left\{ \sum_{i=1}^{n} v_i(S_i) \mid S = (S_1, \dots, S_n) \in \mathcal{O} \right\}.$$

The brute force method to determine an efficient outcome looks at all possible allocations of M. These can be generated by partitioning the m elements and assigning the classes of the partition to players. We will see that the number of possible partitions alone is exponential in m.

Definition 5.7. The *Stirling number of the second kind* $\left\{ {n \atop k} \right\}$ is equal to the number of ways to partition an n-element set into k non-empty subsets.

Simple cases:

- $\left\{ {n \atop 0} \right\} = 0$, if $n \geq 1$. $\left\{ {0 \atop k} \right\} = 0$, if $k \geq 1$.
- If $n \geq 1$, then $\left\{ {n \atop 1} \right\} = 1$, $\left\{ {n \atop n} \right\} = 1$ and $\left\{ {n \atop n-1} \right\} = \binom{n}{2}$.
- $\left\{ {0 \atop 0} \right\} := 1$ and $\left\{ {n \atop k} \right\} := 0$, if $k < 0$ or $k > n$.

Theorem 5.4. *If $n \geq 1$, then*

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}.$$

Proof. Pick an element x of the n-element set X. We distinguish two cases:

- Partition $X \setminus \{x\}$ into k sets. This can be done in $\left\{ {n-1 \atop k} \right\}$ ways, then x can be added to any of the k classes.
- Partition $X \setminus \{x\}$ into $k-1$ sets. This can be done in $\left\{ {n-1 \atop k-1} \right\}$ ways. The singleton $\{x\}$ itself forms one class.

□

Remark 5.3. Due to the properties of binomial coefficients, $\sum_{k=0}^{n} \left\{ {n \atop k} \right\} \geq \sum_{k=0}^{n} \binom{n}{k} = 2^n$.

So the brute force method does not provide a result in time for many items.

5.3.1 Winner Determination Problem

The determination of the efficient allocation in combinatorial auctions can also be obtained as the solution of an integer programming problem. For this, we need a few more concepts.

Definition 5.8. $x_i : \mathcal{P}(M) \to \{0,1\}$ $(i \in N)$ is an *allocation*, where $x_i(S) = 1$ if and only if i receives S.

Definition 5.9. The allocation $(x_i)_{i\in N}$ is *feasible* if

- $\sum_{i\in N} \sum_{j\in S\subseteq M} x_i(S) \leq 1$ for every $j \in M$ and
- $\sum_{S\subseteq M} x_i(S) \leq 1$ for every $i \in N$.

The above definition, point (i), says that each item can be allocated at most once, while point (ii) says that each participant can receive at most one set of items. Denote by $\mathcal{X}$ the set of feasible allocations and $v_i(S)$ the bid of player $i \in N$ for the set $S \subseteq M$ of items.

Definition 5.10. The winner determination problem (WDP) is, given bids $(v_i)_{i\in N}$, to determine

$$(x_i^*)_{i\in N} \in \arg\max \left\{ \sum_{i\in N, S\subseteq M} v_i(S)x_i(S) \mid (x_i)_{i\in N} \in \mathcal{X} \right\}.$$

Remark 5.4. The length of the input of WDP, if given by listing all elementary bids $(S, v_i(S))$, is exponential in m. Otherwise, WDP is a special integer linear programming problem.

So the many possible elementary bids alone cause a problem. We will show that WDP is NP-hard even for elementary bids with simple structure and input length linear in m.

Definition 5.11. A player's bid is *single-minded* if $v : \mathcal{P}(M) \to \mathbb{R}_+$ is such that $\exists S^* \subseteq M$:

$$\forall S \supseteq S^* : v(S) = v(S^*) \text{ and } \forall S \not\supseteq S^* : v(S) = 0.$$

Remark 5.5. A single-minded player's bid can be compactly given by the elementary bid $(S^*, v(S^*))$.

Definition 5.12. The WDP* problem is the WDP problem for single-minded players. Its input is $(S_i^*, v_i^*)_{i\in N}$ and its output is the set $W \subseteq N$ of *winners* such that under the assumption $\forall i \neq j \in W : S_i^* \cap S_j^* = \emptyset$, $\sum_{i\in W} v_i^*$ is maximized.

WDP* is an optimization problem, whose corresponding WDP$_d^*$ decision problem only tells whether the auctioneer's total revenue $\sum_{i\in W} v_i^*$ reaches a given level k.

Theorem 5.5. *WDP$_d^*$ is NP-complete.*

Proof. WDP$_d^*$ is clearly in NP, since a revenue-maximizing allocation is a good hint.

We show that INDSET$\prec$WDP$_d^*$. For this, we first assign to an instance $G = (V, E)$ and $k \in \mathbb{Z}_+$ of INDSET an instance $(S_i^*, v_i^*)_{i\in N}$ of WDP$_d^*$. Let $M = E$, $N = V$, $S_i^* = \{e = uv \in E \mid i = u \vee i = v\}$ and $v_i^* = v_i(S_i^*) = 1$ for every $i \in N$.

Note that $S_i^* \cap S_j^* = \emptyset$ holds if and only if $\{i, j\} \subseteq V$ is an independent set. So there exists a set of winners $W \subseteq N$ with $\sum_{i\in W} v_i^* = |W| \geq k$ if and only if there is an independent set $X \subseteq V$ of size at least k. □

Remark 5.6. In a few special cases, an efficient algorithm for the WDP problem can be given.

- Bids can be made on sets of at most two elements.
- The items are linearly ordered and only connected sets (according to this order) can be bid on (e.g., docks).

For combinatorial auctions, further difficulties may be:

1. for the bidder, determining their own type, i.e., determining their bid $v : \mathcal{P}(M) \to \mathbb{R}_+$;
2. the communication requirement between bidders and the auctioneer, exponential in m;
3. achieving economic efficiency (implementation of the VCG mechanism discussed later may fail due to computational complexity).

5.3.2 Communication between Auctioneer and Bidders

Theorem 5.6. *There exists a bid that requires an exponential number of bits to be communicated, for any communication protocol.*

Proof. Assume $m = 2k$ is even, and consider the following bid:

$$v_i(S) = \begin{cases} 0, & \text{if } |S| < \frac{m}{2}; \\ 1, & \text{if } |S| > \frac{m}{2}; \\ 0 \text{ or } 1, & \text{if } |S| = \frac{m}{2}. \end{cases}$$

Since the number of sets with freely choosable values is $\binom{m}{m/2}$, the number of different bids of this type is $2^{\binom{m}{m/2}}$, whose communication requires $\binom{m}{m/2}$ bits.

Using Stirling's formula, which says

$$\lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{n!} = 1,$$

we have

$$\begin{aligned} \binom{m}{m/2} &= \binom{2k}{k} = \frac{(2k)!}{k! \cdot k!} \\ &\approx \frac{\sqrt{2\pi 2k}\left(\frac{2k}{e}\right)^{2k}}{\sqrt{2\pi k}\left(\frac{k}{e}\right)^k \cdot \sqrt{2\pi k}\left(\frac{k}{e}\right)^k} \\ &= \frac{1}{\sqrt{\pi k}} 2^{2k} = \frac{\sqrt{2}}{\sqrt{\pi m}} 2^m = O(2^m). \end{aligned}$$

□

The communication between the auctioneer and the bidder is made possible by so-called bidding languages. According to the previous theorem, for sufficiently large m, reporting all elementary bids cannot be expected, so work is being done on formulating languages with which bidders can "comfortably" specify relatively many bids.

A bid in the XOR language is of the form:

$$(S_1, v_1) \text{ XOR } (S_2, v_2) \text{ XOR } \ldots \text{ XOR } (S_k, v_k).$$

The above bid should be read as: the bidder wishes to have exactly one of the k elementary bids fulfilled. For example, they will not buy the set $S_1 \cup S_2$ at value $v_1 + v_2$, even if $S_1 \cap S_2 = \emptyset$.

A bid in the OR language is of the form:

$$(S_1, v_1) \text{ OR } (S_2, v_2) \text{ OR } \ldots \text{ OR } (S_k, v_k).$$

The above bid should be read as: the bidder is willing to have any of the k elementary bids on disjoint item sets fulfilled simultaneously. For example, they are willing to buy the set $S_1 \cup S_2$ at value $v_1 + v_2$ if $S_1 \cap S_2 = \emptyset$. The OR language assumes a kind of additivity for fulfilling elementary bids on disjoint item sets.

Theorem 5.7. *If we bid $v(S) = |S|$ for $S \subseteq M$, then this can be given in the OR language with m elementary bids, while in the XOR language it requires 2^m elementary bids.*

Proof. In the OR language, take the elementary bids $\forall x \in M : (\{x\}, 1)$. Then the bid $(S, |S|)$ can be given by

$$(\{x_1\}, 1) \text{ OR } (\{x_2\}, 1) \text{ OR } \ldots \text{ OR } (\{x_k\}, 1)$$

where $S = \{x_1, x_2, \ldots, x_k\}$.

Since in the XOR language at most one bid in the expression can be fulfilled, we must XOR all $(S, |S|)$-type elementary bids for all subsets S of M, of which there are 2^m. □

5.4 Social Choice

Given a finite set A of *alternatives* with at least three elements, which is the set of possible outcomes of collective decisions. Here one can think of presidential candidates, possible projects, or allocations. Let L be the set of linear orders (rankings) over the alternative set A and $N = \{1, \ldots, n\}$ the set of participants, henceforth the voters. We refer to L as the set of preference orders or rankings, and to L^n as the set of preference profiles (briefly profiles). Let $top(\succ)$ denote the most preferred alternative in the ranking $\succ \in L$. Knowing all voters' rankings, we should be able to rank the alternatives socially, or at least select one alternative, to which the following concepts are related.

Definition 5.13. An $F : L^n \to L$ is a *social choice rule.*

Definition 5.14. An $f : L^n \to A$ is a *social choice function.*

Note that the social choice rule F naturally induces a social choice function f defined as follows:

$$f(\succ_1, \ldots, \succ_n) = top(F(\succ_1, \ldots, \succ_n))$$

for any profile $(\succ_1, \ldots, \succ_n) \in L^n$. Since we require the social choice function to select one alternative and — as we will see — the voting procedures used to compute the result of the social choice function may result in ties, from now on we assume that in case of ties, the ranking of the participant with the smallest index is decisive.[2] Similarly, we resolve ties that arise in the computation of the social choice rule.

5.4.1 Some Notable Voting Procedures

Let's look at a three-alternative example:

Example 5.2. Consider the following three possible orders

[2] This is one of the many possible so-called *tie-breaking rules*. The most straightforward tie-breaking rules resolve ties using a predefined linear order over the alternative set or over the set of participants.

$\succ$	$\succ'$	$\succ''$
b	c	a
c	b	b
a	a	c

and let 9, 10, 11 persons have orders $\succ$, $\succ'$, $\succ''$, respectively.

We illustrate the three voting procedures we will describe on Example 5.2. Start with plurality voting, for which only the voters' most preferred alternatives need to be known. Therefore, in implementing plurality voting, each voter only needs to vote for one alternative. Alternative $x \in A$ is socially at least as good as alternative $y \in A$ if the number of voters who consider x their most preferred alternative is at least as large as the number of voters who consider y their most preferred alternative. In Example 5.2, a receives 11, b receives 9, and c receives 10 votes. Thus, the social ranking according to plurality voting is $a \succ c \succ b$, and the winner is a.

In the Borda count, each voter assigns a score to each alternative, which is the number of alternatives they prefer less than the given alternative. In the three-alternative case, the most preferred alternative gets 2 points, the second gets 1 point, and the least preferred gets 0 points. In Example 5.2, a receives $9{\cdot}0{+}10{\cdot}0{+}11{\cdot}2 = 22$ points, b receives $9{\cdot}2{+}10{\cdot}1{+}11{\cdot}1 = 39$ points, and c receives $9 \cdot 1 + 10 \cdot 2 + 11 \cdot 0 = 29$ points. Thus, the social ranking is $b \succ c \succ a$, and the winner is b.

In pairwise majority voting, for any two alternatives $x, y \in A$, the number of voters who prefer x to y is compared to the number of voters who prefer y to x. If the former is greater, then x is considered socially better than y; if the latter is greater, then y is considered socially better than x; and if they are equal, then the two alternatives are considered socially equally good. In Example 5.2, b is preferred to c by $9 + 11 = 20$ people, while c is preferred to b by 10 people, so b is socially better than c. Similarly, b is preferred to a by $9 + 10 = 19$ people, while a is preferred to b by 11 people, so b is socially better than a. Finally, c is preferred to a by $9 + 10 = 19$ people, while a is preferred to c by 11 people, so c is socially better than a. Thus, the social ranking is $b \succ c \succ a$, and the winner is b.

5.4.2 Desirable Properties of Social Choice Functions

Let's denote the social choice function by $f : L^n \to A$.

Definition 5.15. f is *Pareto efficient* if for any profile $(\succ_1, \ldots, \succ_n) \in L^n$ and any alternatives $x, y \in A$, if $x \succ_i y$ for all $i \in N$, then $y \neq f(\succ_1, \ldots, \succ_n)$.

The Pareto property states that if all participants prefer an alternative x to an alternative y, then y cannot be the winner.

Definition 5.16. f is *monotonic* if for any profile $(\succ_1, \ldots, \succ_n) \in L^n$, any $x = f(\succ_1, \ldots, \succ_n)$, and any profile $(\succ'_1, \ldots, \succ'_n) \in L^n$ such that for all $i \in N$ and $y \in A$, $x \succ_i y$ implies $x \succ'_i y$, we have $x = f(\succ'_1, \ldots, \succ'_n)$.

Monotonicity means that if an alternative x is the winner in a given profile, and then we modify the profile such that in every voter's ranking, x is moved forward relative to the other alternatives (i.e., the relative order of the other alternatives may change arbitrarily), then x remains the winner.

Definition 5.17. f is *dictatorial* if there exists a $d \in N$ such that for any profile $(\succ_1, \ldots, \succ_n) \in L^n$, $f(\succ_1, \ldots, \succ_n) = top(\succ_d)$.

Dictatorship means that the social choice function always selects the most preferred alternative of a fixed voter, regardless of the preferences of the other voters.

5.4.3 Arrow's Impossibility Theorem

In this subsection, we present one of the most important theorems of social choice theory, which essentially states that there is no "perfect" social choice rule.

Theorem 5.8 (Arrow's Impossibility Theorem)**.** *If $|A| \geq 3$, then any social choice rule $F : L^n \to L$ that satisfies*

(i) Pareto property*: for any profile $(\succ_1, \ldots, \succ_n) \in L^n$ and any alternatives $x, y \in A$, if $x \succ_i y$ for all $i \in N$, then $x \succ y$ in $F(\succ_1, \ldots, \succ_n)$,*

(ii) independence of irrelevant alternatives*: for any profile $(\succ_1, \ldots, \succ_n) \in L^n$ and any alternatives $x, y \in A$, if for all $i \in N$, $x \succ_i y$ if and only if $x \succ'_i y$, then $x \succ y$ in $F(\succ_1, \ldots, \succ_n)$ if and only if $x \succ y$ in $F(\succ'_1, \ldots, \succ'_n)$,*

is dictatorial*, i.e., there exists a $d \in N$ such that for any profile $(\succ_1, \ldots, \succ_n) \in L^n$ and any alternatives $x, y \in A$, $x \succ_d y$ implies $x \succ y$ in $F(\succ_1, \ldots, \succ_n)$.*

The proof of Arrow's theorem is lengthy and will not be presented here. For three short proofs see Geanakoplos (2005).

5.4.4 Gibbard–Satterthwaite Theorem

We now turn to the question of strategic manipulability of social choice functions. We will see that under certain conditions, no non-dictatorial social choice function is strategy-proof.

Definition 5.18. A social choice function $f : L^n \to A$ is *manipulable* by voter $i \in N$ at profile $(\succ_1, \ldots, \succ_n) \in L^n$ if there exists a ranking $\succ'_i \in L$ such that $f(\succ_1, \ldots, \succ_{i-1}, \succ'_i, \succ_{i+1}, \ldots, \succ_n) \succ_i f(\succ_1, \ldots, \succ_n)$.

In other words, voter i can achieve a better outcome (according to their true preferences) by misrepresenting their preferences.

Definition 5.19. A social choice function $f : L^n \to A$ is *strategy-proof* if it is not manipulable by any voter at any profile.

Theorem 5.9 (Gibbard–Satterthwaite Theorem)**.** *If $|A| \geq 3$ and $f : L^n \to A$ is a surjective and strategy-proof social choice function, then f is dictatorial.*

The Gibbard–Satterthwaite theorem is a cornerstone of social choice theory and mechanism design. It shows that under very general conditions, any non-dictatorial social choice function is vulnerable to strategic manipulation. The proof is also omitted here.

5.4.5 Computational Complexity of Strategic Manipulation

Given the negative results of the Gibbard–Satterthwaite theorem, one might hope that although manipulation is theoretically possible, it might be computationally difficult to find a successful manipulation. This could provide a practical barrier to manipulation. However, for many common voting rules, it turns out that finding a manipulation is computationally easy.

Theorem 5.10. *For the plurality rule, there is a polynomial-time algorithm that, given a profile and a manipulator's true preferences, finds a ranking that, if reported by the manipulator, leads to an outcome that is at least as good as the truthful outcome (and strictly better if possible).*

Proof. The manipulator simply votes for their most preferred alternative that has a chance of winning. More precisely, let a be the manipulator's most preferred alternative. If by voting for a, a becomes the winner or ties for first place (and wins after tie-breaking), then the manipulator should vote for a. Otherwise, the manipulator should vote for the alternative that is currently leading (or tied for the lead) and is most preferred by the manipulator among such alternatives. This can be computed in polynomial time. □

Similar results hold for many other voting rules, such as the Borda count and pairwise majority voting. However, for some voting rules, the manipulation problem is NP-hard.

Theorem 5.11. *For the second-order Copeland rule, it is NP-hard to decide whether a given voter can manipulate the election.*

The Copeland rule is defined as follows: each alternative receives one point for every other alternative it defeats in a pairwise majority comparison (and half a point for a tie). The second-order Copeland rule breaks ties by the sum of the Copeland scores of the alternatives it defeats. The proof of NP-hardness for this rule is involved and uses a reduction from a known NP-complete problem.

These results suggest that computational complexity might be used as a barrier to manipulation in some cases, but for many common rules, manipulation remains computationally easy.

5.5 Fair Division Games

The problem of fair division is probably as old as humanity. According to Brams and Taylor (1996), the first documented fair division procedure can be found in Hesiod's (ca. 7th century BC) epic "Theogonia". In it, Prometheus divided the meat to be consumed jointly with Zeus by first cutting it into two parts, then Zeus selected one of the two parts for himself, leaving the other to Prometheus. This procedure is known as "divide and choose". The procedure ensures that regardless of the other party's actions, both parties can receive half of the quantity to be divided. For this, the divider must divide the quantity into two equal parts according to his own valuation, and then the chooser selects the not smaller part according to his own valuation.

The "divide and choose" procedure is still used today. Notably, the International Seabed Exploitation Agreement, signed by 159 states and entering into force on November 16, 1994. According to the agreement, if a company from a developed country wants to exploit the seabed somewhere, it must first divide the area to be exploited into two parts. Then, an international organization representing the interests of developing countries selects the part for the developing countries that the company from the developed country cannot touch, thus ensuring the possibility of later exploitation for developing countries.

A good example of multi-player fair division problems is the issue of land division. How do three siblings divide inherited land "fairly" among themselves? The three procedures to be presented are suitable for solving such problems.

5.5.1 Fair Division Problem

We will now deal exclusively with problems where the object to be divided is arbitrarily divisible. Let us precisely state what kind of fair division problems we will examine.

Definition 5.20. The mathematical structure $\left(N, \Omega, \mathcal{A}, (\mu_i)_{i \in N}\right)$ is called a *fair division problem* if

- N is a finite set of participants in the division;
- Ω denotes the entire object (territory) to be divided, which we will henceforth call the *cake*;
- $\mathcal{A}$ is the set of possible cake slices, which satisfies the following conditions
  - (i) $\Omega \in \mathcal{A}$, i.e., the whole cake is a possible slice;
  - (ii) if $A \in \mathcal{A}$ and $B \in \mathcal{A}$ are two possible slices, then $A \cup B \in \mathcal{A}$;
  - (iii) if $A \in \mathcal{A}$, then $A^c \in \mathcal{A}$;
- μ_i is the cake slice valuation function of person $i \in N$, for which
  - (i) $\mu_i : \mathcal{A} \to [0, \infty)$, i.e., i characterizes every possible slice with a non-negative number;
  - (ii) if $A, B \in \mathcal{A}$ and $A \cap B = \emptyset$, then $\mu_i(A \cup B) = \mu_i(A) + \mu_i(B)$ (additivity);
  - (iii) $\mu_i(\Omega) = 1$ (normalization);
  - (iv) for every slice $A \in \mathcal{A}$ there is a subslice $B \in \mathcal{A}$, $B \subseteq A$ with $\mu_i(B) = \lambda \mu_i(A)$ for any $\lambda \in [0, 1]$ (continuity of divisibility);
- only participant $i \in N$ knows his valuation function μ_i.

Remark 5.7. A collection of sets that satisfies the three conditions imposed on the set system $\mathcal{A}$ in the fair division problem is called an *algebra*.

Remark 5.8. The additivity of μ_i implies finite additivity of μ_i. A non-negative and additive set function defined on an algebra is called a *finitely additive measure*.

Remark 5.9. The continuity of divisibility assumption implies that for any element $\omega \in \Omega$ that is also a possible slice (i.e., $\{\omega\} \in \mathcal{A}$), necessarily $\mu_i(\{\omega\}) = 0$.

For simplicity, in our fair division procedures we identify the cake with the interval $[0, 1]$. In this case, the cut points $x, y \in [0, 1]$ indicate the positions of parallel cuts to be made on the cake, or on a circular cake they can indicate radial cuts. If $\Omega = [0, 1]$, then $\mathcal{A}$ is the collection of sets that can be obtained as unions of finitely many subintervals of $[0, 1]$. Furthermore, under continuity of divisibility, the function $F_i(x) := \mu_i([0, x))$ is continuous on $[0, 1]$, with the help of which the value of a slice $[x, y]$ can be calculated as $\mu_i([x, y]) = \mu_i([x, y)) = \mu_i([0, y)) - \mu_i([0, x)) = F_i(y) - F_i(x)$.

A *fair division procedure* can instruct the participants according to specified rules to perform cuts and evaluate cake slices. The former means specifying a set $A \in \mathcal{A}$, while the latter aims to learn the valuation $\mu_i(A)$ of the asked participant i. Based on the cuts made and answers given, the procedure calls on participants to perform further cuts and evaluations as long as there is still cake to be divided. A procedure is finite if after performing the mentioned steps a finite number of times, it divides the cake for any fair division problem. A fair division procedure can be used to determine a *division* $(A_i)_{i \in N}$ (where $A_i \in \mathcal{A}$) of the cake, which is a partition of Ω. Given a fair division problem, a given fair division procedure defines a game in which the participants' (players') actions are the cuts and evaluations, and the outcome is a partition of the cake with the associated valuations.

We can characterize fair division procedures using several criteria. Since the applied procedure does not know the cake slice valuation functions of the individual participants, we expect the procedures to *encourage truth-telling*, i.e., when evaluating a cake slice, participants give their true valuations, and when requiring a cut of size $\alpha \in (0, 1)$, the instructed participant

indeed cuts off a slice of size α according to his own valuation. A procedure can achieve truth-telling by ensuring that any participant can only harm himself by "lying". A very simple truth-eliciting procedure is the "dictatorial" one, which gives the whole cake to a predetermined person. In contrast to the "dictatorial" procedure, we are looking for procedures that also satisfy some fairness criterion. The simplest fairness criterion is *proportionality*, which requires that each participant can guarantee for himself (independently of the actions of other participants) a proportional part of the cake (with n participants, a part worth $1/n$ according to his own valuation). A stronger fairness criterion is *envy-freeness*, which requires that each participant receives one of the most valuable parts of the cake according to his own valuation ($\forall i, j \in N : \mu_i(A_i) \geq \mu_i(A_j)$, where $(A_i)_{i \in N}$ is a division of the cake).

5.5.2 Banach–Knaster Procedure

Steinhaus (1948) gave the first proportional fair division procedure for three persons. However, his procedure cannot be generalized to arbitrary n. The first n-person proportional fair division procedure is associated with the names of Steinhaus's students, Banach and Knaster. For $n = 2$ they also use the "divide and choose" procedure. If $n > 2$, then as a first step, 1 cuts off a slice from the cake, then passes the cut slice to 2. 2 looks at the slice, and if he thinks he can adjust the slice with one cut. Depending on his decision, he passes the original unadjusted or the adjusted slice to 3. The procedure repeats like this until a slice reaches n. Then n can decide whether to accept the slice or reject it. If n accepts, then n leaves with the slice and $1, \ldots, n-1$ divide all remaining parts. If n rejects the slice, then the slice must be taken by the one who cut last. The last cutter leaves, while the other $n-1$ persons divide all remaining cake slices among themselves.

Example 5.3. Let $\Omega = [0,1]$, $N = \{1,2,3\}$, $F_1(x) = x$, $F_2(x) = x^2$ and $F_3(x) = x^4$ ($x \in [0,1]$). Determine a proportional division using the Banach–Knaster procedure!

For simplicity, assume that persons cut from the left. First, 1 cuts at $\frac{1}{3}$ and gives this slice $[0, \frac{1}{3})$ to 2. For 2, this slice is worth only $F\left(\frac{1}{3}\right) = \frac{1}{9}$, so he passes it to 3 without adjustment, for whom this slice is worth even less. Thus, 1 must take the slice $[0, \frac{1}{3})$. 2 and 3 divide the

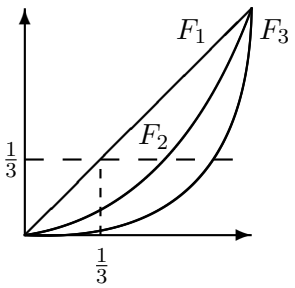


Figure 5.3: Banach–Knaster procedure

remaining slice $[\frac{1}{3}, 1]$. If 2 divides, then he divides the slice worth $\frac{8}{9}$ for him into two slices worth $\frac{4}{9}$ each. For this, he solves the equation $x^2 - \frac{1}{9} = \frac{4}{9}$, whence $x = \frac{\sqrt{5}}{3}$ is where he will cut. 3 chooses the $[x, 1]$ slice from the two, since $F_3(1) - F_3(x) > F_3(x) - F_3(\frac{1}{3})$. Thus, 2 gets the remaining slice $[\frac{1}{3}, x)$.

First, let's see how many cuts the Banach–Knaster procedure results in in the worst case. In the first round, at most $n-1$ persons cut. After that, in the second round, only $n-1$ persons divide the remaining part, which in the worst case can lead to $n-2$ cuts. Now only $n-2$ persons participate in dividing the remaining parts. Continuing this reasoning, we get that $T^{\text{cut}}_{\max}(n) = (n-1) + (n-2) + \ldots + 2 + 1 = (n-1)n/2$, i.e., $T^{\text{cut}}_{\max} = \Theta(n^2)$.

Now we just need to consider whether the Banach–Knaster procedure indeed guarantees a proportional share for everyone. For $n = 1$ and $n = 2$, this is obviously true. Assume that the Banach–Knaster procedure ensures for $n - 1$ persons — independently of the others' actions — a proportional share from the cake. Examining the n-person case, as an initial step 1 must cut off at least an n-th part of the cake according to his own valuation, because if he cuts off a part smaller than $1/n$, he risks having to take the slice smaller than $1/n$, since he might end up in the role of the last cutter. And if 1 cuts off a part larger than $1/n$, then it is possible that the last cutter or n takes away a slice that is larger than $1/n$ according to 1's valuation. In that case, however, 1 will participate in an $n - 1$ person division in which only less than $\frac{1}{n-1}\frac{n-1}{n} = \frac{1}{n}$ fraction is guaranteed for him. Repeating the argument for 1 for the other participants, they also must cut off a slice worth $1/n$ according to their own valuation from the piece that comes to them, if it is worth more than $1/n$ to them. After the first round, one person leaves with at least $1/n$, and a part remains that is valued by the others individually at least $(n-1)/n$, which they then divide among themselves using an $n-1$ person procedure.

5.5.3 Even–Paz Procedure

The Even–Paz procedure (1984) uses the divide and conquer principle. For $n = 2$, apply the "divide and choose" procedure. For $n = 3$, use the Banach–Knaster procedure.

Consider the $n = 4$ case. Then we ask persons 1, 2, and 3 to divide the cake $\Omega = [0, 1]$ into two equal parts with parallel cuts. Denote by $x_1, x_2, x_3 \in (0, 1)$ the cut points of persons 1, 2, and 3 respectively. Without loss of generality, assume $x_1 \leq x_2 \leq x_3$. If $\mu_4([0, x_2]) \geq \mu_4([x_2, 1])$, then let 4 divide with 1 on the part $[0, x_2)$, and 2 and 3 on the part $[x_2, 1]$. If $\mu_4([0, x_2)) < \mu_4([x_2, 1])$, then let 4 divide with 3 on the part $[x_2, 1]$, and 1 and 2 on the part $[0, x_2)$. 2 in this situation is the *median cutter*. If everyone is truthful, it is easy to see that a quarter of the cake is guaranteed for all four persons. 1 is truthful, because only by cutting at a point greater than x_2 would the outcome of the division change. However, then if 4 chose the part to the left of the median cut, 1 would have to divide with 3 on a part that he values less than half. Similarly, it can be reasoned that 2 and 3 are also truthful. 4 is obviously truthful. So indeed we have given a proportional procedure for $n = 4$. Determine the number of required cuts: $T^{\text{cut}}_{\max}(4) = 3 + 1 + 1 = 5$.

Example 5.4. Let $\Omega = [0, 1]$, $N = \{1, 2, 3\}$, $F_1(x) = x$, $F_2(x) = x^2$, $F_3(x) = \sqrt{x}$ and $F_4(x) = x$ $(x \in [0, 1])$. Determine a proportional division using the Even–Paz procedure!

According to the procedure, 1, 2, and 3 cut at $x_1 = \frac{1}{2}$, $x_2 = \frac{1}{4}$, and $x_3 = \frac{1}{\sqrt{2}}$ respectively (see 5.4). So 1 is the median cutter. Since for 4 the slices $[0, x_1)$ and $[x_1, 1]$ are of equal value, he can choose either slice. For simpler calculation, assume that 4 chooses the slice $[0, x_1)$. Then on $[0, \frac{1}{2})$, 3 and 4, while on $[\frac{1}{2}, 1]$, 1 and 2 divide. Let 1 and 4 be the dividers. Then we

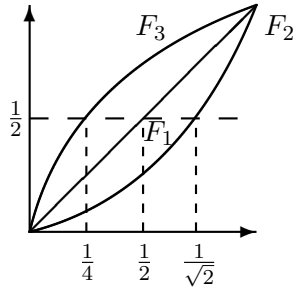


Figure 5.4: Even–Paz procedure

get the slices $[0, \frac{1}{4})$, $[\frac{1}{4}, \frac{1}{2})$, $[\frac{1}{2}, \frac{3}{4})$ and $[\frac{3}{4}, 1]$, from which 2 chooses the last and 3 the first. 4 gets the second and 1 the third slice.

Now turn to the $n = 5$ case. Ask persons 1, 2, 3, and 4 to divide the cake $\Omega = [0, 1]$ into a $2 : 3$ ratio with parallel cuts. Denote by $x_1, x_2, x_3, x_4 \in (0, 1)$ the cut points of persons 1, 2, 3, and 4 respectively. Again without loss of generality, assume $x_1 \leq x_2 \leq x_3 \leq x_4$. If $\mu_5([0, x_2)) \geq 2/5$, then let 5 divide with 1 on the part $[0, x_2]$, and 2, 3, and 4 on the part $[x_2, 1]$. If $\mu_5([0, x_2)) < 2/5$, then let 5 divide with 3 and 4 on the part $[x_2, 1]$, and 1 and 2 on the part $[0, x_2)$. Similar considerations as for $n = 4$ can verify that if everyone is truthful, then a fifth of the cake is guaranteed for all five persons. And the individual participants are truthful because by lying they can only harm themselves by being forced into a less favorable division. The number of required cuts is now $T^{\text{cut}}_{\max}(5) = 4 + 3 + 1 = 8$, since first four persons cut the cake in half, then three persons divide on one half and two on the other half.

In the $n = 2k$ even case, the participants can be grouped into two k-person groups similar to the $n = 4$ case, each dividing on a half of the cake that each group member values at least half. For this, first ask the first $2k - 1$ persons to halve the cake with parallel cuts. Denote in the interval $[0, 1]$ the corresponding cut points by $x_1, x_2, \ldots, x_{2k-1}$. Without loss of generality, assume $x_1 \leq x_2 \leq \ldots \leq x_{2k-1}$. Then the $2k$-th person decides whether $\mu_{2k}([0, x_k)) \geq 1/2$. If yes, then persons $1, 2, \ldots, k - 1, 2k$ divide the part $[0, x_k)$ with the already defined k-person procedure, and persons $k, k+1, \ldots, 2k-1$ also divide with the k-person procedure on the part $[x_k, 1]$. If not, then persons $1, 2, \ldots, k$ divide the part $[0, x_k)$ with the k-person procedure and persons $k + 1, \ldots, 2k$ divide on $[x_k, 1]$ with the k-person procedure. It can be reasoned that now it is also in everyone's interest to cut at their own halving point. The operation count can be given by the recurrence $T^{\text{cut}}_{\max}(2k) = 2k - 1 + 2T(k)$.

Now turn to the $n = 2k+1$ odd participant case, which is analogous to the five-participant case. The first $2k$ persons cut the cake with parallel cuts into $k : k + 1$ ratio parts, then the $2k + 1$-th person checks whether $\mu_{2k+1}([0, x_k)) \geq k/(2k + 1)$. If yes, then persons $1, 2, \ldots, k - 1, 2k + 1$ divide the part $[0, x_k]$ with the k-person procedure, and persons $k, k + 1, \ldots, 2k$ divide the part $[x_k, 1]$ with the $k + 1$-person procedure. If not, then persons $1, 2, \ldots, k$ divide on $[0, x_k)$ with the k-person procedure, and persons $k + 1, \ldots, 2k + 1$ divide on $[x_k, 1]$ with the $k + 1$-person procedure. It can be proved that giving a false $k : k + 1$ ratio cut point risks a participant ending up in a division that no longer guarantees a proportional share. The operation count can now be given by the recurrence $T^{\text{cut}}_{\max}(2k + 1) = 2k + T(k) + T(k + 1)$.

To determine the order of magnitude of the operation count, start with the $n = 2^m$ case:

$$\begin{aligned} T^{\text{cut}}_{\max}(n) &= n - 1 + 2T\left(\frac{n}{2}\right) = n - 1 + n - 2 + 4T\left(\frac{n}{4}\right) = \ldots = \\ &= n - 2^0 + n - 2^1 + n - 2^2 + \ldots + n - 2^{m-2} + 2^{m-1}T\left(\frac{n}{2^{m-1}}\right) = \\ &= (m - 1)\, n - \left(2^{m-1} - 1\right) + 2^{m-1} = n \log_2 n - n + 1, \end{aligned}$$

from which we conjecture that the order of the operation count is $n \log_2 n$. To verify our conjecture, look at the cases $2^{m-1} < n < 2^m$. For such n, it can be checked that $T\left(2^{m-1}\right) < T(n) < T\left(2^m\right)$ and therefore

$$\frac{2^{m-1}\,(m-2)+1}{2^m m} = \frac{T\left(2^{m-1}\right)}{2^m m} < \frac{T(n)}{n \log_2 n} < \frac{T\left(2^m\right)}{2^{m-1}(m-1)} = \frac{2^m\,(m-1)+1}{2^{m-1}\,(m-1)}. \tag{5.1}$$

If n and m tend to infinity such that $2^{m-1} < n \leq 2^m$ holds, then based on (5.1)

$$\frac{1}{2} \leq \liminf_{n\to\infty} \frac{T(n)}{n \log_2 n} \leq \limsup_{n\to\infty} \frac{T(n)}{n \log_2 n} \leq 2.$$

Consequently, $T_{\max}^{\text{cut}} = \Theta(n \log_2 n)$, i.e., the Even–Paz procedure requires cuts of order $n \log_2 n$.

The Even–Paz procedure is the known proportional fair division procedure requiring the fewest cuts in order of magnitude. Sgall and Woeginger (2007) proved that among proportional fair division procedures where individual participants receive adjacent intervals,[3] there is no procedure requiring fewer cuts in order of magnitude than $n \log_2 n$. However, it is an open question whether, if participants can receive non-adjacent slices, a proportional procedure can be constructed that requires fewer cuts in order of magnitude. For more details on fair division procedures, see Robertson and Webb (1998).

5.5.4 Selfridge–Conway Procedure

The Selfridge–Conway procedure (see e.g. Brams and Taylor, 1996) provides an envy-free division for three-person fair division problems.

Ask 1 to divide the cake into three equal parts. Denote the resulting three slices I, J, and K such that $\mu_2(I) \geq \mu_2(J) \geq \mu_2(K)$, i.e., for 2, I is the most valuable, J the second most valuable, and K the third most valuable slice. Ask 2 to trim I to equal value with J. Denote the trimmed slice by L and the trimmed-off piece by $M = I \setminus L$.[4] Now let the three persons choose from the slices J, K, and L in the order 3, 2, 1, where if 3 does not choose L, then 2 is obliged to take L.[5] We distinguish two cases.

(a) If 2 took L, then ask 3 to divide M into three equal parts. Then let the three persons choose from the three slices in the order 2, 1, 3.

(b) If 3 took L, then ask 2 to divide M into three parts. Then 3, 1, and 2 choose from the three slices in that order.

We will show that the Selfridge–Conway procedure provides an envy-free division. Consider case (a). 1 cannot envy 2, because 2 only got a part of I. 1 cannot envy 3 either, since in the first round he got a slice of equal value to 3 and in the second round he chooses before 3. 2 cannot envy anyone, since in the first round he took one of the largest slices and in the second round he could choose first. 3 cannot envy anyone, because in the first round he chose first and in the second round he got one of the three equal slices. Case (b) can be proved similarly. Furthermore, it is not hard to see that the procedure encourages truth-telling.

The Selfridge–Conway procedure is a finite envy-free procedure that cannot be extended to more participants. Although envy-free finite procedures for more participants are known, these are *not bounded* (see for example Robertson and Webb, 1998), meaning that for a fixed number of participants, fair division problems requiring arbitrarily many cuts can be constructed. Stromquist (2008) proved that for at least three-person fair division problems there cannot exist a finite (even unbounded) envy-free division procedure that gives each participant adjacent slices. Procaccia (2009) proved that the difficulty of finding an envy-free division grows asymptotically at least quadratically in the number of participants. Based on this, it can be stated that finding an envy-free division is certainly a harder task in terms of complexity theory than finding a proportional division. Nevertheless, Stromquist (1980) showed that for n participants there always exists an envy-free division requiring $n-1$ cuts. Since his proof is not constructive, many unanswered questions remain in the area of computability of envy-free divisions.

[3] i.e., participants receiving multiple slices all get slices adjacent to each other.
[4] $M = \emptyset$ if for B, I and J are of equal value.
[5] If $M = \emptyset$, the procedure already ends here.

5.5.5 Brams–Taylor–Zwicker Procedure

The Brams–Taylor–Zwicker procedure (1997) is a four-person envy-free bounded moving-knife procedure, which builds on Austin's (1982) two-person procedure that gives exactly 50% to both parties, and on ideas inherent in the Selfridge–Conway procedure.

In Austin's procedure, one participant, say 1, moves two parallel knives continuously from left to right such that the area between the two knives is always worth half of the cake to him. Initially, the left knife is at the left edge of the cake, while the right knife is above 1's midpoint.[6] At the moment when 2 stops the two knives, 1 makes the two cuts at the positions indicated by the knives. Then by lot (e.g., coin toss) it is decided which participant gets the middle slice. Then the other participant gets the two outer slices. If 2 did not stop the knives before the right knife reached the right edge of the cake, then 1 would cut the cake in half with the left knife, and then the slices' fate would be decided by coin toss. Obviously, 1 must always ensure that the slice between the two knives is worth half of the cake to him. 2 should stop the knives when his valuation coincides with 1's. It is also questionable whether there is such a moment? If 2's valuation of the two half cakes initially coincides with 1's valuation, then we have already found such a moment. Otherwise, the slice between the two knives is (i) more valuable, or (ii) less valuable than the other slice to 2. If 2 let the right knife go to the right edge of the cake, then the slice between the two knives would become (i) less valuable, or (ii) more valuable than the other slice. By continuity considerations, there is an intermediate moment when the slice between the two knives is of equal value to the two outer slices for 2.

Now turn to the Brams–Taylor–Zwicker procedure. First, by applying Austin's procedure twice, 1 and 2 divide the cake into fourths. For the resulting X_1, X_2, X_3, and X_4 parts we then have $\mu_1(X_i) = \mu_2(X_j)$ for all $i, j = 1, 2, 3, 4$. Assume that $\mu_3(X_1) \geq \mu_3(X_2) \geq \mu_3(X_3) \geq \mu_3(X_4)$. Let 3 divide the X_1 part into Y_1 and Y_2 parts such that $\mu_3(Y_1) = \mu_3(X_2)$. Let them choose from the parts Y_1, X_2, X_3, and X_4 in the order $4, 3, 2, 1$, with the stipulation that if 4 does not choose Y_1, then 3 is obliged to take Y_1. After the first (selection) round, no one envies anyone, because 4 is the first chooser, 3 took one of the two largest parts according to his valuation, and $1, 2$ got two $1/4$ valued parts.

Consider the case when 4 takes Y_1. Then let 2 and 3 divide the Y_2 part by Austin's procedure into four equal parts. For the resulting Z_1, Z_2, Z_3, and Z_4 parts we have $Y_2 = \cup_{i=1}^4 Z_i$, $\mu_2(Z_i) = \mu_2(Y_2)/4$ and $\mu_3(Z_i) = \mu_3(Y_2)/4$ for all $i = 1, 2, 3, 4$. Then the participants choose from the parts Z_1, Z_2, Z_3, Z_4 in the order $4, 1, 3, 2$. Any of them could only envy someone who chose before them, and 2 and 3 envy no one, since they divided Y_2 into four equal parts. Finally, 1 does not envy 4, because 4 can get at most a part worth X_1.

The other case, when 3 takes Y_1, is similar. Then let 2 and 4 divide Y_2 by Austin's procedure into four equal parts. The participants now choose in the order $3, 1, 4, 2$. Any of them could only envy someone who chose before them, and 2 and 4 envy no one, since they divided Y_2 into four equal parts. Finally, 1 does not envy 3, because 3 gets a subslice of X_1.

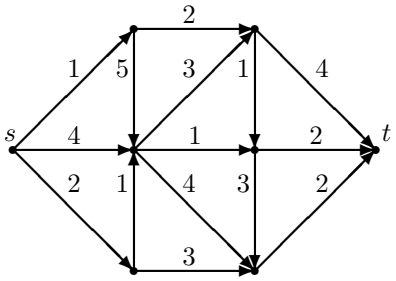
About a decade after Brams, Taylor, and Zwicker (1997), Saberi and Wang (2009) gave a five-person envy-free bounded moving-knife procedure. Note that for more than five persons, only $\varepsilon > 0$ approximate or unbounded envy-free procedures are known (see for example Brams and Taylor, 1996 or Robertson and Webb, 1998).

5.6 Exercises

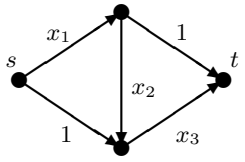
Exercise 5.1. In the following network, the edges are owned by providers, and the packet forwarding costs are shown on the edges. According to the learned mechanism, what transfers

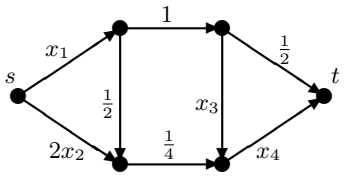
[6] For simplicity, assume there are no cake slices of zero value for 1 and 2.

are given to the various participants if the costs on the edges are the declared costs?



Exercise 5.2. Determine the socially optimal and Nash equilibrium flows for the following two networks!





Exercise 5.3. The cake slice valuation distribution functions of persons 1 and 2 are $F_1(x) = x^2$ and $F_2(x) = \sqrt{x}$ $(x \in [0,1])$. Determine a proportional division using the "one divides, the other chooses" procedure!

Exercise 5.4. The cake slice valuation distribution functions of persons $1, 2$, and 3 are

$$F_1(x) = \begin{cases} 2x, & \text{if } x \in [0, \frac{1}{2}], \\ 1, & \text{if } x \in (\frac{1}{2}, 1] \end{cases}$$

$F_2(x) = x^2$ and $F_3(x) = \sqrt{x}$ $(x \in [0,1])$. Determine a proportional division using the Banach–Knaster procedure!

Exercise 5.5. On the cake $\Omega = [0,1]$, persons $N = \{1,2,3,4\}$ divide. The persons' cake slice valuation distribution functions are $F_1(x) = \min\{2x, 1\}$, $F_2(x) = \min\{\frac{3}{2}x, \frac{1}{2}(x+1)\}$, $F_3(x) = x$ and $F_4(x) = \max\left\{\frac{x}{2}, \frac{3}{2}x - \frac{1}{2}\right\}$ $(x \in [0,1])$. Determine the proportional division using the Banach–Knaster procedure!

Exercise 5.6. On the cake $\Omega = [0,1]$, persons $N = \{1,2,3,4\}$ divide. The persons' cake slice valuation distribution functions are $F_1(x) = x$, $F_2(x) = x^2$, $F_3(x) = \log_2(1+x)$ and $F_4(x) = x^3$ $(x \in [0,1])$. Determine the proportional division using the Even–Paz procedure!

Exercise 5.7. Let $F_1(x) = \sqrt[7]{x}$, $F_2(x) = \sqrt{x}$, $F_3(x) = x$, $F_4(x) = x^3$ and $F_5(x) = x$ be the cake slice valuation distribution functions of persons $1, 2, 3, 4, 5$ $(x \in [0,1])$. Determine the proportional division using the Even–Paz procedure!

Exercise 5.8. On the cake $\Omega = [0,1]$, persons $N = \{1,2,3\}$ divide. The persons' cake slice valuation distribution functions are $F_1(x) = \max\left\{\frac{x}{2}, \frac{3}{2}x - \frac{1}{2}\right\}$, $F_2(x) = x^2$ and $F_3(x) = \sqrt{x}$ $(x \in [0,1])$. Determine the envy-free division using the Selfridge–Conway procedure!

Bibliography

[1] Arrow, K.J.: *Social choice and individual values.* Wiley, NewYork, 1951.

[2] Austin, A.K.: Sharing a Cake. *Mathematical Gazette*, **66**, 212–215, 1982.

[3] Brams, S.J. – Taylor, A.D.: *Fair Division: From Cake Cutting to Dispute Resolution.* Cambridge University Press, Cambridge UK, 1996.

[4] Brams, S.J. – Taylor, A.D. – Zwicker, W.S.: A moving-knife solution to the four-person envy-free cake division problem, *Proceedings of the American Mathematical Society*, **125**, 547–554, 1997.

[5] Church, A.: An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, **58**, 345–363, 1936.

[6] Even, S. – Paz, A.: A Note on Cake Cutting. *Discrete Applied Mathematics*, **7**, 285–296, 1984.

[7] Ferenczi M.: *Mathematical Logic* (in Hungarian), Műszaki Könyvkiadó, 2002.

[8] Forgó F. – Pintér M. – Simonovits A. – Solymosi T.: *Game Theory* (electronic notes, in Hungarian). 2005.[7]

[9] Geanakoplos, J.: Three brief proofs of Arrow' impossibility theorem. *Economic Theory*, **26**, 211–215, 2005.

[10] Gibbard, A.: Manipulation of voting schemes: a general result. *Econometrica*, **41**, 587–601, 1973.

[11] Mala J.: Arrow-type impossibility theorems (in Hungarian). *Szigma*, **38**, 77–110, 2007.

[12] Moulin, H.: On strategy-proofness and single peakedness. *Public Choice*, **68**, 437–455, 1980.

[13] Nisan, N.: Introduction to mechanism design (for computer scientists). In: Nisan, N. – Roughgarden, T. – Tardos E. – Vazirani, V. (eds.) *Algorithmic Game Theory*, 209–241, Cambridge University Press, Cambridge, 2007.

[14] Nisan, N. – Ronen, A.: Algorithmic mechanism design. *Games and Economic Behavior*, **35**, 197–211, 2001.

[15] Papadimitriou, C.H.: *Computational Complexity.* Novadat, Budapest, 1999 (Hungarian translation).

[7] http://web.uni-corvinus.hu/~pmiklos/Works/PDF/forgo_jatekelmelet.pdf (accessed: May 11, 2013.)

[16] Procaccia, A.D.: Thou shalt covet thy neighbor's cake. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 239–244, 2009.

[17] Robertson, J. – Webb, W.: *Cake-cutting algorithms: be fair if you can.* A K Peters, Ltd., Natick, 1998.

[18] Rónyai L. – Ivanyos G. – Szabó R.: *Algorithms* (in Hungarian). TypoTex, Budapest, 1999.

[19] Roughgarden, T.: *Selfish Routing and the Price of Anarchy.* MIT Press, 2005.

[20] Saberi, A. – Wang, Y.: Cutting a cake for five people. *Lecture Notes in Computer Science*, **5564**, 292–300, 2009.

[21] Satterthwaite, M.A.: Strategy-proofness and Arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, **10**, 187–217, 1975.

[22] Sgall, J. – Woeginger, G.J: On the complexity of cake-cutting. *Discrete Optimization*, **2**, 213–220, 2007.

[23] Steinhaus, H.: The problem of fair division. *Econometrica*, **16**, 101–104, 1948.

[24] Stromquist, W.: How to cut a cake fairly. *American Mathematical Monthly*, **87**, 640–644, 1980 and Addendum, **88**, 613–614, 1981.

[25] Stromquist, W.: Envy-free cake divisions cannot be found by finite protocols. *The Electronic Journal of Combinatorics*, **15**, research paper no. R11, 2008.

[26] Summerfield, M.: *Python 3 Programming.* Kiskapu Kft., Budapest, 2009 (Hungarian translation).

[27] Tasnádi A.: *Computer Science for Business Informatics Students* (in Hungarian). AULA Kiadó, Budapest, 2006.