Course number 02249

Lecture Notes

Computationally Hard Problems

Paul Fischer, Carsten Witt
DTU Compute, Danmarks Tekniske Universitet

Version 2.8 (Fall 2020)

Kgs. Lyngby, 10. august 2020

Version 2.8 (Fall 2020), 10. August 2020
This script was produced using LaTeX2ε.

Contents

1 Introduction

In this course, we present some central concepts and results from the area of computer science known as *Complexity Theory*. We will focus on problems that are hard to solve. Everybody has a feeling whether a problem is easy to solve or not. However, this feeling is often very individual. Baking a loaf of bread is easy for some people and hard for others. Also, some problems become easy to solve all of a sudden if one knows the right trick.

We want to specify what it means that a problem is *hard to solve for a computer*. To this end, we shall make precise what we mean by a "problem", by "solving a problem" and of course what we mean by a *computer*. The latter is important because different computers can perform instructions at very different speeds. There are specialized chips for performing complex operations (for example in signal processing) extremely fast. We will thus have to specify the computational model before talking complexity of a solution.

Finally, we have to specify what we mean by "hard". The term *hardness* has very different meanings in Computer Science. It can mean that a problem cannot be solved by any computer, no matter how powerful. One such problem is the *halting problem* which can be stated as: "Given a computer program P and an input I, does P stop on input I or run forever?" Provably, there is no computer program H which takes as input a pair (P, I) (program, input) and which always correctly determines whether P stops on input I. This is **not** the kind of hardness we consider here. Let us remark that there are programs that can do this for **some special** pairs (P, I), but not for all such pairs.

Another way to define hardness of a problem is by saying "solvable in principle but it might take a **very** long time." This **is** the kind of hardness considered here. In Chapters 2 and 3, we supply definitions of the terms mentioned above including the notion of hardness. To this end, we utilize the concept of *randomized algorithms*. These are algorithms which from time to time "flip a coin" and base the further computation on the outcome of this coin flip. One might think of the coin flip as a "potentially helpful guess" that can speed up computation. In Chapter 4 we deal with problems that are hard in our sense. We will see a number of generic problems that are known to be hard. We introduce methods for proving that a new problem is hard.

In Chapter 5, we will use randomized algorithms in a more constructive way. Some design techniques are presented for solving problems by randomized algorithms. Due to randomization, we cannot expect that an algorithm always computes the same output on a fixed input. Different outcomes of the coin tosses in different executions of the algorithm are responsible for this. However, if there is a probabilistic (probability-theoretical) guarantee on the output, then these algorithms are very helpful. Such a probability-theoretical guarantee could have the form:

- The algorithm is always fast and outputs the correct result with high probability.
- The algorithm always computes the right result but the running time can vary considerably on the same input according to some probability-theoretical law.

In the first case, we might occasionally end up with a wrong result. This might be tolerable in some situations. Otherwise we run the algorithm a number of times on the same input. It still might always output the wrong answer but the probability of this to happen is exponentially decreasing. In the second case, we compute how likely a certain short running time on a given input is. Then, we run the algorithm on that input. If it stops fast, we are happy. Otherwise we terminate it when the short running time is exceeded and start it again. The probability-theoretical law tells us how often we have to repeat this process to have the algorithm stop by itself at least once (with high probability).

The authors are grateful for any corrections and suggestions. Please do not hesitate to report typographical errors, to point out parts that are unclear, to complain about confusing notation and also tell us if you would like to see an example for this or that. This document goes back to the 2008 edition of the notes (version 1.8). It includes both corrections and new material, most notably in Section 4.3 and in Section 5.

Further Reading

The following books are suggestions:

- The book [GJ79] is *the* classical book on NP-completeness, the topics considered in Chapter 4. It also contains a collection of more than 200 hard problems. One of the authors (Johnson) regularly wrote articles on new results in the area. These were published as *The NP-completeness column: An ongoing guide.* in the Journal of Algorithms throughout the 1980's.
- The book [Weg05] can be used as a supplement to some of the polynomial-time reductions presented in Chapter 4. Moreover, it covers more recent approaches in complexity theory.

- The book [HU79] emphasis computational models like the one introduced in Chapter 2. It also covers more restricted models like Finite Automata in great detail. A revised and more up-to-date version of this book is [HMU01].
- For the randomized algorithms discussed in Chapter 5, the following books are good choices: [MR95], [MU05].
- More about the analysis of bio-inspired search heuristics, introduced in the last two sections of Chapter 5, can be found in the book [NW10].

2 Definitions and Notation

2.1 Formal Languages

Talking of algorithms and problems includes talking about inputs to an algorithm and its output. For an analysis we have to formalize these notions. In this section, we supply the necessary concepts from the theory of formal languages.

Definition 2.1

- An *alphabet* is a finite set of symbols (also called *letters*). We often use the Greek letter Σ to denote alphabets.
- A *word* $\boldsymbol{w} = w_1 \cdots w_n$ over an alphabet Σ is a finite sequence of symbols of Σ.
- The length $|\boldsymbol{w}|$ of a word $\boldsymbol{w} = w_1 \cdots w_n$ is the number of symbols it contains: $|\boldsymbol{w}| = n$.
- By Σ^n we denote the set of all words over Σ of length exactly n, $n \in \mathbb{N}$. The set Σ^0 contains the *empty word* ε.
- By Σ^* we denote the set of all words over Σ, i.e., $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$.
- A *language* L over Σ is a collection of words over Σ, i.e., $L \subseteq \Sigma^*$.

Note that we have letters, words, and languages but no sentences!

Example 2.2 Let the alphabet be $\Sigma = \{a, b\}$. Then

$$\Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$$

The set $L_{=2b,3}$ of all words over Σ^3 which have exactly 2 b's is a finite language.

$$L_{=2b,3} = \{abb, bab, bba\}$$

The set $L_{=2b}$ of all words over Σ^* which have exactly 2 b's is an infinite language.

$$L_{=2b} = \{bb, abb, bab, bba, aabb, abab, abba, baab, baba, bbaa, aaabb, aabab, \ldots\}$$

Example 2.3 [Language of undirected graphs] We shall use languages to encode the input of algorithms. This example shows how to encode undirected graphs. We encode a graph using its adjacency matrix. There are of course other ways of doing this, like listing all edges as pairs of nodes.

The adjacency matrix of an undirected graph is symmetric and has zeros on the diagonal. Hence, it is sufficient to list the entries above the diagonal. If the graph has n vertices, which we without of generality call $1, \ldots, n$, there are $n(n-1)/2$ such entries. We make the convention that we code the graph row-wise: The first $n-1$ letters are the entries of the first row of the adjacency matrix to the right of the diagonal. The next $n-2$ letters are the entries of the second row to the right of the diagonal, etc.

Every undirected graph with n vertices can thus be coded by a word over $\Sigma = \{0,1\}$ of length $n(n-1)/2$ and every such word encodes an undirected graph. The language of graphs L_{graphs} then is

$$L_{graphs} = \bigcup_{n=0}^{\infty} \{0,1\}^{n(n-1)/2} = \left\{ w \in \{0,1\}^k \mid \exists n \colon k = n(n-1)/2 \right\}$$

Example 2.4 [Coding of clauses] Let a non-empty set $C = \{c_1, \ldots, c_k\}$ of clauses over the boolean variables $x_1, \ldots, x_n$ be given. We code this set by listing the indices of the variables of each clause and indicating for each one whether it is negated or not by a leading $-$ resp. $+$. We also use a special symbol $\#$ to separate the clauses. Hence, our alphabet is $\Sigma_c = \{0,1,2,3,4,5,6,7,8,9,\#,+,-\}$.

We begin by listing the variables of the first clause, then the second etc. The resulting language of clauses is denoted by $L_{clauses}$. Any word not matching the above specification is not in $L_{clauses}$.

Consider the following set C with three clauses:

$$\begin{aligned} C &= \{c_1, c_2, c_3\} \\ c_1 &= \overline{x}_1 \vee \overline{x}_3 \vee x_5 \vee x_4 \vee x_2 \\ c_2 &= x_2 \vee \overline{x}_4 \vee \overline{x}_5 \\ c_3 &= x_2 \vee \overline{x}_3 \vee x_5 \vee x_4 \end{aligned}$$

This is then encoded by the word w:

$$w = \texttt{-1-3+5+4+2\#+2-4-5\#+2-3+5+4}$$

Languages are used to specify the input type for an algorithm. A program implementing a graph algorithm like DFS (depth-first-search) expects (an encoding of) a graph as input. If the input is not (an encoding of) a graph then the program cannot solve the problem and might crash. Hence, it is wise that the program first checks whether the input has the expected form. This is a general and important problem of computer science:

Definition 2.5 Given a language L over some alphabet Σ the *word problem* for L is the problem to decide for every word $w \in \Sigma^*$ whether $w \in L$.

An algorithm *solves* the word problem for L if given a word $w \in \Sigma^*$ it decides whether $w \in L$ or $w \notin L$.

Example 2.6 Consider the problem of deciding whether a graph has a triangle. Then we want to solve the word problem for the language L_{triag}:

$$L_{triag} = \big\{w \in \{0,1\}^k \mid \exists n\colon k = n(n-1)/2 \wedge \text{the graph coded by } w \text{ has a triangle}\big\}$$

When we talk of algorithms (not of programs) in this course, we omit this test of correctness and always assume that the input is of the correct form.

The running time of an algorithm as defined in the next section usually depends on the input. Some inputs might be "easy" for the algorithm, resulting in a short running time. Other inputs are "hard" and take a very long time to be processed. In the analysis of algorithms, one does not consider individual inputs because there are so many and they can be so different that the analysis becomes clumsy. Instead, one groups together inputs (hereinafter mostly denoted by $\boldsymbol{X}$ instead of $\boldsymbol{w}$) of the same *size*. The definition of input size varies with problem under consideration. In any case, the size of an input $\boldsymbol{X} \in \Sigma^*$ has to be proportional to the word length, i. e., to the number of symbols of Σ appearing in $\boldsymbol{X}$.

Definition 2.7 Let Σ be an alphabet and let $\boldsymbol{X} \in \Sigma^*$ be a word over Σ that is an input to some algorithm. Then we denote the *size* by $\|\boldsymbol{X}\|$.

Example 2.8 Consider the language L_{graphs} of graphs. A suitable measure of input size is the word length. Using Example 2.3, this is exactly the potential number of edges in the graph represented by the input. Note that there are then no inputs of size m if m is not of the form $k(k-1)/2$ for some $k \in \mathbb{N}$.

Example 2.9 Consider the language $L_{clauses}$ clauses from Example 2.4. Again one can take the word length to be the input size: $\|\boldsymbol{X}\| = |\boldsymbol{X}|$. Sometimes one considers the total number of occurrences of all variables in the clauses, i. e., the number of plus and minus signs in $\boldsymbol{X}$. This definition does not meet our requirement of being proportional to the word length because coding the variable x_k requires $\lceil \log_{10}(k+1) \rceil$ letters. Nevertheless, this measure of size is often used because the number of different variables is often assumed to be small compared to the number of occurrences of variables.

2.2 Algorithms

There cannot be a mathematically sound definition of the term "algorithm". The reason for this is that an algorithm has to be "general" in some sense, whereas a mathematical definition requires a very high level of specification. The following "definition" has to be seen in this way.

Definition 2.10 An *algorithm* is a non-ambiguous, abstract description of a method for solving a problem or performing a task. The semantics of the algorithm has to be clearly defined, i. e., its input-output behaviour has to be uniquely defined.

The terms non-ambiguous and abstract should be understood in the following way.

non-ambiguous Anyone following the description will get the same result.

abstract The description is independent of the specific environment in which the actions are performed.

The name "algorithm" is a adaption of the name of Persian scholar Abu Ja'far Muhammad ibn Musa **al-Khwarizmi** (780–850). He was the first to schematically describe methods for solving linear and quadratic equations in an algebraic – non-geometric – way. Actually, the word "algebra" is taken from the title of one of his books.

The following is an example of an algorithm. It is abstract as it does not depend on a specific kitchen. It is un-ambiguous but it assumes some common knowledge, e. g., it is assumed that the meaning of "kneading" is known.

Example 2.11

Output: one white bread of 750 g

Input:

500	g	wheat flour
300	ml	water
30	g	yeast
5	g	salt
5	g	sugar

Algorithm: Mix yeast with 100 g flour, 100 ml water and the sugar and let rise for 10 min.
Mix this and all other ingredients and knead for 10 min.
Form a ball and let rise until the volume has doubled.
Knead again, form a loaf and let rise for 40 min.
Then bake in a pre-heated oven for 40 min at 220°C.

Definition 2.12 The *complexity of an algorithm* is its consumption of resources such as

- **Time**
- Memory
- Accesses to background storage
- Network usage
- Number of parallel processes
- Weighted combinations of some of the above
- Power
- ...

Of course, other resources might be important for specific algorithms. Here, we shall focus on time which generally is considered to be the most precious resource. The reason is that "one cannot buy time" while one can buy memory or a faster network.

Any characterization of algorithmic complexity has to be done in an abstract way, i. e., machine-independent. Only then results of different authors are comparable. Also an analysis does not become outdated when the next generation of processors is released.

For an abstract definition of complexity, one has to specify what one means by *atomic computational steps* each of which counts one unit of time. Usually, the following are considered to be atomic computational steps

- Assignment of "small" data types ($a \leftarrow b$)
- Arithmetic operations on "small" numbers ($a + b$, $a * b$)
- Comparison and branching (IF ($a < b$) THEN ... ELSE ...)
- Reading an input bit, writing an output bit

Depending on the problem which the algorithm solves, different and more complex operations can be considered an atomic computational step. For example, for graph algorithms one can use "removing an edge" or "adding a node" as such. One has, however, to make sure that the operation can be performed in constant time, i. e., the time per operation should not grow if one goes from a small to a larger input for the algorithm.

Usually **not** one unit of time: trigonometric operations, exponentiation, arithmetic on "large" numbers, copy-assignments of large data types etc.

Once one has agreed on the atomic computational steps, one is in a position to define the running time of an algorithm. There are three types of running times which are considered: Worst-case, best-case and average case.

Definition 2.13 [Running time] Let A be an algorithm.

- The *running time* $T^A(\boldsymbol{X})$ of an algorithm A for a fixed input $\boldsymbol{X}$ is the number of atomic computational steps that A performs on input $\boldsymbol{X}$ until it stops, and $T^A(\boldsymbol{X}) = \infty$ if A does not stop on $\boldsymbol{X}$.
- The *worst-case running time* $T^A_w(n)$ of algorithm A for input size n is the longest running time of the algorithm on an input of size n:

$$T^A_w(n) := \max\{T^A(\boldsymbol{X}) \mid \|\boldsymbol{X}\| = n\}$$

- The *best-case running time* $T^A_b(n)$ of algorithm A for input size n is the shortest running time of the algorithm on an input of size n:

$$T^A_b(n) := \min\{T^A(\boldsymbol{X}) \mid \|\boldsymbol{X}\| = n\}$$

- The definition of *average-case running time* $T_a^A(n)$ of algorithm A for input size n assumes a distribution P_n on the inputs of size n. It then is the average with respect to P of all running times on inputs of size n:

$$T_a^A(n) := \sum_{\boldsymbol{X}:\ \|\boldsymbol{X}\|=n} P_n(\boldsymbol{X}) \cdot T^A(\boldsymbol{X})$$

If the algorithm A is clear from the context, we drop the superscript and use $T(\boldsymbol{X})$ instead of $T^A(\boldsymbol{X})$.

The best-case running time is often uninteresting because it can frequently be made to be $T_b^A(n) = n$: It is often possible to identify for every input size n one specific input $\boldsymbol{X}^*$ by simply reading the input in time $O(n)$. If the output for the $\boldsymbol{X}^*$ is known it can be given right away. For example assume we want to check whether a given undirected graph has a triangle. Then we could check by just reading the input whether it has at most two edges. If that is the case we can immediately output NO. The best-case running time then is $O(n)$.

In this course we will mainly consider the worst-case and average-case running times.

The running times of algorithms are classified according to the scheme in Table 2.1.

Running time	**Name**
$O(1)$	constant
$o(n)$	sub-linear
$O(n)$	linear
$O(n \log(n))$	—
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^{\text{constant}})$	polynomial
$O\big(n^{\log(n)}\big)$	super-polynomial / sub-exponential
$O(\text{constant}^n)$	exponential (e. g. 2^n)

Table 2.1: Magnitudes of running times for input size n and their names.

It is natural to ask when we consider an algorithm to be "good", i. e., "fast". The answer to this question depends on the person we ask. Almost everybody will agree

that sub-linear, linear and $O(n \log n)$ are fast running times. A person from practice might also consider quadratic running time OK but not really fast; anything beyond, he will not consider practical. A theoretical computer scientist considers any polynomial running time fast, even if it is n^{17}. The reason is given in the following proposition the proof of which is an exercise.

Proposition 2.14 *Suppose an algorithm has been implemented on some machine. With the current hardware, this can solve problems up to size n in a fixed time t_0. Now suppose that the speed of the hardware is doubled. If the running time of the algorithm is polynomial then there is a constant $c_p > 1$ such that one can now solve problems of size $c_p \cdot n$ in time t_0. If the running time of the algorithm is exponential then there is a constant $c_e > 0$ such that one can now solve problems of size $c_e + n$ in time t_0.*

Proposition 2.14 motivates the following definition.

Definition 2.15 Let A be an algorithm. We call A *efficient* if its worst-case running time is upper bounded by a polynomial, i. e., if there is a polynomial p such that

$$\forall n \in \mathbb{N}\colon T_w^A(n) \leq p(n).$$

Figure 2.1 shows our view on the world. Table 2.2 shows the running times of some

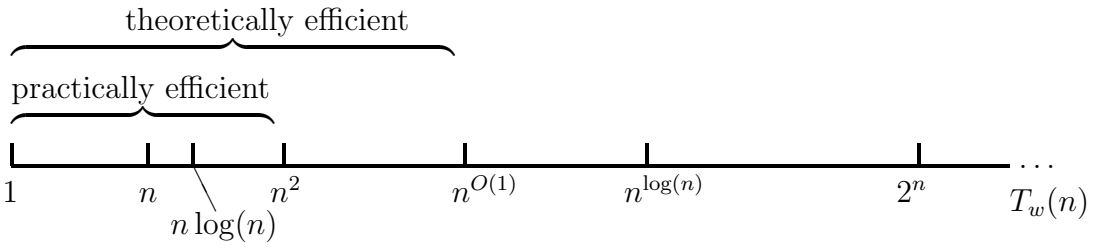


Figure 2.1: A view on the running times of algorithms.

algorithms. Note that we did not always pick the fastest known algorithm for a specific problem.

Algorithm	$T_w(n)$	$T_a(n)$
Median on n small numbers	$O(n)$	$O(n)$
Quicksort on n small numbers	$O(n^2)$	$O(n \log(n))$
Heapsort on n small numbers	$O(n \log(n))$	$O(n \log(n))$
$n \times n$ Matrix multiplication (naive) $N = n^2$	$O(n^3) = O(N^{1.5})$	$O(n^3)$
$n \times n$ Matrix multiplication (Strassen) $N = n^2$	$O(n^{2.8}) = O(N^{1.4})$	$O(n^{2.8})$
DFS on graphs $\|E\| = m$, $\|V\| = n$	$O(n + m)$	$O(n + m)$
Triangle in a graph $\|E\| = m$, $\|V\| = n$	$O(n^3)$	*)
Backtrack for Euler cycle	$O(n!)$	*)
Backtrack for Hamiltonian cycle	$O(n!)$	*)

*) depends on the random graph model.

Table 2.2: Worst- and average-case running times of some computer algorithms. For some of the problems faster algorithms than the ones listed here are known.

2.3 Problems

We shall now define what we mean by a *problem*. Here we are mostly considering *decision problems*, also called *yes-no-problems*. An solution to such a problem is either YES or NO. Most of the time, a decision problem has an associated *optimization problem*.

Here are some informal examples of such problems:

Problem 2.16
Input: A sequence $a_1, \ldots, a_n$ of n integers.
Output for the decision version: YES if the sequence is ascending, i. e., if $a_1 \leq a_2 \leq \cdots \leq a_n$ and NO otherwise.
Output for the optimizing version: An increasingly sorted sequence consisting of the input integers.

For an undirected graph $G = (V, E)$, a *clique* is a subset $V' \subseteq V$ of the vertices such that all edges are present, i. e., for all $v, w \in V'$, $v \neq w$: $(v, w) \in E$. We say that G has a *k-clique* if $|V'| = k$.

Problem 2.17 [CLIQUE]
Input for the decision version: An undirected graph G and a number k.
Input for the optimizing version: An undirected graph G.
Output for the decision version: YES if G contains a clique of k vertices and NO otherwise.
Output for the optimizing version: A set of vertices which forms a clique of maximal size.

Problem 2.18 [SATISFIABILITY]
Input: A set of clauses over n boolean variables.
Output for the decision version: YES if there is is an assignment of truth values to the variables which satisfies all clauses and NO otherwise.
Output for the optimizing version: An assignment of truth values to the variables which satisfies all clauses if such an assignment exists and NO otherwise.

Now we can formally define a decision problem to be a language. For example the problem SATISFIABILITY is the language L_{SAT} over the alphabet Σ_{clauses} (defined in Example 2.4) which consists of all words w such that w is the encoding of a set of clauses which has a satisfying assignment.

$$L_{\text{SAT}} := \{\boldsymbol{w} \in L_{\text{clauses}} \mid \boldsymbol{w} \text{ has a satisfying assignment}\}$$

Note that L_{SAT} is a proper subset of L_{clauses}.

Definition 2.19

- A *decision problem* P is a language L_P over an input alphabet Σ.
- We say that an algorithm A *solves* the problem P if it solves the word problem for L_P. That is, if it decides whether a word $w \in \Sigma^*$ is in L_p or not. We call A a *decision algorithm*.

In this course we, shall often specify a problem in an informal way without defining an alphabet and a language. The reader is advised to perform this for some problems to gain a better understanding of the formal definition.

Remark 2.20 There is a difference between a problem and a *problem instance*. A *problem* is a language. A *problem instance* is a word of the language.

For example, a problem instance of the SATISFIABILITY problem is (the word describing) a concrete set of clauses.

In the literature (including this notes), the distinction is not always made. So the term "problem" might sometimes refer to a problem instance.

2.4 Conversion from Decision to Optimization Problems

Clearly, any algorithm solving the optimization version of a problem also solves the decision version. We shall now see that the converse is also often true: One can sometimes use a decision algorithm for a problem as a *black box* for solving the optimization version of the problem. What we mean by this is that one can construct an algorithm for solving the optimization problem that from time to time uses the decision algorithm as a subroutine. In the following, we use CLIQUE as an example and further examples are given in the exercises.

Suppose we have a decision algorithm A_d for the CLIQUE problem. An algorithm A_o for the optimization version can be derived as follows: Given the encoding of a graph $G = (V, E)$, we first determine the size $k^* \geq 1$ of a maximal clique (note that every vertex is a clique of size 1). This is done by binary search. That is, we first invoke the decision algorith A_d with G and $k = n$. If the answer is YES, we know that $k^* = n$. If the answer is NO, we invoke the decision algorithm with $k = \lceil n/2 \rceil$ and proceed accordingly. After at most $\lceil \log_2 n \rceil$ calls to A_d, we have determined k^*. The aim is now to output the vertices of a clique of that size. Note that there might be more than one such clique.

To find the vertices of a clique of size k^*, it is convenient to identify the edges between the vertices of the clique. This is done in the following way: We initialize a set of edges C to be empty. Then we delete an arbitrary edge from $E \setminus C$. Let the resulting graph be denoted by $G = (V, E)$ again. We then run A_d on it, using $k := k^*$ in this and all future calls. If A_d answers YES, we delete another edge and so on until A_d answers NO. In this case the last edge, call it e_1, that was deleted belongs to a k-clique, which was destroyed by deleting e_1. We put e_1 into the set C which eventually will contain the edges of a k^*-clique. No edge in C will ever be deleted, i. e., from now on always $e_1 \in E$. We now go on deleting edges from $E \setminus C$ until again A_d answers NO on the resulting graph. Again the edge e_2 deleted last is put into C. We claim that in the resulting graph (i. e., with e_2 put back into E) all existing k-cliques contain both e_1

and e_2. If, e. g., there were a k-clique containing e_2 but not e_1 then that clique was also present, when A_d was run after deleting e_1. But then A_d would have answered YES, contradicting that e_1 was deleted. Now we iterate the above procedure until the size of C becomes $k^*(k^*-1)/2$. We claim that the set C then contains the edges of a k^*-clique and we can output the vertices belonging to the edges in C as a clique.

The algorithm is correct for the following reason: As long as the decision algorithm returns YES, the is a clique of size k^* on the graph given to the decision algorithm. In particular, there is a clique on the graph obtained after having deleted some edges. If the decision algorithm answers NO, it must have answered YES in the previous call, and the edge e that was deleted must be necessary for all remaining k^*-cliques. Since e is immediately restored after the NO and put into C, we can be sure that the clique constructed in the end contains e.

Altogether the algorithm A_o makes at most $\lceil \log n \rceil + n(n-1)/2$ calls to A_d, where $n = |V|$. It needs constant time in each iteration, whence the running time is $O(n^2)$ if a call to A_d is constant time.

The procedure described above is a *black-box approach*. The decision algorithm is the black-box, the internal implementation of which is not known and in fact irrelevant. It is used as an "oracle", to which one poses specific questions. The questions are formulated in such a way that the oracle's answers help solving the optimization problem.

If the optimization problem can be solved in polynomial time by such a black-box approach (counting each question to the oracle as one step) then we say that we have an *efficient conversion* to the optimization problem.

3 Randomized Algorithms and Complexity Classes

So far we have looked at classical *deterministic algorithms*. Such algorithms perform the same steps every time they are presented the same input and thus produce the same output. This is certainly a desirable behaviour. Nevertheless, we shall now look at algorithms which are allowed to perform different computations if the same input is given repeatedly. We shall later see how we can benefit from this concept in two ways. First, we will use it to classify certain problems as being hard, secondly we shall use these algorithms to solve problems fast which are hard (or, at least, more difficult or less elegantly) to solve for deterministic algorithms (the notion of "solving" has to made precise in this context).

3.1 Definition of Randomized Algorithms

We are introducing the non-deterministic behavior in the following way. The algorithms now have access to a *fair coin*. This is a device that on demand outputs a 0 or 1; we shall call the execution of one such experiment a *coin flip* or *trial*. The numbers are assumed to be generated **independently according to uniform distribution** on $\{0, 1\}$. This implies that the probability of seeing a 0 in a trial is 0.5 as is that for observing a 1. The algorithms can use these coin flips in the following way through a *randomized IF-statement*:

if $(coin = 1)$ **then**
  do something
else
  do something else
end if

If random numbers other than 0 and 1 are needed, these can be approximately produced by repeatedly using the fair coin.

If it is helpful, we assume that we have access to a *random number generator*. This is a subroutine $rand(a, b)$ which receives two integers a, b, $a < b$ and returns a random number drawn **independently according to uniform distribution** from the set $\{a, a+1, \ldots, b-1, b\}$. Setting $a = 0$ and $b = 1$ will make the random generator into the fair coin. One can use this to assign a random number to a variable.

INTEGER v
$v \leftarrow \text{rand}(a, b)$

Generating a random number counts as one computational step. We shall say that the algorithms *calls* the random number generator.

Remark 3.1 We shall always assume that we are dealing with ideal random number generators, i.e, that they produce numbers that are stochastically **independent** and produced **according to uniform distribution**. Software random generators do not meet this requirement because they themselves are deterministic algorithms. Nevertheless, good random generators – like RAND55 described in the appendix – are close enough to ideal random generators for most practical purposes.

Definition 3.2 An algorithm which has access to a random number generator is called a *randomized algorithm*.

For randomized algorithms, the notions of running time as given in Definition 2.13 do not make sense any more. The number of steps until the algorithm stops is not determined even for a fixed input; it depends on the outcomes of the random experiments.

Also the output is not determined any longer, because it depends on the coin flips. Both running time and output are **random variables** or random vectors. If there is no control on these variables then a randomized algorithms is not of great use. Often, however, there is a **probabilistic (probability-theoretical) control** on these quantities. At least, a finite expected value of the running time is desired.

Definition 3.3 [Running time of randomized algorithms] Let R be a randomized algorithm.

- The *expected running time* $\boldsymbol{E}\big[\, T^R(\boldsymbol{X}) \,\big]$ of an algorithm R for a fixed input $\boldsymbol{X}$ is the expected value of the number of atomic computational steps that R performs on input $\boldsymbol{X}$ until it stops ($\boldsymbol{E}\big[\, T^R(\boldsymbol{X}) \,\big] = \infty$ is possible).
- The *worst-case running time* $T^R_w(n)$ of algorithm R for input size n is the longest running time of the algorithm on an input of size n, where a worst possible

outcome of the random numbers and the input is assumed:

$$T^R_{\mathrm{w}}(n) := \max\{t \mid \boldsymbol{P}\big[\, T^R(\boldsymbol{X}) = t \,\big] > 0, \|\boldsymbol{X}\| = n\}$$

- The *best-case running time* $T^R_b(n)$ of algorithm R for input size n is the shortest running time of the algorithm on an input of size n, where a best possible outcome of the random numbers and the input is assumed:

$$T^R_{\mathrm{b}}(n) := \min\{t \mid \boldsymbol{P}\big[\, T^R(\boldsymbol{X}) = t \,\big] > 0, \|\boldsymbol{X}\| = n\}$$

In the following, we shall present an example to which we shall return later in Chapter 5 where we perform a thorough analysis of the expected running time.

3.2 Examples of Randomized Algorithms

In this chapter, we shall use randomized algorithms to classify problems as more or less hard. Besides this, there are good reasons for using randomized algorithms in practical applications for the following reasons.

- A better running time than deterministic algorithms.
- Less memory requirements than deterministic algorithms.
- Easier implementation than deterministic algorithms.
- The opportunity to foil an adversary.

Example 3.4 [Expected Running Time] Consider an algorithm which performs a loop in which a coin is flipped. The loop is exited if the coin returns a 1.

while $(coin = 0)$ **do**
  do something
end while

This program does not terminate if the coin never shows 1. We shall now calculate the probability that the loop is performed exactly n times. If X is the random variable representing the number of times the loop is performed until termination then we are interested in the quantities $\boldsymbol{P}[\, X = n \,]$, $n \in \mathbb{N}$. For the loop not to be executed at all $(n = 0)$, the first coin flip has to be 1. This happens with probability 0.5, whence $\boldsymbol{P}[\, X = 0 \,] = 0.5$. For the loop to be executed exactly once $(n = 1)$, the first coin flip has to be 0 **and** the second has to be 1. This happens with probabilities 0.5 and 0.5 respectively, whence $\boldsymbol{P}[\, X = 1 \,] = 0.5 \cdot 0.5 = 0.25$. Here we use Formula (A.1) as the

coin flips are independent. For the loop to be executed exactly n times, the first n coin flips have **all** to be 0 **and** the $(n+1)$-st coin flip has to be 1. This happens with probabilities 0.5^n and 0.5 respectively, whence the general formula is

$$\boldsymbol{P}[\,X = n\,] = 0.5^n \cdot 0.5 = 0.5^{n+1}\ .$$

Note that the sum $\sum_{n=0}^{\infty} \boldsymbol{P}[\,X = n\,]$ of these probabilities is 1 and that the probabilities are exponentially decreasing with n. Long running times are highly unlikely.

We now compute the expected running time of the algorithm, i.e., the expected number $\boldsymbol{E}[\,X\,]$ of times the loop is executed. By the Formula (A.6), this is the sum of the possible executions of the loop $(0, 1, 2, \ldots)$ times the probabilities that they occur:

$$\boldsymbol{E}[\,X\,] = \sum_{n=0}^{\infty} n \cdot \boldsymbol{P}[\,X = n\,] = \sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n+1}\ .$$

We claim that this sum is 1. To prove the claim, we use an *accounting method* (also called *bookkeeper's trick*) of reordering the terms and summing in a different way:

$$\begin{array}{rcccccccccccl}
\sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n+1} & = & 0 \cdot \frac{1}{2} & + & 1 \cdot \left(\frac{1}{2}\right)^2 & + & 2 \cdot \left(\frac{1}{2}\right)^3 & + & 3 \cdot \left(\frac{1}{2}\right)^4 & + & 4 \cdot \left(\frac{1}{2}\right)^5 & + \cdots & \\
& = & 0 & + & \left(\frac{1}{2}\right)^2 & + & \left(\frac{1}{2}\right)^3 & + & \left(\frac{1}{2}\right)^4 & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & + & \left(\frac{1}{2}\right)^3 & + & \left(\frac{1}{2}\right)^4 & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & & & + & \left(\frac{1}{2}\right)^4 & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & & & & & + & \left(\frac{1}{2}\right)^5 & + \cdots & \\
& & & & & & & & & & \ddots & + \cdots & \\
& = & 0 & + & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + \cdots & = \frac{1}{2} \\
& & & & & + & \frac{1}{8} & + & \frac{1}{16} & + & \frac{1}{32} & + \cdots & = \frac{1}{4} \\
& & & & & & & + & \frac{1}{16} & + & \frac{1}{32} & + \cdots & = \frac{1}{8} \\
& & & & & & & & & + & \frac{1}{32} & + \cdots & = \frac{1}{16} \\
& & & & & & & & & & \ddots & + \cdots & = \frac{1}{32} \\
\hline
& & & & & & & & & & & & 1
\end{array}$$

After expanding the formula, we compute the sums S_n over the rows to get the well known value $S_n = (1/2)^n$ for the n-th row. Then we sum the row sums to get $\sum_{n=1}^{\infty} S_n = \sum_{n=1}^{\infty} \frac{1}{2}^n$ which is 1. The reordering of the infinite sum is allowed because all terms are positive.

Appendix B.3 contains more general formulas that allow us to treat infinite sums as in the previous example.

Example 3.5 [Foiling an Adversary] Assume that we are given $n = 3k$ balls. A third is colored red, a third green, and the last third blue. The balls are also numbered $1, \ldots, n$, but the numbers do not give information on the color. We have to find three equally colored balls. We do not see the balls but we can ask someone a questions like: "Do the balls 1, 12, and 142 have the same color?" In case of a "no" we are not told the colors. We could enumerate all $\binom{n}{3}$ triples $\{i, j, m\}$ of different numbers and ask the corresponding question.

Now suppose the balls are numbered by an *adversary* whose aim it is to maximize the running time. If the adversary knows what order we enumerate the triples in, he will number the balls in such away that we find a triple of one color as late as possible.

Now consider a randomized strategy, which picks a triple at random until it is sccessful.

```
boolean success
success ← FALSE
while not success do
  i ← rand(1, n)
  j ← rand(1, n)
  m ← rand(1, n)
  if i, j, m are different then
    if i, j, m have the same color then success ← TRUE
    end if
  end if
end while
```

Here are two conditions to be met to be successful. First, we have to get three **different** numbers in the three calls to $rand(1, n)$. Secondly, if the numbers are different then the tree balls have to have the same color.

Let us denote the probability of the first condition by q. We claim that $q = 1-3/n+2/n^2$ and leave the proof to the reader. If the three balls are different then we compute the probability of them having the same color as follows. Assume the first ball is red. Then the chances for the second one to be red also are $(k-1)/(n-1)$. Then the chances for the third one to be red also are $(k-2)/(n-2)$. Hence the probability for three red balls is $((k-1)(k-2))/((n-1)(n-2))$ if the first is red. So the probability of getting three balls with the same color is

$$\frac{(k-1)(k-2)}{(n-1)(n-2)},$$

which approaches $(1/9)$ from below if n grows. Altogether, the probability p of success in a single execution of the loop is

$$p := (1 - 3/n + 2/n^2) \cdot \frac{(k-1)(k-2)}{(n-1)(n-2)} .$$

Let us determine the probability $\boldsymbol{P}[\,X = t\,]$ that it takes exactly t rounds (executions of the while loop) to success. In order that the first success is in round t, there must be failures in rounds $1, \ldots, t-1$ and a success in round t, i.e.,

$$\boldsymbol{P}[\,X = t\,] = (1-p)^{t-1} \cdot p.$$

The expected running time is:

$$\boldsymbol{E}[\,X\,] = \sum_{t=1}^{\infty} t \cdot (1-p)^{t-1} \cdot p.$$

In the table below the values for $\boldsymbol{P}[\,X = t\,]$ and $k = 100$ ($n = 300$) are listed for $t = 1, \ldots, 10$. The expected running time is (numerically calculated) $\boldsymbol{E}[\,X\,] = 9.092$.

t	$(1-p)^{t-1} \cdot p$	$t \cdot (1-p)^{t-1} \cdot p$
1	0.1078000000	0.1078000000
2	0.09606204126	0.1921240825
3	0.08560218711	0.2568065613
4	0.07628126929	0.3051250772
5	0.06797527308	0.3398763654
6	0.06057368726	0.3634421236
7	0.05397803381	0.3778462367
8	0.04810055761	0.3848044609
9	0.04286305890	0.3857675301
10	0.03819585281	0.3819585281

Example 3.6 [MaxCut] The MaximumCut (or MaxCut) problem is defined on Page 61. The following randomized algorithm computes a cut in a graph $G = (V, E)$, i.e., it partitions $V = \{v_1, v_2, \ldots, v_n\}$ into V_1, V_2. We want to determine the expected number X of edges between V_1, V_2.

Algorithm 3.7

for $i = 1, 2, \ldots, |V|$ **do**
  if coin $= 0$ **then**
    put v_i into V_1
  else
    put v_i into V_2
  end if
end for

Claim 3.8 *Algorithm 3.7 runs in time $O(n)$. The expected number of edges between V_1, V_2 is $|E|/2$.*

Proof. The running time obviously is $O(n)$. Consider an edge $\{a, b\} \in E$. Let X be the random variable denoting the number of edges in the cut. Edge $\{a, b\}$ is in the cut iff

$$[(a \in V_1) \wedge (b \in V_2)] \vee [(b \in V_1) \wedge (a \in V_2)] \tag{3.1}$$

As we use a fair coin, we have

$$\boldsymbol{P}[\, a \in V_1 \,] = 0.5, \quad \boldsymbol{P}[\, a \in V_2 \,] = 0.5, \quad \boldsymbol{P}[\, b \in V_1 \,] = 0.5, \quad \boldsymbol{P}[\, b \in V_2 \,] = 0.5 \ .$$

Then event 3.1 happens with probability

$$\begin{aligned} \boldsymbol{P}[\, \{a, b\} \text{ is in cut} \,] &= \boldsymbol{P}[\, a \in V_1 \,] \cdot \boldsymbol{P}[\, b \in V_2 \,] + \boldsymbol{P}[\, b \in V_1 \,] \cdot \boldsymbol{P}[\, a \in V_2 \,] \\ &= 0.5 \cdot 0.5 + 0.5 \cdot 0.5 = 0.5. \end{aligned}$$

Using Equation A.19, we get

$$\boldsymbol{E}[\, X \,] = \sum_{\{a,b\} \in E} \boldsymbol{P}[\, \{a, b\} \text{ is in cut} \,] = \sum_{\{a,b\} \in E} 0.5 = 0.5 \cdot |E| \,,$$

which finishes the proof. □

Let us sketch how one uses such algorithms in practice:

- Given a graph $G(V, E)$.
- Run the randomized algorithm, an let C be the resulting cut, i.e., the set of edges between V_1 and V_2.
- If $|C| \geq (1/2)\,|E|$, be happy and stop.
- If $|C| < (1/2)\,|E|$, run the algorithm again.
- Reason: Because the **expected** size of a cut is $(1/2)\,|E|$ **some** executions of the algorithm have to yield cuts of that size or larger.
- We repeat until we are lucky.
- As the running time is fast, we can do this many times.

Example 3.9 [Graph Coloring] The problem is defined on Page 61.

Our algorithm RandGC receives the graph G and the colors $1, \ldots, k$ as input. It randomly assigns a color to every vertex and then checks whether this is a legal coloring.

```
proc RandGC(G,k)
for all nodes v ∈ V do
  color(v) ← rand(1, k)
end for
for all edges {v, w} ∈ E do
  if color(v) = color(w) then
    return(NO)
  end if
end for
return(YES)
end proc
```

If the graph G does not have a k-coloring then algorithm RandGC will always answer NO. If G does have a k-coloring, the algorithm **can** guess it. That means, there is a positive chance that all assignments of the form $color(v) \leftarrow rand(1, k)$ will produce such a coloring. In the worst case, G has only a single k-coloring (up to permuting the colors). We very crudely estimate the probability to guess this. Then for every vertex the chance to receive the right color is $(1/k)$. The probability that all nodes get the right color thus is

$$\boldsymbol{P}[\,\text{success}\,] = \left(\frac{1}{k}\right)^n \cdot k! \qquad \text{where} \qquad n = |V|\,.$$

The term $k!$ is the number of permutations of the k colors. For n only slightly larger than k, this probability is very small.

More examples will be presented in Chapter 5.

3.3 Complexity Classes

In this chapter, we shall use the concept of randomized algorithms to classify problems as being hard. To this end, we augment randomized algorithms in such a way that they always terminate after a predetermined number of steps. Let $f\colon \mathbb{N} \mapsto \mathbb{N}$ be an easily computable function. We want to ensure that the algorithms always stop after $f(n)$ steps, where n is the input size. Let A be a randomized algorithm. We construct a terminating algorithm A' as follows. Algorithm A' first computes the input length n by

scanning the input. Then A' computes the runtime bound $f(n)$ and initializes a counter $c = 0$. Then the code of A follows, where at every instruction (atomic computational step) of A, code is added to perform the following:

- Increment the counter c by one.
- Check if c exceeds the running time bound ($c > f(n)$).
- If this is the case then A' is terminated. If an output is expected, A' will deliver a default value, for example DON'T KNOW.
- If A' stops earlier, it returns the output which A would have returned.

Sometimes, a less complicated way can be chosen to guarantee termination after a fixed number of steps. In the program below, it suffices to guarantee termination of the while-loop because there is no call to the random generation in its body.

$bound \leftarrow x$
$count \leftarrow 0$
while $((count < bound) \wedge (coin = 0))$ **do**
  do something
  $count \leftarrow count + 1$
end while

Assume that "do something" takes constant time.

Definition 3.10 A randomized algorithm whose running time is bounded by a function $f\colon \mathbb{N} \mapsto \mathbb{N}$ is called *f-bounded*.

That is, the algorithm stops after at most $f(|\boldsymbol{x}|)$ steps on input $\boldsymbol{x}$. The output might be the default output ("DON'T KNOW").

Note that using the method to terminate the algorithm described before the definition increases the running time only by a constant factor. Also note that the function f measures the atomic computational steps of the original algorithm A. We shall now present a new view on randomized algorithms using the following observation.

Observation 3.11 *Let $f\colon \mathbb{N} \mapsto \mathbb{N}$ be a function. On input $\boldsymbol{X}$ an f-bounded randomized algorithm makes at most $f(\|\boldsymbol{X}\|)$ calls to the random number generator.*

We can thus produce enough random numbers in advance by calling the random number generator $f(\|\boldsymbol{X}\|)$ times and supply them to the algorithm as an additional input R. The algorithm then stores these numbers in a list. Every time it needs a new

random number, it takes a **new** one from the list instead of calling the random number generator. We shall write

$$A(\boldsymbol{X}, R)$$

to indicate that f-bounded algorithm A is run on input $\boldsymbol{X}$, and R is (a coding of) a sequence of $f(\|\boldsymbol{X}\|)$ random numbers. We shall always assume without further notice that the random numbers in R are of the correct type. So, if A uses a fair coin then R will be a random bit string, if A needs numbers from the set $\{2, 3, 4\}$ then R will consist such numbers.

When talking of the input size, we only consider the "true" input $\boldsymbol{X}$, not the random string R. The reason is that the string R is used to model the – otherwise internal – randomization of the algorithm.

The randomness of an algorithm is now induced through the additional argument R. For fixed $\boldsymbol{X}$ and R, the algorithm behaves deterministically. Different random information R might, however, give different outputs for the same $\boldsymbol{X}$. Hence, the output is a random variable with respect to the the distribution of R. Let $\boldsymbol{X}$ be an input for a f-bounded randomized algorithm A. Let $a, b \in \mathbb{N}$, $a \leq b$ be such that A only used random numbers from $\{a, \ldots, b\}$. Then for $A(\boldsymbol{X}, R)$ the additional argument R is an element of $\{a, \ldots, b\}^{f(\|X\|)}$. We make the following general assumption:

Assumption 3.12 *Let $\boldsymbol{X}$ be an input for a f-bounded randomized algorithm A. The random argument R of an algorithm is generated according to* ***uniform distribution*** *on $\{a, \ldots, b\}^{f(\|\boldsymbol{X}\|)}$.*

The random argument R can be regarded as **help information** for the algorithm. One could imagine that this information helps algorithm do the computations fast. As the help information is random, we can only hope that it really helps **most of the times**. That is, we want to focus on situations where there is some **probabilistic guarantee** that the random information really helps.

We shall classify problems according to the strength of the probabilistic guarantee that can be imposed on algorithms solving them. We restrict ourselves to yes-no-problems. A *complexity class* is a set of problems, i.e., a set of languages possibly over different alphabets. Problems in the same class have similar computational complexity. We shall define the classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{RP}$, $\mathcal{BPP}$, and $\mathcal{ZPP}$ below. Many more classes have been defined over the years but these remained the most important ones.

The first class of problems we describe are those which are efficiently solvable in the classical sense, i.e., by deterministic algorithms.

Definition 3.13 A yes-no–problem is in $\mathcal{P}$ if there is a polynomial p and a **deterministic** p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & A(\boldsymbol{X}) = \text{YES}. \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & A(\boldsymbol{X}) = \text{NO}. \end{array}$$

We start with very weak probabilistic conditions by defining when a problem is in the *complexity class* $\mathcal{NP}$, where $\mathcal{NP}$ stands for *non-deterministic polynomial time*.

Definition 3.14 A yes-no–problem is in $\mathcal{NP}$ if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \exists R, \|R\| \leq p(\|\boldsymbol{X}\|) \;:\; & A(\boldsymbol{X}, R) = \text{YES}. \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \forall R \;:\; & A(\boldsymbol{X}, R) = \text{NO}. \end{array}$$

Here, R is a sequence of random numbers of the type required by the algorithm. An algorithm with these properties is called an $\mathcal{NP}$*-algorithm*.

Let us discuss the definition. The first condition only requires that for YES-inputs there is **some** random information R such that the algorithm computes the correct answer. The algorithm is allowed to compute the wrong answer for all other choices of R. For NO-inputs, the algorithm has **always** to give the correct answer no matter what random information R is supplied. We call such an behaviour *one-sided error*, because errors are only on the YES-inputs.

We will use the discussion in the last paragraph to reformulate Definition 3.14 in terms of probabilities. The first condition expresses that the chances that for a fixed YES-input $\boldsymbol{X}$ a randomly chosen R will lead to a correct result might be small but is not zero. The second condition states that the chances of a wrong answer on a NO-input are zero.

Definition 3.15 A yes-no–problem is in $\mathcal{NP}$ if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & > \quad 0, \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = \quad 1, \end{array}$$

where $\boldsymbol{P}_R[\, Z\,]$ denotes the probability of event Z over uniform distribution of R, $\|R\| \leq p(\|\boldsymbol{X}\|)$

If we strengthen the probabilistic conditions, we get other complexity classes.

Definition 3.16 A yes-no–problem is in $\mathcal{RP}$ (*random polynomial time*) if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & \geq \frac{1}{2}, \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = 1. \end{array}$$

An algorithm with these properties is called an $\mathcal{RP}$*-algorithm*.

$\mathcal{RP}$-algorithms are also called *Monte Carlo*-algorithms. They have one-sided error. In contrast to $\mathcal{NP}$-algorithms, there is a good chance of getting the correct result for YES-inputs, namely at least 50%.

Definition 3.17 A yes-no–problem is in $\mathcal{BPP}$ (*bounded error probabilistic polynomial*) if there is a polynomial p and an $\varepsilon > 0$ (independent of $\|\boldsymbol{X}\|$) and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & \geq \frac{1}{2} + \varepsilon. \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & \geq \frac{1}{2} + \varepsilon. \end{array}$$

An algorithm with these properties is called an $\mathcal{BPP}$*-algorithm*.

$\mathcal{BPP}$-algorithms are allowed to have two-sided error, but the probability of answering correctly should be strictly greater than 1/2 in both cases. Note that we only demand that *there is* a positive constant ε, and not all possible values of ε have to be considered to prove that a problem is in $\mathcal{BPP}$. No matter if the success probability (the probability of answering correctly) of a randomized p-bounded algorithm is 99%, or 75%, or 50.1% etc., these success probabilities are sufficient to prove that the problem is in $\mathcal{BPP}$ (using $\varepsilon = 0.49$, $\varepsilon = 0.25$, and $\varepsilon = 0.01$, respectively). In the literature, people sometimes define $\mathcal{BPP}$ with success probability 2/3 and sometimes with success probability 3/4, all of which are equivalent. The proof is left to the reader.

Definition 3.18 A yes-no–problem is in $\mathcal{ZPP}$ (*zero error probabilistic polynomial*) if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lrlll}
\text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & \geq \frac{1}{2}. \\
\text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = 0. \\
\text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & \geq \frac{1}{2}. \\
\text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & = 0.
\end{array}$$

An algorithm with these properties is called a *$\mathcal{ZPP}$-algorithm*.

Note that the definition of $\mathcal{ZPP}$ allows that we do not get an answer at all. Alternatively, one can assume the answer DON'T KNOW in that case. $\mathcal{ZPP}$-algorithms are also called *Las Vegas*-algorithms.

The above definitions are summarized in the following table:

Class	**True answer YES**	**True answer NO**
$\mathcal{NP}$	$\boldsymbol{P}[\,\text{YES}\,] > 0$	$\boldsymbol{P}[\,\text{NO}\,] = 1$
$\mathcal{RP}$	$\boldsymbol{P}[\,\text{YES}\,] \geq \frac{1}{2}$	$\boldsymbol{P}[\,\text{NO}\,] = 1$
$\mathcal{BPP}$	$\boldsymbol{P}[\,\text{YES}\,] \geq \frac{1}{2} + \varepsilon$	$\boldsymbol{P}[\,\text{NO}\,] \geq \frac{1}{2} + \varepsilon$
$\mathcal{ZPP}$	$\boldsymbol{P}[\,\text{YES}\,] \geq \frac{1}{2}\,, \quad \boldsymbol{P}[\,\text{NO}\,] = 0$	$\boldsymbol{P}[\,\text{NO}\,] \geq \frac{1}{2}\,, \quad \boldsymbol{P}[\,\text{YES}\,] = 0$

Table 3.1: Overview over the complexity classes.

Theorem 3.19 *The following relations hold among the complexity classes, see also Figure 3.1 for an illustration.*

$$\begin{aligned}
\mathcal{P} &\subseteq \mathcal{ZPP} \\
\mathcal{ZPP} &\subseteq \mathcal{RP} \\
\mathcal{RP} &\subseteq \mathcal{NP} \\
\mathcal{RP} &\subseteq \mathcal{BPP} \\
\mathcal{P} &\subseteq \mathcal{BPP}
\end{aligned}$$

Proof. The proof is by looking at the definitions of the classes. If the definition of class $\mathcal{A}$ is a restriction of the definition of class $\mathcal{B}$ then we have $\mathcal{A} \subset \mathcal{B}$. Ony the relation $\mathcal{RP} \subseteq \mathcal{BPP}$ is non-trivial to prove. The additional argument required here is as follows:

Boost the success probability of the $\mathcal{RP}$-algorithm to at least $1/2 + \epsilon$ by invoking the original $\mathcal{RP}$-algorithm several times. The details should be filled in by the reader (in an exercise). □

Since the success probability of an $\mathcal{NP}$-algorithm might be exponentially small (e. g., 2^{-n}, where n is the input length), polynomially many calls of an $\mathcal{NP}$-algorithm are not necessarily enough to obtain a reasonable success probability. In particular, exponentially many calls might be needed to boost the success probability to at least $1/2$. Hence, boosting the success probability does not work to show that $\mathcal{NP} \subseteq \mathcal{RP}$, and most likely, this statement does not hold anyway.

Finally, we remark that the relation between $\mathcal{NP}$ and $\mathcal{BPP}$ is not known.

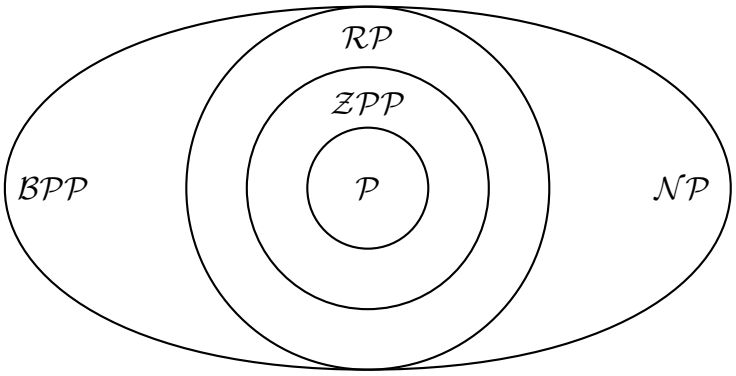


Figure 3.1: The relation of the complexity classes.

4 The Class $\mathcal{NP}$

4.1 Definition and Examples

We repeat the definition from page 27.

Definition 4.1 A yes-no-problem is in $\mathcal{NP}$ if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

$$\begin{array}{lllll} \text{True answer for } \boldsymbol{X} \text{ is} & \text{YES} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{YES}\,] & > \quad 0, \\ \text{True answer for } \boldsymbol{X} \text{ is} & \text{NO} & \text{then} & \boldsymbol{P}_R[\, A(\boldsymbol{X}, R) = \text{NO}\,] & = \quad 1, \end{array}$$

where $\boldsymbol{P}_R[\, Z\,]$ denotes the probability of event Z over uniform distribution of R.

Our intention is to show that this class contains problems which are computationally hard. However, not all problems in $\mathcal{NP}$ are hard; we shall see in the next section how to filter out the really hard ones.

In the following, we present a list of problems and show for each of them that it is in $\mathcal{NP}$. We shall formulate them as "real world" problems. Usually, they are presented in a much more formal (mathematical) way. The goal of our approach is to show that in practice the first – and often the crucial – step towards the solution of a problem is that of *abstraction*. This means that one extracts the abstract structural nature of a real world problem by stripping off all things that are unnecessary for the solution. We shall see examples for this later.

Problem 4.2 [PotatoSoup] A recipe calls for B grams of potatoes. You have a K kilo bag with n potatoes. Can one select some of them such that their weight is exactly B grams?

Problem 4.3 [GlassesInACupboard] You have n glasses of equal height. If glass g_j is put into glass g_i let d_{ij} be the amount of g_j above the rim of g_i. You want to stack them into a single stack, so they fit into a cupboard of height h; is that possible?

Problem 4.4 [MissionToMars] The Danish space agency is planning a manned mission to Mars. There are n astronauts but only $k \leq n$ can go. Every astronaut has some skills (like being a navigator, geologist, astronomer, cook, medical doctor, etc.). There are m skills. Every astronaut can have more than one skill and more than one astronaut can have the same skill. Does there exist a group of at most k astronauts such that all m skills are present in the group?

Problem 4.5 [Sorted] Given is a sequence $a_1, \ldots, a_n$ of n integers. Is the sequence sorted, i. e., is $a_i \leq a_{i+1}$ for $i = 1, \ldots, n-1$?

Problem 4.6 [TravelingSalesperson] A salesperson has customers in n different cities. He or she lives in city number 1. There is direct road between every pair of different cities and no possibility to change roads outside the cities. The distance between city i and city j is $d(i, j) \in \mathbb{N}$. The salesperson wants to visit all cities (in any order) and then return home without visiting another city twice. Is there such a tour whose total length is at most some given value B?

Problem 4.7 [RoadMaintenance] A person is responsible for maintaining roads in some area. There are n cities and roads between some of the cities. There are no possibilities to change roads outside the cities. The person wants to check all roads by starting and ending the tour in city 1. The person does not want to use a road twice but might visit a city more than once. Is such a tour possible?

The next problem which we already saw in Chapter 2 is formulated in an abstract fashion.

Problem 4.8 [SATISFIABILITY]
Input: A set of clauses $C = \{c_1, \ldots, c_k\}$ over n boolean variables $x_1, \ldots, x_n$.
Output: YES is there is a satisfying assignment, i. e., if there is an assignment

$$a\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$$

such that every clause c_j is satisfied, and NO otherwise.

Of the above problems SORTED is obviously solvable in polynomial – even linear – time by a deterministic algorithm. The problems TRAVELING SALESPERSON and ROAD MAINTENANCE seem to be quite similar. Their complexity is, however, very different: While there is an efficient solution algorithm to ROAD MAINTENANCE, the problem TRAVELING SALESPERSON is probably not efficiently solvable.

We shall show for some problems that they are in the class $\mathcal{NP}$ and leave this task as an exercise for some others.

The proofs use so called *guess-verify algorithms*. The random information R (the guess) is interpreted as potential solution. It is then checked (verified) whether it really is a solution. While doing all this, only polynomial time may be used. In particular, even though the random information can potentially be very large, only a polynomially large part of it may be used.

Summarizing: In order to prove that a problem P is in $\mathcal{NP}$, we have to:

1) Design a deterministic algorithm A which takes as input a problem instance $\boldsymbol{X}$ and a random sequence R. Especially:
   - 1a) Specify what the random sequence R consists of: Bits, integers in some **finite** range, etc.
   - 1b) Specify how A interprets R as a guess.
   - 1c) Specify how A verifies the guess.
2) Show that the two conditions are met:
   - 2a) If the answer to $\boldsymbol{X}$ is YES, then there is a string R_0 with positive probability such that $A(\boldsymbol{X}, R_0) = \text{YES}$.
   - 2b) If the answer to $\boldsymbol{X}$ is NO, then $A(\boldsymbol{X}, R) = \text{NO}$ for all R.
3) Show that A is p-bounded for some polynomial p.

The order of some of the task just mentioned can be changed. Often is is convenient to treat 1b) and 1c) together. Moreover, as a part of the proof that A is p-bounded, it is convenient to specify already in 1b) how much random information is used.

Claim 4.9 POTATO SOUP *is in* $\mathcal{NP}$.

Proof. We have to show that there is a polynomial p and a randomized p-bounded algorithm A which satisfies the two conditions of Definition 3.15. In the following, we describe a **possible** such algorithm.

1) Let $w_1, \ldots, w_n$ be the weights of the potatoes and let B be the desired weight. Let p be the polynomial $p(n) = 10n$. Hence, the first part of the input is $\boldsymbol{X} = ((w_1, \ldots, w_n), B)$. Let the input size be n.

1a) The random string R consists of random bits.

1b) Now, on input $(((w_1, \ldots, w_n), B), R)$ algorithm A checks whether R contains at least n bits. If R is shorter, then A returns NO. Otherwise A uses the first n random bits $r_1, \ldots, r_n$ of R to select some potatoes: If $r_i = 1$ then the i-th potato is selected, otherwise it is not.

1c) The total weight of the selected potatoes is computed and compared to the desired value B. If the two are equal, the answer is YES, and NO otherwise. Formally:

$$A\left(((w_1, \ldots, w_n), B), R\right) = \begin{cases} \text{YES}, & \text{if} \quad |R| \geq n \ \wedge \ \sum_{i=1}^{n} r_i w_i = B \\ \text{NO} & \text{otherwise} \end{cases}$$

Now we check that A meets the requirements of Definition 3.15.

3) First, the running time of A is bounded by p: A checks whether R has length at least n by looking at the first up to n bits. This requires n steps (plus some constant overhead). If there are not enough bits, then A terminates with NO in additional constant time. This results in time n (plus some constant overhead). If there are n random bits, then the sum $\sum_{i=1}^{n} r_i w_i$ of the products $r_i w_i$ is computed in time $2n$, again plus some constant overhead. Finally the sum is compared to B and – depending on the result of the comparison – YES or NO is returned. This gives another constant time. This results in time $n + 2n$ (plus some constant overhead). The running time does not exceed $3n$ (plus some constant overhead), which is a polynomial in n. Let us say in this example that the constant overhead is at most 7. Hence, we can bound the polynomial by the even simpler polynomial $p(n) = 10n$.

2) The next step now is to establish the two conditions of Definition 3.15.

2a) Let us first assume that the true answer is YES, i. e., there is a subset of the potatoes weighing exactly B g. Let $L \subseteq \{1, \ldots, n\}$ be the set of (indices of) these

potatoes. Consider the random bit string $R_0 = r_1 \cdots r_n$ of length n which has a one on exactly the positions in L, i. e., defined by

$$r_i = 1 \iff i \in L \ .$$

If R_0 is given to A then $\sum_{i=1}^{n} r_i w_i = B$ and A will correctly return YES. Hence, there is a string of length at most n which leads to a correct answer YES. As there are exactly $2^{n+1} - 1$ strings of length at most n (including the empty string), only a finite number of strings is considered. Hence, the probability of R_0 to be generated by a uniform random choice from this set of $2^{n+1} - 1$ strings is (small but) positive. Here it is important that only a polynomially large amount of random information may be used. Hence, the first condition of Definition 3.15 is satisfied.

2b) Now, assume that the true answer is NO, i. e., there is no subset of the potatoes weighing exactly B g. Then no sum of the form $\sum_{i=1}^{n} r_i w_i$ can ever be equal to B, regardless of the random string R. The algorithm will always output NO in this case, whence the second condition of Definition 3.15 is satisfied. This completes the proof. □

The proof just given is very detailed. In the future, we will, e. g., not specify the polynomial p in advance. Instead will pick the polynomial "which comes out of the analysis". Also we will not bother to give the precise constants, i. e., we will use $O(n^3)$ instead of $(1/2)n^3 + 4n^2 - 2n + 7$.

What we can learn from the last proof is that the crucial step is an appropriate way to use the random bits. One interprets them to code a possible solution, and then the algorithm merely checks whether this is a solution. Above, we interpreted the random bits to code a subset of the potatoes. We present another example for this approach, however, we give it less detailed than the previous one.

Claim 4.10 Traveling Salesperson *is in* $\mathcal{NP}$.

Proof. We want to use the random string to "guess" the order in which the cities are visited. Let n be the number of the cities. We describe an algorithm A satisfying the requirements of Definition 3.15. Let $(\boldsymbol{X}, R)$ be the input, $\boldsymbol{X}$ coding the problem (e. g., as a list of the $d(i, j)$ and the value of B) and R is a random list of integers in the range $\{1, \ldots, n\}$. If R is shorter than n then A returns NO. Otherwise let z_i denote the i-th integer from R. Now z_i is interpreted the number of the i-th city visited on our tour. In order to have a legal tour, A checks the following:

1. $z_1 = 1$, start in city 1.
2. $\forall i \colon 1 \leq z_i \leq n$, only numbers of existing cities.

3. For i, j with $i \neq j$ we have $z_i \neq z_j$, every city is visited exactly once.

If this check fails, A returns NO. Otherwise, the sequence $z_1, z_2, \ldots, z_n$ describes a legal tour. In this case, we compute the cost C of the tour by

$$C = d(z_n, z_1) + \sum_{i=1}^{n-1} d(z_i, z_{i+1}) \ .$$

Then we compare this with the desired bound: If $C \leq B$ then the answer is YES and NO otherwise.

Going through the above description, one can see that the algorithm runs in time $O(n^2)$, the time needed to read the input. If there is a tour with cost at most B then there is a string R_0 that codes this tour, whence the first condition of the definition is met. If no such tour exists, no string can ever code one and the answer is NO. □

4.2 $\mathcal{NP}$-completeness and Cook's Theorem

The example problems of previous section are not all hard. As we saw, SORTED can be solved in linear time. This is also true for ROAD MAINTENANCE. Actually ROAD MAINTENANCE is a re-formulation of the well-known graph-theoretical problem *Euler cycle*. This is the problem, given an undirected graph $G = (V, E)$, whether it contains an Euler cycle, i. e., a sequence $e_1, \ldots, e_m$ of edges containing every edge exactly once and for $i = 1, \ldots, m-1$ the edges e_i, and e_{i+1} share a node as do e_1, and e_m. Euler (1707–1783) has shown in 1735 that such a cycle exists iff every node in the graph has even degree. This can be checked in time $O(|V| + |E|)$. For the other problems no efficient solutions are known by now, although an enormous effort has been put into them. It is thus believed that these problems are computationally hard.

As membership in $\mathcal{NP}$ does not suffice to establish hardness, we shall now introduce a subclass of $\mathcal{NP}$ which consists entirely of problems believed to be hard. To this end, we want to make precise what it means that one problem is "harder" than another one. Informally, problem P_2 is harder than problem P_1 if an algorithm A_2 for solving P_2 can also be used to solve P_1. This is done by transforming an input $\boldsymbol{X}_1$ for P_1 into an input $\boldsymbol{X}_2$ for P_2, then using A_2 to compute the solution s_2 on $\boldsymbol{X}_2$, and finally transforming s_2 back into a solution s_1 for $\boldsymbol{X}_1$. This "reduces" solving P_1 to solving P_2. In order that this really specifies the concept of "being harder", we have to avoid that the real work for the solution is done in the transformations. We shall thus limit the computational power of the input transformation algorithm T in the following formal definitions. As we are considering yes-no-problems, the solutions s_1 and s_2 are from $\{\text{YES}, \text{NO}\}$, and we require the back-transform to be the identity, i. e., $s_1 = \text{YES} \iff s_2 = \text{YES}$.

Definition 4.11 Let P_1 and P_2 be yes-no-problems. Let Σ_1 and Σ_2 be the respective input alphabets. We identify the problems with the languages, i. e., $P_i \subseteq \Sigma_i^*$. We say that P_1 is *polynomial-time reducible* to P_2 (*reducible*, for short) if there is a polynomially bounded **deterministic** algorithm $T\colon \Sigma_1^* \to \Sigma_2^*$ such that

$$\forall \boldsymbol{X}_1 \in \Sigma_1^*\colon \boldsymbol{X}_1 \in P_1 \iff T(\boldsymbol{X}_1) \in P_2 \ . \tag{4.1}$$

We write $P_1 \leq_p P_2$ if P_1 is reducible to P_2.

Let us review the definition briefly. Algorithm T transforms input $\boldsymbol{X}_1$ for P_1 into an input $\boldsymbol{X}_2 := T(\boldsymbol{X}_1)$ for P_2. Now suppose that there is an algorithm A_2 for solving P_2, i. e., for checking whether $\boldsymbol{X} \in P_2$. Then, by the condition (4.1), the answer it computes for $\boldsymbol{X}_2$ is also the answer for $\boldsymbol{X}_1$ and problem P_1. Hence, applying A_2 to $T(\boldsymbol{X}_1)$ is an algorithm A_1 for P_1:

$$A_1(\boldsymbol{X}_1) = A_2\left(T(\boldsymbol{X}_1)\right) \ .$$

We state three important facts in the next claims.

Claim 4.12 *If $P_1 \leq_p P_2$ and P_2 is efficiently solvable then so is P_1.*

Proof. Let A_2 be an efficient deterministic algorithm for P_2. Let p be a polynomial bounding its running time. Let T be a polynomial-time deterministic transformation, with running time bounded by some polynomial q. If T is run on $\boldsymbol{X}_1 \in \Sigma_1$ and $|\boldsymbol{X}_1| = n$ then for the output $\boldsymbol{X}_2 = T(\boldsymbol{X}_1)$ we have $|\boldsymbol{X}_2| \leq q(n)$. This is because writing an output bit counts one unit of time. Then the composition $A_2 \circ T$ is an efficient algorithm with running time bounded by $p(q(n))$, which is again a polynomial. □

Claim 4.12 expresses that problem P_2 is "at least as hard" as P_1. The next claim is just the contraposition of the previous one.

Claim 4.13 *If $P_1 \leq_p P_2$ and P_1 is not efficiently solvable then P_2 is also not efficiently solvable.*

Claim 4.14 *The relation $\leq_p$ is transitive, i. e., for all $P, R, S \in \mathcal{NP}$ we have*

$$((P \leq_p R) \wedge (R \leq_p S)) \implies (P \leq_p S) \ . \tag{4.2}$$

The proof is left as an exercise.

We now formulate the central definition.

Definition 4.15 A problem P is called $\mathcal{NP}$-*complete* if

$$P \in \mathcal{NP} \ , \tag{4.3}$$

$$\forall P' \in \mathcal{NP} \colon P' \leq_p P \ . \tag{4.4}$$

Condition (4.4) expresses that problem P is at least as hard as **any** problem in $\mathcal{NP}$. An algorithm for solving P can be converted into an algorithm for solving any problem in $\mathcal{NP}$ with only a polynomial increase in time. This also explains the name "complete": Problem P has "complete information" on all of $\mathcal{NP}$.

The important thing about $\mathcal{NP}$-complete problems is that they are **believed** to be computationally hard in the sense that they do not have deterministic polynomial-time algorithms. This would mean that the classes $\mathcal{P}$ and $\mathcal{NP}$ are different. The belief is that for an $\mathcal{NP}$-complete problem there is basically only one way of solving it: One has to try out all potential solutions and check whether at least one really solves the problem. This is reflected in the definition of $\mathcal{NP}$ where we only demand that for a YES-input "exists" one random string R such that the correct answer is found. As the complete problems have information on all $\mathcal{NP}$ problems they are believed to be the hardest.

Unfortunately there is **no proof** of this belief yet. Computer scientists have tried to prove that the $\mathcal{NP}$-complete problems cannot be solved in polynomial time for more than 40 years. Without success. Hence there still is a chance that there are efficient algorithms for solving the $\mathcal{NP}$-complete problems. In fact, it would suffice to find a polynomial time deterministic algorithms for only one $\mathcal{NP}$-complete problem. Then one could construct efficient algorithms for all other $\mathcal{NP}$-complete problem through the polynomial time reduction and have $\mathcal{P} = \mathcal{NP}$. The fact that no such efficient algorithm has be found in more than 40 years – although the best computer scientist invested a lot of time to do so – is a very strong indication that no such algorithm exists, but it is not a proof.

Why is that so hard to prove? Well, one has to show that no $\mathcal{NP}$-complete problem has an efficient deterministic algorithm. By the definition of $\mathcal{NP}$-completeness, it suffices to show this for one problem P. That is, one has to show that all algorithms will be super-polynomial for infinitely many inputs for P. There is not yet any technique known to prove such a statement about **all** algorithms. The difficulty is that there are infinitely many, arbitrarily weird algorithms and a proof would have to show for **all** of them that they do not always solve the problem in polynomial time.

The problem just discussed is often called the $\mathcal{P}$-$\mathcal{NP}$-problem or the $\mathcal{P} \neq \mathcal{NP}$-problem. From time to time, people announce that they found a solution in one or the other direction but until today all the arguments have been shown to be false.

If one wants to prove that a problem P is $\mathcal{NP}$-complete then Definition 4.15 demands to prove relation (4.4) for every $P' \in \mathcal{NP}$. As there are infinitely many problems in

$\mathcal{NP}$ this cannot be done. However, claim 4.14 provides a practical approach to prove the $\mathcal{NP}$-completeness of a problem P. We pick for P' in Condition 4.4 a problem P_c, which is not only in $\mathcal{NP}$ but one that is itself already $\mathcal{NP}$-**complete**. Then we prove $P_c \leq_p P$ for this single problem P_c. As P_c is $\mathcal{NP}$-complete, we know that $P' \leq_p P_c$ for all $P' \in \mathcal{NP}$. Now, by the transitivity of the relation $\leq_p$, we have $P' \leq_p P$ for all $P' \in \mathcal{NP}$. As this approach is fundamental for the rest of the chapter, we summarize the steps.

Algorithm 4.16 [Steps to prove that a problem P is $\mathcal{NP}$-complete]

1) Prove that $P \in \mathcal{NP}$.
2) Find a suitable problem P_c which is known to be $\mathcal{NP}$-complete.
3) Prove $P_c \leq_p P$, especially:
   - 3a) Describe a transformation T which transforms every instance $\boldsymbol{X}$ of P_c into an instance $T(\boldsymbol{X})$ of P and which runs polynomial in the size $|\boldsymbol{X}|$ of $\boldsymbol{X}$.
   - 3b) Show that if the answer to $\boldsymbol{X}$ is YES then so is the answer to $T(\boldsymbol{X})$.
   - 3c) Show that if the answer to $\boldsymbol{X}$ is NO then so is the answer to $T(\boldsymbol{X})$. Alternatively, and equivalently, show that if the answer to $T(\boldsymbol{X})$ is YES then so is the answer to $\boldsymbol{X}$.

Step 1 is mostly done by guessing a solution and verifying it. We saw examples for this in Section 4.1.

Step 2 is a matter of experience. Often the problem P_c is chosen to be similar to P in some sense. Sometimes, however, P_c is a completely different type of problem.

Step 3 is guided by the choice of problem P_c. It basically consists of establishing the deterministic, polynomial-time transformation T of the input. Step 3c) is described in two variants that are equivalent. Sometimes it is easier to prove the implication $T(\boldsymbol{X}_1) \in P_2 \Rightarrow \boldsymbol{X}_1 \in P_1$ contained in Definition 4.1 directly, sometimes the so-called contraposition $\boldsymbol{X}_1 \notin P_1 \Rightarrow T(\boldsymbol{X}_1) \notin P_2$ is easier to prove.

In order for this approach to work, we have to know at least one $\mathcal{NP}$-complete problem P_c. In 1971, Cook proved that the satisfiability problem SAT is $\mathcal{NP}$-complete.

Theorem 4.17 (Cook's Theorem) SAT *is $\mathcal{NP}$-complete.*

Proof. We only sketch the proof here. For a formal proof, it is helpful to severely restrict the set of instructions of an algorithm to very few and also to model the memory used during the computation. For example, it is wise not allow indirect addressing. Generally, this is done by introducing the *Turing machine* as a computational model. A complete proof can be found in the book by Garey and Johnson, [GJ79]. Methods for proving membership in $\mathcal{NP}$ were presented in the last section. They can be used to show that SAT is in $\mathcal{NP}$, thus establishing condition (4.3).

It remains to prove condition (4.4), i. e., that every problem $P' \in \mathcal{NP}$ is reducible to SAT. The proof idea is the following: Let P' be an arbitrary problem in $\mathcal{NP}$. Let p be a polynomial and let $A_{P'}$ be a p-bounded randomized algorithm for P'. We assume that $A_{P'}$ is given in some syntactical form (as pseudo code). The code for $A_{P'}$ has a fixed finite length independent of the input. Let ℓ be the number of instructions. As the algorithm is p-bounded, it cannot use more than $p(|\boldsymbol{X}|)$ memory on input $(\boldsymbol{X}, R)$. We also assume that every memory cell can hold only values from some finite set, say $\{0, 1, \ldots, k\}$.

The transformation algorithm T transforms the input $(\boldsymbol{X}, R)$ into a set of clauses. These clauses describe the computation of $A_{P'}$ on $\boldsymbol{X}$ using the random string R. They certify that at every time t, $0 \leq t \leq p(|\boldsymbol{X}|)$, the computation is correct. The boolean variables used have a semantical meaning. For example, there are variables $Q(t, j)$, for $1 \leq j \leq \ell$, $0 \leq t \leq p(|\boldsymbol{X}|)$. Then $Q(t, j)$ is true iff at time t the j-th instruction is executed. Another group of variables are denoted by $M(t, i, c)$ $1 \leq i \leq \ell$, $0 \leq t \leq p(|\boldsymbol{X}|)$, $0 \leq c \leq k$. Then $M(t, i, c)$ is true iff at time t the i-th memory cell contains value c.

Now some clauses establish that at each time t at least one instruction is executed. The following $p(|\boldsymbol{X}|) + 1$ clauses do this:

$$Q(t, 1) \vee Q(t, 2) \vee \cdots \vee Q(t, k), \qquad 0 \leq t \leq p(|\boldsymbol{X}|) \ .$$

Other formulas establish that at most one instruction is executed at every time t, by saying that not two instructions j and j' are executed at time t:

$$\neg(Q(t, j) \wedge Q(t, j')), \qquad 0 \leq t \leq p(|\boldsymbol{X}|),\ 1 \leq j < j' \leq k \ .$$

These are not clauses but can be made into clauses by deMorgan's laws:

$$(\neg Q(t, j) \vee \neg Q(t, j')), \qquad 0 \leq t \leq p(|\boldsymbol{X}|),\ 1 \leq j < j' \leq k \ .$$

Another group of instructions ensures that the memory is changed correctly. Assume that the j-th instruction writes a value c' into memory cell i. If the content of memory cell i is c at time t and instruction j is executed at time t then memory cell i contains c' at time $t + 1$:

$$[M(t, i, c) \wedge Q(t, j)] \implies M(t + 1, i, c') \ .$$

Again this is not a clause but can be made into one using the identity $(A \implies B) \iff (\neg A \vee B)$.

$$[\neg M(t, i, c) \vee \neg Q(t, j)] \vee M(t + 1, i, c') \ .$$

Let us remark that especially at this point the instruction set of the computational model has to be carefully selected in a formal proof. The less powerful the individual instructions are, the easier and shorter the clauses are. On the other hand, the number of clauses will grow, as more of the simple instructions are in the algorithm.

There are more clauses ensuring that reading from a memory cell, starting of the algorithm etc. are correct. There is one special clause *OUTYES* which says that the algorithm outputs a YES when it terminates. One can show that the number and length of the clauses are bounded by a polynomial q in $|\boldsymbol{X}|$. (The number of instructions ℓ and of different memory values k are constants.) Also the clauses can be constructed in time polynomial in $|\boldsymbol{X}|$. This concludes the sketch of the transformation algorithm.

The other thing one can show is that there is an assignment of truth values to the variables if and only if the the correct answer on the input $\boldsymbol{X}$ is YES, i. e., if $\boldsymbol{X} \in P'$. This is true because a legal computation of $A_{P'}$ is reflected in the clauses and vice versa. If the computation outputs a YES then also the special clause *OUTYES* is satisfied.

Let A_{SAT} be an algorithm for the satisfiability problem. Let $\boldsymbol{X}$ be an input for the problem P', and let $T(\boldsymbol{X})$ be the set of clauses constructed from $\boldsymbol{X}$ by the transformation sketched above. If and only if A_{SAT} finds that there is a satisfying assignment for $T(\boldsymbol{X})$ (i. e., if A_{SAT} computes YES) then there is a legal computation of $A_{P'}$ on $\boldsymbol{X}$. Hence,

$$\forall \boldsymbol{X} : \boldsymbol{X} \in P' \iff T(\boldsymbol{X}) \in \text{SAT} ,$$

which is condition (4.4). □

4.3 Techniques for Proving $\mathcal{NP}$-completeness

In this section, we present some techniques for proving the $\mathcal{NP}$-completeness of a problem. Especially, we present techniques for selecting the right problem and for constructing the reduction. Most of the methods can be put into one of the following categories: *component design*, *local replacement* and *restriction*. Sometimes, a simple reformulation reveals that the problem is already known under a different name; in this case we speak of *modelling*. Typically, proofs using component design are more difficult than proofs using local replacement, and proofs using restriction or modelling are even less involved. Examples for the different techniques are given later in individual proofs, and are summarized in the end of this chapter (Table 4.1).

All our proofs follow the scheme given in 4.16. We shall also define some more problems and show that they are $\mathcal{NP}$-complete. The problems come from different areas such as Logic, Graph Theory, and Optimization Theory.

The first problem is a logical one, in fact a variant of SATISFIABILITY. Changing the definition of $\mathcal{NP}$-complete problem a bit can make the problem easy, i. e., efficiently

solvable, but the problem might remain hard as well. On the other hand there are efficiently solvable problems which become $\mathcal{NP}$-complete if they are slightly changed.

In the following, the notion *literal* means a negated or an unnegated variable. For example, $\overline{x_1}$ and x_1 are the two literals belonging to the variable x_1.

Problem 4.18 [3-SATISFIABILITY]
Input: A set of clauses $C = \{c_1, \ldots, c_k\}$ over n boolean variables $x_1, \ldots, x_n$, where every clause contains exactly three literals.
Output: YES if there is a satisfying assignment, i. e., if there is an assignment

$$a\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$$

such that every clause c_j is satisfied, and NO otherwise.

Theorem 4.19 *The problem* 3-SATISFIABILITY *is $\mathcal{NP}$-complete.*

Proof. We leave it to the reader to show that 3-SATISFIABILITY is in $\mathcal{NP}$.

The proof of the completeness is by local replacement. The reference problem is of course SATISFIABILITY because we do not know any other $\mathcal{NP}$-complete problem by now. We shall now show that

$$\text{SATISFIABILITY} \leq_p \text{3-SATISFIABILITY} .$$

We do this by first defining an algorithm T for transforming inputs for SATISFIABILITY into inputs for 3-SATISFIABILITY and then show that, with this transformation, the Condition (4.1) holds. Now let $C = \{c_1, \ldots, c_k\}$ be a set of clauses over n boolean variables $x_1, \ldots, x_n$, with no restriction on the length of the clauses. Transformation T constructs a set $C' = \{c'_1, \ldots, c'_\ell\}$ of clauses of length 3. We then show that if there is satisfying assignment for C there is one for C', and vice versa in order to establish (4.1).

We shall "break" long clauses c_j into many clauses $c'_{j1}, c'_{j2,\ldots}$ of length three. However, the c'_{j1} have to be linked in order to have the same satisfiability property as c_j. To this end, we introduce new boolean variables, which we denote by a y with an index. Every original clause c_j is treated individually and has its own new variables. Let $c_j = z_1 \vee z_2 \vee \cdots \vee z_m$ be a clause, where the z_i are literals. We consider four cases depending on the number m of literals.

Case 1: $m = 1$**.** Then $c_j = z_1$. We have to have length three. We take two new variables $y_{j,1}$ and $y_{j,2}$ and define four new clauses:

$$c'_{j1} = z_1 \vee y_{j,1} \vee y_{j,2}, \quad c'_{j2} = z_1 \vee \bar{y}_{j,1} \vee y_{j,2}, \quad c'_{j3} = z_1 \vee y_{j,1} \vee \bar{y}_{j,2}, \quad c'_{j4} = z_1 \vee \bar{y}_{j,1} \vee \bar{y}_{j,2} \, .$$

Case 2: $m = 2$**.** Then $c_j = z_1 \vee z_2$. We take one new variable $y_{j,1}$ and define two new clauses:

$$c'_{j1} = z_1 \vee z_2 \vee y_{j,1}, \quad c'_{j2} = z_1 \vee z_2 \vee \bar{y}_{j,1} \, .$$

Case 3: $m = 3$**.** Then $c_j = z_1 \vee z_2 \vee z_3$. The c_j has length three already, nothing is to do.

Case 4: $m > 3$**.** Then $c_j = z_1 \vee z_2 \vee \cdots \vee z_m$. We take $m - 3$ new variables $y_{j,1}, y_{j,2}, \ldots, y_{j,m-3}$, and define $m - 2$ clauses:

$$\begin{array}{lcllllll}
c'_{j,1} & = & z_1 & \vee & z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \bar{y}_{j,1} & \vee & z_3 & \vee & y_{j,2} \\
c'_{j,3} & = & \bar{y}_{j,2} & \vee & z_4 & \vee & y_{j,3} \\
c'_{j,4} & = & \bar{y}_{j,3} & \vee & z_5 & \vee & y_{j,4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,i} & = & \bar{y}_{j,i-1} & \vee & z_{i+1} & \vee & y_{j,i} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \bar{y}_{j,m-4} & \vee & z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \bar{y}_{j,m-3} & \vee & z_{m-1} & \vee & z_m \, .
\end{array}$$

It is left as an exercise to the reader to prove that this transformation can be computed in time polynomial in $|C|$.

Now suppose that the original set C of clauses is satisfiable, i. e., there is an assignment of truth values to the boolean variables $x_1, \ldots, x_n$ such that every clause $c_j \in C$ is satisfied. We claim that then there exists an assignment of truth values to the old boolean variables $x_1, \ldots, x_n$ and the new variables y_{ji} such that all clauses in the transformed set C' are satisfied. Consider the clause $c_j = z_1 \vee z_2 \vee \ldots \vee z_m$. As the clause is satisfied, at least one z_i is **true**. If $m \leq 3$ then we can check by inspecting the Cases 1, 2, and 3 that all clauses c' defined there are satisfied. If $m > 3$ then we are in Case 4. If $z_1 =$ **true** or $z_2 =$ **true** then the clause $c'_{j,1}$ is satisfied. In order to satisfy the remaining clauses $c'_{j,2}$ through $c'_{j,m-2}$, we assign **false** to $y_{j,1}$ through $y_{j,m-3}$. Then the first literals $\bar{y}_{j,1}$ through $\bar{y}_{j,m-3}$ of these clauses become **true**. A similar argument holds if at least one of z_{m-1} or z_m is assigned **true**. So, let $z_i =$ **true**, for some i, $2 < i < m - 1$. Then the clause

$$c'_{j,i-1} = \bar{y}_{j,i-2} \vee z_i \vee y_{j,i-1}$$

is satisfied. Hence, we are free to choose assignments for $y_{j,i-2}$ and $y_{j,i-1}$. Actually, we set $y_{j,k} =$ **true** for $k = 1, 2, \ldots, i - 2$, thus satisfying $c'_{j,1}$ through $c'_{j,i-2}$. By setting

$y_{j,k} = \textbf{false}$ for $k = i-1, i, \ldots, m-2$ (i. e., $\bar{y}_{j,k} = \textbf{true}$), the clauses c_i through c_{m-2} are satisfied. This can be done for all original clauses c_j, hence, we constructed a satisfying assignment for the transformed set C'.

For the converse, let us assume that there is an assignment α' of truth values to the variables x_i and $y_{j,k}$ satisfying all clauses in the transformed set C'. We claim that then there is also an assignment of truth values to the variables x_i satisfying all clauses in C. Again consider the clause $c_j = z_1 \vee z_2 \vee \ldots \vee z_m$. In Cases 1, 2, and 3, any assignment satisfying the transformed set of clauses must also satisfy c_j. We only present Case 2 here: An assignment satisfying both

$$c'_{j,1} = z_1 \vee z_2 \vee y_{j,1} \quad \text{and} \quad c'_{j,2} = z_1 \vee z_2 \vee \bar{y}_{j,1}$$

has to make at least one of z_1 or z_2 **true**. Namely, if $z_1 = z_2 = \textbf{false}$ then setting $y_{j,1} = \textbf{true}$ would make $c'_{j,2}$ **false**, while setting $y_{j,1} = \textbf{false}$ would make $c'_{j,1}$ **false**. Finally, consider Case 4 ($m > 4$). We show that any assignment satisfying $c'_{j,1}$ through $c'_{j,m-2}$ also sets at least one literal z_i, $1 \leq i \leq m$, to **true**. Assume that the assignment sets $z_1 = z_2 = \cdots = z_m = \textbf{false}$. Then $y_{j,1} = \textbf{true}$ in order to satisfy clause $c'_{j,1}$. But then $\bar{y}_{j,1} = \textbf{false}$ and – as $z_3 = \textbf{false}$ – we must have $y_{j,2} = \textbf{true}$ to satisfy $c'_{j,2}$. Iterating this argument, we see that it must be that all y-variables are true: $y_{j,1} = y_{j,2} = \cdots = y_{j,m-3} = \textbf{true}$. But then in last clause $c'_{j,m-2} = \bar{y}_{j,m-3} \vee z_{m-1} \vee z_m$ all three literals are **false** and the clause is not satisfied, contrary to our assumption that we have a satisfying assignment for the set C'. Hence at least one z_i is **true** and c_j is satisfied. As this holds for all j, every clause c_j in the set C is satisfied. □

Now that we know that 3-SATISFIABILITY is $\mathcal{NP}$-complete, we can use it as a reference problem in other reductions. The following reductions will be described less detailed than the first one.

We first consider a subset problem and prove that it is $\mathcal{NP}$-complete. The problem asks for a subset of a set of natural numbers such that the sum of the numbers in the subset equals a given value.

Problem 4.20 [SUBSETSUM]
Input: Natural numbers $s_1, \ldots, s_n$ and B.
Output: YES if there is a set $A \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in A} s_i = B \ , \tag{4.5}$$

and NO otherwise.

Theorem 4.21 *The problem* SUBSETSUM *is $\mathcal{NP}$-complete.*

Proof. The proof is by component design, using 3-SATISFIABILITY as a reference problem. To show that SUBSETSUM is in $\mathcal{NP}$, we randomly select a subset $A \subseteq \{1, \ldots, n\}$ and check that it satisfies (4.5). If a subset exists that satisfies this condition, there is positive chance of guessing it, otherwise we cannot guess such a subset. Checking (4.5) can be done in polynomial time.

We shall now describe how to transform an input for 3-SATISFIABILITY to one for SUBSETSUM in such a way that a subset A satisfying (4.5) exists if and only if the clauses have a satisfying assignment. To show the "if and only if" equivalence, we follow the two Steps 3b) and 3c) of Algorithm 4.16. First, we show that if there is a satisfying assignment to the clauses, i. e., the answer is YES for 3-SATISFIABILITY, then an appropriate subset A can be found, i. e., the answer is also YES for SUBSETSUM. Second, we proceed with the NO-case and show that if there is *no* satisfying assignment then there is *no* appropriate subset A.

Let $\{x_1, x_2, \ldots, x_n\}$ be the set of Boolean variables and let $\{c_1, c_2, \ldots, c_m\}$ be the set of clauses with three literals each. The main idea of the following construction is to map the components of the 3-SATISFIABILITY instance to specific locations of numbers that become part of the input for the SUBSETSUM problem. For this reason, the numbers will have so many decimal digits that there is a unique location for each variable and each clause. The structure of satisfying assignments will reappear in a particular way to sum up the numbers with B as result.

We describe the input to SUBSETSUM more precisely. There are $2n + 2m$ numbers, denoted as a_i and b_i, $1 \leq i \leq n$, as well as d_j and e_j, $1 \leq j \leq m$, all of which have $n + m$ decimal digits. We first describe the last n digits, which contain only zeros and ones. Actually, all the last n digits are 0 in the d_j- and e_j-numbers. The numbers a_i and b_i contain exactly one 1-entry, namely at the i-th position within the block of the last n digits. All remaining entries of this block are set to 0. The last n digits of the number B are set completely to 1, see Figure 4.1 for an example. Altogether, for each digit from the block of the last n digits, there are only two numbers (except B) having a 1-entry there, while the rest of the numbers has a 0-entry at the digit. This means that we are forced, for each $i \in \{1, \ldots, n\}$, to choose either a_i or b_i for the subset of numbers that sum up to B. We cannot obtain a 1-entry in one of the last n positions of the sum just by taking d_j- or e_j-numbers, and if a_i and b_i were both chosen, we would obtain a 2 in the relevant position (recall that we are dealing with decimal numbers, hence there cannot be a carry-over).

The interpretation is as follows: If we choose a_i for the subset, this corresponds to the setting $x_i = 1$, and if b_i is chosen instead, this means $x_i = 0$. Up to now, we have made sure that a selection of numbers in the SUBSETSUM problem encodes an unambiguous assignment to the variables in the 3-SATISFIABILITY problem.

	c_1	c_2	c_3	x_1	x_2	x_3	x_4
a_1	1	0	0	1	0	0	0
a_2	0	1	0	0	1	0	0
a_3	1	0	0	0	0	1	0
a_4	0	0	0	0	0	0	1
b_1	0	1	1	1	0	0	0
b_2	1	0	1	0	1	0	0
b_3	0	0	1	0	0	1	0
b_4	0	1	0	0	0	0	1
d_1	1	0	0	0	0	0	0
d_2	0	1	0	0	0	0	0
d_3	0	0	1	0	0	0	0
e_1	1	0	0	0	0	0	0
e_2	0	1	0	0	0	0	0
e_3	0	0	1	0	0	0	0
B	3	3	3	1	1	1	1

Figure 4.1: Example for the reduction 3-SATISFIABILITY $\leq_p$ SUBSETSUM, where $c_1 = x_1 + \overline{x}_2 + x_3$, $c_2 = \overline{x}_1 + x_2 + \overline{x}_4$ and $c_3 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$.

More work is required to identify selections of numbers that satisfy (4.5) with assignment that *also satisfy* the 3-SATISFIABILITY instance. To this end, the first m positions of the numbers a_i, b_j, d_j and e_j become relevant, where each of these positions refers to a specific clause. Let us consider an arbitrary clause $c_j = z_1 \vee z_2 \vee z_3$. For each of the literals z_1, z_2, z_3, we proceed in the same way, hence we describe the approach only for z_1. If $z_1 = x_k$ for an unnegated variable x_k then the number a_k receives a 1-entry at the j-th position within the first m digits. Similarly, if $z_1 = \overline{x}_\ell$ with the variable x_ℓ negated then the number b_ℓ receives a 1-entry at the j-th position within the first m digits. This has the following consequences when we go from an arbitrary assignment of the n variables to the corresponding selection of a_i- and b_i-numbers: Let S denote the sum of the numbers that have been chosen for the subset and let s_j be the j-th within the first m digits of S. Then s_j denotes the number of literals that are set to true in the clause c_j. Hence, all m clauses are satisfied by an assignment if we can choose a subset from the a_i- and b_i-numbers (of course choosing *either* a_i *or* b_i) such that the sum has only entries from $\{1, 2, 3\}$ within the first m digits and only 1-entries within the last n digits.

The last property does still not ensure that the numbers corresponding to a satisfying assignment sum up to a *fixed* value B. To resolve this problem, the d_j- and e_j-numbers come into play as "fill-up" elements. We let d_j, $1 \leq j \leq m$, have a 1-entry at the j-th position from left, i. e., within the block that is relevant for the m clauses; all remaining entries are 0. Moreover, we set $e_j = d_j$ for all $j \in \{1, \ldots, m\}$. Finally, the first m entries of the number B are set to 3. Now, if we have a selection of a_i- and b_i-numbers that sum up to a value in $\{1, 2, 3\}$ in the j-th of the first m digits, we can obtain a value

of exactly 3 there by additionally including d_j and e_j, or only d_j, or none of these in the subset of chosen numbers. Again, no carry-over is possible since there are only 3 literals in each clause and only two fill-up elements per clause.

On the other hand, if we have an assignment that does *not* satisfy at least one clause c_j then we cannot obtain a 3 at the j-th position of the sum of the related numbers chosen for the subset. Namely, the sum of the a_i- and b_i-numbers would be 0 at this position, and we could only obtain at most 2 there by additionally choosing d_j and e_j. This proves Condition (4.1) from the definition of a polynomial-time reduction. Since all numbers can be written down in polynomial time $O((n+m)^2)$, we really have a polynomial-time reduction. □

We now consider a set partition problem that is a special case of the SubsetSum problem and prove that it is still $\mathcal{NP}$-complete. The problem asks for a dichotomy of a set of natural numbers such that the number of both parts have the same sum.

Problem 4.22 [Partition]
Input: Natural numbers $s_1, \ldots, s_n$.
Output: YES if there is a set $A \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in A} s_i = \sum_{i \in \{1,\ldots,n\} \setminus A} s_i \ , \tag{4.6}$$

and NO otherwise.

Theorem 4.23 *The problem* Partition *is $\mathcal{NP}$-complete.*

Proof. The property that Partition is in $\mathcal{NP}$ can be proved with the very same ideas as in the proof of Theorem 4.21. To show the polynomial-time reduction, we also use component design but cope with only two additional components. The reference problem is SubsetSum.

Let $(s_1, \ldots, s_n, B)$ be an input to the SubsetSum problem. We may assume that $B \leq B^*$, where $B^* = s_1 + \cdots + s_n$ is the sum of all numbers. The input for the Partition problem consists of $n + 2$ numbers, namely $s_1, \ldots, s_n$ and the two special numbers $2B^* - B$ as well as $B^* + B$. This is doable is polynomial time. All $n + 2$ numbers together sum up to $s_1 + \cdots + s_n + (2B^* - B) + (B^* + B) = 4B^*$. Hence, if the answer to the Partition problem is YES, the solution corresponds to a subset A whose elements sum up to $2B^*$.

We prove that the answer to the SUBSETSUM instance is YES if and only if this is the case for the PARTITION instance. If the answer is YES for the SUBSETSUM problem, i. e., there is a subset A' of the numbers $s_1, \ldots, s_n$ with sum B, then the set A' together with the special number $2B^* - B$ sums up to $2B^*$ and is a solution corresponding to a YES answer for the PARTITION problem. For the other direction, we convince ourselves in an exercise that a solution corresponding to a YES for the PARTITION problem must assign exactly one of the special numbers $2B^* - B$ and $B^* + B$ to the subset A. Now assume that the answer to the PARTITION problem is in fact YES, i. e., we have a subset A such that the numbers sum up to $2B^*$. If the special number $B^* + B$ instead of $2B^* - B$ is in the subset, we switch our consideration to the numbers that are *not* indexed by A; these sum up to $2B^*$ as well and include $2B^* - B$. This means that the numbers chosen from $s_1, \ldots, s_n$ sum up to B and are a solution showing YES for the SUBSETSUM problem. □

Problem 4.24 [VERTEXCOVER]
Input: An undirected graph $G = (V, E)$ and a natural number B.
Output: YES if there is a set $V' \subseteq V$ such that $|V'| \leq B$ and V' covers the edges, i. e., for all $e = \{v, w\} \in E$: $(v \in V') \vee (w \in V')$. NO, otherwise.

Theorem 4.25 *The problem* VERTEXCOVER *is $\mathcal{NP}$-complete.*

Proof. The proof is by component design. The reference problem is 3-SATISFIABILITY. To prove that VERTEXCOVER is in $\mathcal{NP}$, we interpret the random string R as a guess for the B vertices of a cover and check whether this is the case.

We now show that 3-SATISFIABILITY $\leq_p$ VERTEXCOVER. Let $\boldsymbol{X} = (C, X)$ be an instance of 3-SATISFIABILITY, where $X = \{x_1, \ldots, x_n\}$ is a set of Boolean variables and $C = \{c_1, \ldots, c_m\}$ is a set of clauses over X. The aim is to construct a graph $G = (V, E)$ and a number $k \in \mathbb{N}$ such that G has a vertex cover of size at most k iff $\boldsymbol{X}$ has a satisfying assignment.

We build small "component" graphs that are later connected to form the desired graph G. There are components which ensure a correct setting of the truth values, others which test satisfiability and components to connect them. The vertex set V is defined as follows

$$V = \{x_1, \ldots, x_n\} \cup \{\bar{x}_1, \ldots, \bar{x}_n\} \cup \bigcup_{j=1}^{m} \{a_1(j), a_2(j), a_3(j)\} \ ,$$

where x_i and $\bar{x}_i$, $1 \leq i \leq n$, correspond to the Boolean variables, while $a_1(j)$, $a_2(j)$ and $a_3(j)$ are additional vertices.

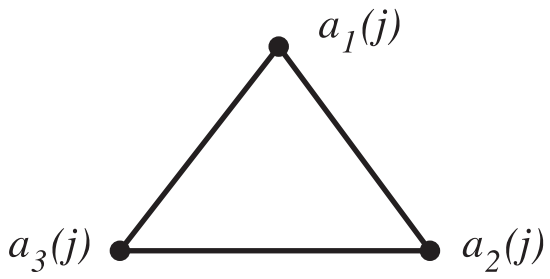
The **truth setting components** are defined to be the edges

$$\forall x_i \in X \text{ let } T_i \text{ be the edge } T_i = \{x_i, \bar{x}_i\} \ .$$

These components ensure that every vertex cover has to contain x_i or $\bar{x}_i$, which will imply that every variable receives a value, as we shall see later.

The **components for clause satisfiability** are defined as follows.

$$\begin{aligned}&\forall c_j \in C\colon \text{ let } S_j \text{ be the set of edges}\\ &S_j = \{\{a_1(j), a_2(j)\}, \{a_2(j), a_3(j)\}, \{a_3(j), a_1(j)\}\} \ .\end{aligned}$$



The **connecting components** are defined as follows. Let $c_j = \ell_1 \vee \ell_2 \vee \ell_3$, where the ℓ_k are literals, i. e., ℓ_k is a boolean variable x_i or a negated boolean variable $\bar{x}_i$. For every c_j the following set of three edges is added

$$K_j = \{\{a_1(j), \ell_1\},\ \{a_2(j), \ell_2\},\ \{a_3(j), \ell_3\}\} \ .$$

Summing up, the edge set E is

$$E = \bigcup_{i=1}^{n} T_i \cup \bigcup_{j=1}^{m} S_j \cup \bigcup_{j=1}^{m} K_j \ .$$

The constant for VertexCover is defined by

$$k = n + 2m \ .$$

The construction of G, i. e., the transformation T, can be performed in time polynomial in n and m.

Example 4.26 Consider the following example with $n = 4$ and $m = 2$:
$X = \{x_1, x_2, x_3, x_4\}$, $C = \{c_1, c_2\}$, $c_1 = x_1 \vee \bar{x}_3 \vee \bar{x}_4$, $c_2 = \bar{x}_1 \vee x_2 \vee \bar{x}_4$. Figure 4.2 shows the resulting graph.

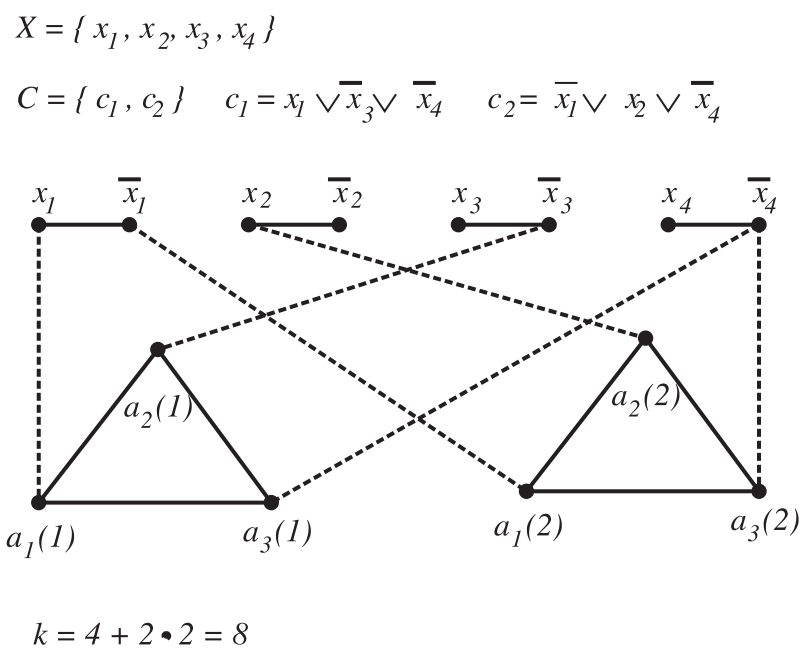


Figure 4.2: The graph for Example 4.26. The components at the top are the T_i, the triangles at the bottom are the S_j. The connecting edges are dashed.

We proceed by showing that $G = (V, E)$ has a vertex cover of size at most k iff the clauses of C have a satisfying assignment.

Let us first assume that $V' \subseteq V$ is a vertex cover for G of size $|V'| \leq k$. In order to cover the edges $T_i = \{x_i, \bar{x}_i\}$, at least one of x_i or $\bar{x}_i$ has to be in V'. In order to cover the edges of S_j, the cover V' has to contain at least two vertices of $a_1(j)$, $a_2(j)$, and $a_3(j)$. Therefore, $|V'| \geq n + 2m = k$, and since $|V'| \leq k$, it follows that $|V'| = k$. Hence, we have that V' contains exactly one vertex from every T_i and exactly two from every S_j.

Therefore, and because V' is a cover, V' contains at least one vertex of every connecting edge K_j. We define an assignment $\alpha\colon X \to \{0, 1\}$ as follows:

$$\alpha(x_i) = \begin{cases} 1 \text{ if } x_i \in V' \,, \\ 0 \text{ if } \bar{x}_i \in V' \,. \end{cases}$$

As V' contains exactly one of x_i or $\bar{x}_i$, the assignment is well defined. It remains to show that assignment α satisfies all clauses c_j. Let $c_j = \ell_1 \vee \ell_2 \vee \ell_3$ be a clause where

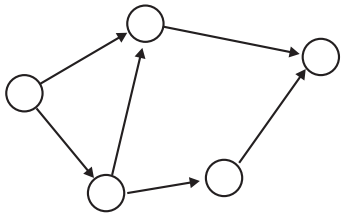


Figure 4.3: A directed graph with no directed Hamilton cycle.

the ℓ_k are literals. For every component S_j, exactly two vertices $a_k(j)$ belong to V', say w. l. o. g. $a_1(j)$ and $a_2(j)$. They cover also the connecting edges $\{a_1(j), \ell_1\}$ and $\{a_2(j), \ell_2\}$. The third connecting edge $\{a_3(j), \ell_3\}$ attached to S_j has thus to be covered by ℓ_3. By construction of G vertex ℓ_3 corresponds to a literal in c_j, by construction of the assignment α, ℓ_3 is set to **true** and clause c_j is thus satisfied.

Now, assume that $\alpha\colon X \to \{0,1\}$ is an assignment which satisfies all clauses c_j. Consider the graph $G = (V, E)$ constructed above. We begin to construct $V' \subseteq V$ by selecting n vertices as follows:

$$\begin{aligned} x_i \in V' &\iff \alpha(x_i) = 1 \ , \\ \bar{x}_i \in V' &\iff \alpha(x_i) = 0 \ . \end{aligned}$$

Then all T_i are covered. Moreover, for every clause c_j at least one connecting edge $\{a_k(j), \ell_k\}$ is covered by an ℓ_k (because some literal in c_j is set to 1). Assume w. l. o. g. that for clause c_j this is ℓ_1. We add $a_2(j)$ and $a_3(j)$ to V'. These two vertices cover the other two connecting edges $\{a_2(j), \ell_2\}$ and $\{a_3(j), \ell_3\}$ and the three edges of the triangle S_j. The set V' covers all edges and has size $k = n + 2m$. This completes the proof. □

An undirected graph $G = (V, E)$, $|V| = n$, has a *Hamilton Cycle*[1] if there is a permutation $v_{i_1}, v_{i_2} \ldots v_{i_n}$ of the nodes such that $\{v_{i_j}, v_{i_{j+1}}\} \in E$ for $j = 1, \ldots, n-1$, and $\{v_{i_n}, v_{i_1}\} \in E$. Informally: One can find a tour visiting all vertices exactly once before returning to the origin.

A **directed** graph $G = (V, E)$, $|V| = n$, has a *Directed Hamilton Cycle* if there is a permutation $v_{i_1}, v_{i_2} \ldots v_{i_n}$ of the nodes such that $(v_{i_j}, v_{i_{j+1}}) \in E$ for $j = 1, \ldots, n-1$ and $(v_{i_n}, v_{i_1}) \in E$. Informally: In addition to the previous definition, the directions of the edges along the cycle must be valid. Figure 4.3 shows an example of a graph not having a directed, but an undirected Hamilton cycle.

[1]Sometimes the problem is called *Hamilton Circuit*

Problem 4.27 [HamiltonCycle]
Input: An undirected graph $G = (V, E)$.
Output: YES if G has a Hamilton cycle and NO otherwise.

Problem 4.28 [DirectedHamiltonCycle]
Input: A directed graph $G = (V, E)$.
Output: YES if G has a directed Hamilton cycle and no otherwise.

Theorem 4.29 *The problem* DirectedHamiltonCycle *is $\mathcal{NP}$-complete.*

Proof. To show that DirectedHamiltonCycle is in $\mathcal{NP}$, we randomly select a sequence of n vertices $v_{i_1}, v_{i_2} \dots v_{i_n}$ and check that it both represents a permutation and satisfies the definition of a directed Hamilton cycle (DHC). If a sequence exists that satisfies these conditions, there is positive chance of guessing it, otherwise we cannot guess a such a sequence. Checking the properties can be done in polynomial time.

The actual polynomial-time reduction works with 3-Satisfiability as the reference problem and proceeds by component design. We now describe how to transform an input for 3-Satisfiability to one for DirectedHamiltonCycle in such a way that a DHC exists if and only if the clauses have a satisfying assignment. Let $\{x_1, x_2, \dots, x_n\}$ be the set of Boolean variables and let $\{c_1, c_2, \dots, c_m\}$ be the set of clauses with three literals each.

The reduction introduces components for variables and clauses (see Figure 4.4). The clause component for c_j, $1 \leq j \leq m$, consists of only one node, which we also call c_j. The components for the variables are more complex. Let $1 \leq i \leq n$. If x_i and $\overline{x_i}$ appear altogether b_i times in clauses, the component for x_i gets exactly $4+3b_i$ nodes. The three nodes $v_{i,1}$, $v_{i,2}$ and $v_{i,3}$ form the base of a component. We have the edges $(v_{i,1}, v_{i,2})$, $(v_{i,1}, v_{i,3})$, $(v_{i,2}, v_{i+1,1})$ and $(v_{i,3}, v_{i+1,1})$, where we count modulo n, i. e., identify $n+1$ with 1. The nodes $v_{i,2}$ and $v_{i,3}$ are connected by a doubly-linked list containing $3b_i + 1$ nodes. In Figure 4.4, a double arrow $u \leftrightarrow w$ denotes the presence of both edges (u, w) and (w, u).

Let us ignore the components for the clauses for a moment. Note that the components for the variables are connected in a circular way. Starting at $v_{1,1}$, we obtain exactly 2^n different directed Hamilton cycles. We can decide at every node $v_{i,1}$ to either go to

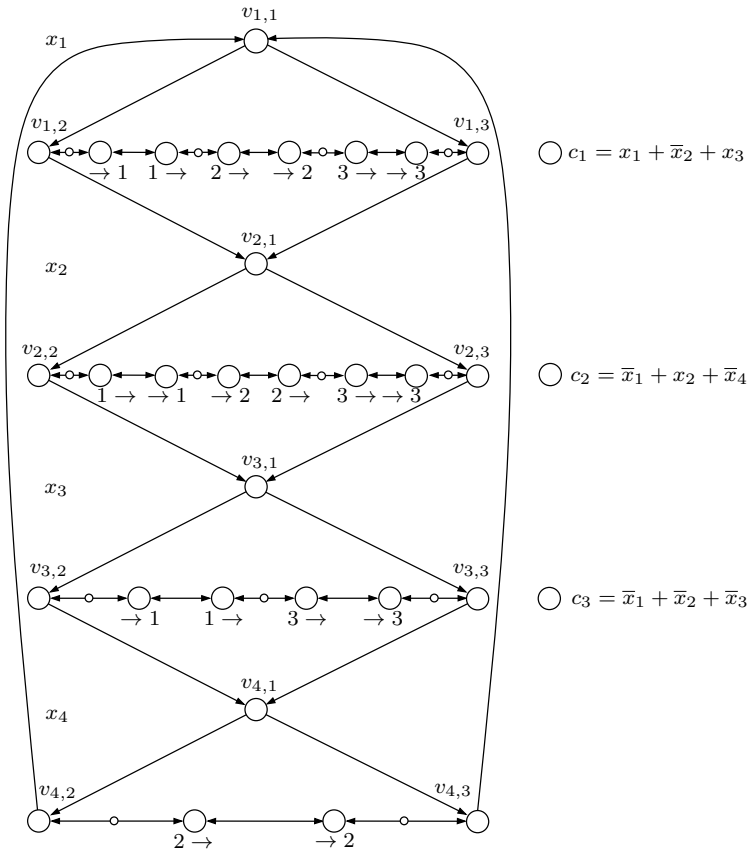


Figure 4.4: An example for the reduction 3-SAT $\leq_p$ DHC. The label "$\to i$" denotes an edge to the node representing c_i while "$i \to$" denotes an edge from the node representing c_i.

$v_{i,2}$, onwards through the doubly-linked list to $v_{i,3}$ and finally to $v_{i+1,1}$, or to do the same with the roles of $v_{i,2}$ and $v_{i,3}$ exchanged. Each of the 2^n Hamilton cycles can be identified with an assignment to the variables. A step from $v_{i,1}$ to $v_{i,2}$ means $x_i = 1$ and a step to $v_{i,3}$ means $x_i = 0$.

It remains to include the m clause components and to connect all components in such a way that satisfying assignments correspond to Hamilton cycles and vice versa. We break up the doubly-linked list connecting $v_{i,2}$ and $v_{i,3}$ into b_i node-disjoint groups consisting of two nodes each. Later, these groups will model whether x_i or $\overline{x_i}$ appears in a clause. The nodes belonging to the groups are depicted by big circles in Figure 4.4. Additionally, there are helper nodes (depicted by small circles, ingoing arrows omitted for the sake of readability) in between the individual groups and between the leftmost group and $v_{i,2}$ as well as the rightmost group and $v_{i,3}$. This amounts to $b_i + 1$ helper nodes. In the following, we concentrate on the $2b_i$ nodes belonging to the b_i groups. Recall that b_i specifies how often x_i or $\overline{x_i}$ appears in a clause. Hence, we consider a group for every appearance.

If x_i appears in c_j, i. e., the variable appears positively, we introduce an edge "$\to j$" from the left node of the respective group to the node c_j, i. e., the clause component,

and an edge "$j \rightarrow$" from c_j to the right node of the group. Analogously, if $\overline{x_i}$ appears in c_j, i. e., the variable appears negatively, we introduce an edge "$j \rightarrow$" from c_j to the left node of the group and an edge "$\rightarrow j$" from the right node of the group to c_j. The whole construction is obviously doable in polynomial time with respect to n and m.

We finish the proof by showing that the constructed graph has a DHC if and only if the 3-Satisfiability instance is satisfiable. Let us first assume that there is a satisfying assignment. Then we consider for every clause a satisfied literal and, thereby, a group in the doubly-linked list for the corresponding variable. If the literal is x_i, the cycle goes from $v_{i,1}$ to $v_{i,2}$; if it is $\overline{x_i}$, it goes to $v_{i,3}$. In the first case, we traverse the doubly-linked list from the left and reach all nodes belonging to the groups from the left, otherwise from the right. If the literal appears positively, there is by construction an edge "$\rightarrow j$" from the left node of the group to c_j and an edge "$j \rightarrow$" back to the right node of the group. Using this "excursion", we integrate c_j in the Hamilton cycle. Analogously, if the literal is $\overline{x_i}$, we go from the right through the list. Then we can, following the construction, use the edge "$\rightarrow j$" from the right node of the group via c_j and onwards to the left node. Since there is at least one satisfied literal in every clause, we can integrate all clause components into the Hamilton cycle and obtain a DHC on the whole graph.

Finally, let us assume that there is a DHC on the whole graph. This cycle must visit each clause component exactly once. We show that the possible visits to clause components are restricted to the "excursions" studied above. Fix any i and study the corresponding variable component. From $v_{i,1}$, there are only two edges to follow. Let us study the case that the DHC takes the edge $(v_{i,1}, v_{i,2})$; the other case is analogous. From $v_{i,2}$, the cycle must continue to the leftmost helper node on the doubly-linked list of the variable component and from there to the left node of the leftmost group. Otherwise, since $v_{i,2}$ has already been visited, the helper node could never be reached again on the cycle. Hence, the leftmost group node on the list is entered from the left. The next helper node on the list ensures that the cycle does not use a potential edge "$\rightarrow k$" outgoing of the right node of the leftmost group. If it used such an edge then, by similar arguments as before, the next helper node could not be on the cycle.

Now, if there is an edge "$\rightarrow k$" from the left node of the leftmost group then there is only a single outgoing edge "$k \rightarrow$" from c_k leading back into the same doubly-linked list (since a variable appears only once in a clause). By construction, this edge leads to the right node of the group we studied. If the cycle did not reach this node from c_k but from the right, coming from a different clause component, the cycle could not be continued from the node since there is only the ingoing edge "$k \rightarrow$". Altogether, a possible excursion from the leftmost group via a clause component leads immediately back into the group and the cycle continues to the right. Inductively, this holds also for all groups. Hence, the whole Hamilton cycle specifies a direction through each variable component that is in accordance with a satisfying assignment. If c_j was entered from the left while traversing the list belonging to x_i, setting $x_i = 1$ satisfies c_j and, analogously,

$x_i = 0$ is satisfying for the clause in the case the component was traversed from the right. □

Theorem 4.30 *The problem* HamiltonCycle *is $\mathcal{NP}$-complete.*

Proof. The proof is by reduction from DirectedHamiltonCycle. It is an exercise. □

An undirected graph $G = (V, E)$, $|V| = n$ has a *Hamilton Path* if there is a permutation $v_{i_1}, v_{i_2} \dots v_{i_n}$ of the nodes such that $\{v_{i_j}, v_{i_{j+1}}\} \in E$ for $j = 1, \dots, n-1$. A directed graph $G = (V, E)$, $|V| = n$, has a *Hamilton Path* if there is a Permutation $v_{i_1}, v_{i_2} \dots v_{i_n}$ of the nodes such that $(v_{i_j}, v_{i_{j+1}}) \in E$ for $j = 1, \dots, n-1$.

Problem 4.31 [HamiltonPath]
Input: An undirected graph $G = (V, E)$
Output: YES if G has a Hamilton Path and no otherwise.

Theorem 4.32 *The problem* HamiltonPath *is $\mathcal{NP}$-complete.*

Proof. The proof is by reduction from HamiltonCycle. It is an exercise. □

Problem 4.33 [DirectedHamiltonPath]
Input: A directed graph $G = (V, E)$.
Output: YES if G has a directed Hamilton Path P and no otherwise.

Theorem 4.34 *The problem* DirectedHamiltonPath *is $\mathcal{NP}$-complete.*

Proof. The proof is by reduction from DirectedHamiltonCycle. It is an exercise. □

Let $G = (V, E)$ be a directed weighted graph. The weight (or cost or length) of edge (v, w) is denoted by $c(v, w) \in \mathbb{N}$. Note that the weights are natural numbers and one can have $c(v, w) \neq c(w, v)$. The weight $c(P)$ of a path P is the sum of the weights of the edges on the path.

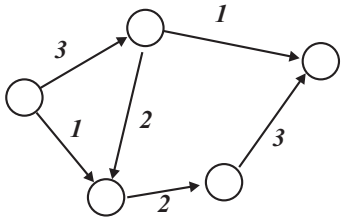


Figure 4.5: A directed graph with with a directed Hamilton path of weight 10.

Problem 4.35 [WeightedDirectedHamiltonPath]
Input: A directed graph $G = (V, E)$ and a natural number B.
Output: YES if G has a directed Hamilton path such that $c(P) \leq B$ and no otherwise.

We write WDHP for short. Figure 4.5 shows an example of a graph with a DHP with weight 10.

Theorem 4.36 *The problem* WeightedDirectedHamiltonPath *is $\mathcal{NP}$-complete.*

Proof. The proof is by restriction. The reference problem is DirectedHamilton-Path (HamiltonPath could be used just as well). Details will be given as an exercise or in the course. □

The directed version DirectedHamiltonCycle and the weighted, directed version WeightedDirectedHamiltonCycle of HamiltonCycle are also $\mathcal{NP}$-complete.

The next problem was already defined on page 14.

Problem 4.37 [Clique]
Input: An undirected graph $G = (V, E)$ and a natural number k.
Output: YES if the graph G has a k-clique, and NO otherwise.

Theorem 4.38 *The problem* Clique *is $\mathcal{NP}$-complete.*

Proof. The proof is by reduction from VertexCover. To see that Clique is in $\mathcal{NP}$, we interpret the random string as coding the nodes of a clique. We then check whether these nodes form a clique. To prove $\mathcal{NP}$-completeness, we show

$$\text{VertexCover} \leq_p \text{Clique} .$$

Let $G = (V, E)$ be the input graph and k the size constant for VertexCover. The input transformation T works as follows: Let $\bar{E}$ be the set of edges which are not in G and let $\bar{G} = (V, \bar{E})$ be the complement graph of G. Let $k' = |V| - k$. Then $(\bar{G}, k')$ is the transformed input for Clique:

$$T((G, k)) = (\bar{G}, k') .$$

The time for the transformation depends on how the graph is represented. For any reasonable representation it is polynomial in $|V| + |E|$. If, for example, the graph is given by its adjacency matrix, then the entries (0 – no edge, 1 – edge) have to be flipped. This takes time $|V|^2$. The computation of k' takes constant time.

Now assume that G has a vertex cover V' of size at most k. Then for all edges $\{a, b\} \in E$ we have $a \in V'$ or $b \in V'$. Consider the set $V \setminus V'$ in the complement graph $\bar{G}$. We have $|V \setminus V'| \geq |V| - k = k'$ Let $c, d \in V \setminus V'$. Then the edge $\{c, d\}$ is in $\bar{E}$, because $\{c, d\}$ is not an edge in G. If $\{c, d\}$ was an edge in G, it would not be covered by V'. Hence, all possible edges between the vertices of $V \setminus V'$ exist. Hence, $V \setminus V'$ is a k'-clique.

For the converse, assume that $\bar{G}$ has a k'-clique. Let $V' \subseteq V$ be the set of vertices of the clique. Let $c, d \in V'$ then $\{c, d\}$ is an edge in $\bar{E}$ and, by construction, $\{c, d\}$ is not an edge in E. Now consider an edge $\{a, b\} \in E$. By construction, $\{a, b\} \notin \bar{E}$. Then $a \notin V'$ or $b \notin V'$ (or both). If a and b were in V', then $\{a, b\}$ would be an edge in $\bar{E}$, as explained above. Hence $a \in V \setminus V'$ or $b \in V \setminus V'$. Then the edge $\{a, b\}$ is covered by $V \setminus V'$, whence $V \setminus V'$ is a vertex cover for G of size $k = |V| - |V'|$. This concludes the proof. □

For an undirected graph $G = (V, E)$, an *independent set* is a subset $V' \subseteq V$ of the vertices such that no edge is present, i. e., for all $v, w \in V'$, $v \neq w$: $(v, w) \notin E$. We say that G has a an *independent set* of size k if $|V'| = k$.

Problem 4.39 [IndependentSet]
Input: An undirected graph $G = (V, E)$ and a natural number k.
Output: YES if the graph G has an independent set of size k, and NO otherwise.

Theorem 4.40 *The problem* Independent Set *is $\mathcal{NP}$-complete.*

Proof. The proof is just a re-formulation: INDEPENDENT SET is CLIQUE for the complement graph. □

Theorem 4.41 *The problem* GLASSES IN A CUPBOARD *is $\mathcal{NP}$-complete.*

Proof. We know that the problem is in $\mathcal{NP}$.

We use DHP as a reference problem and show

$$\text{DHP} \leq_p \text{GLASSES IN A CUPBOARD.}$$

Let $G = (V, E)$ (a directed graph).

We let a glassblower make n glasses, all of the same height 10 cm. The glasses are made in such a way that if $(i, j) \in E$ then, when glass j is put into glass i, it exceeds glass i by 1 cm, i. e., $d_{ij} = 1$. If $(i, j) \notin E$ then, when glass j is put into glass i, it exceeds glass i by 2 cm, i. e., $d_{ij} = 2$. Now we construct a cupboard of height $(10+n-1)$ cm. Assume the glasses fit into it, and let $i_1, i_2, \ldots, i_n$ be the order in which they are stacked. Then for $k = 1, \ldots, n-1$, it must be that $d_{i_k, i_{k+1}} = 1$. This means that (i_k, i_{k+1}) is an edge in G. Hence, the edges $(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$ form a Hamilton path in G. On the other hand, if $(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$ is a Hamilton path in G, then $d_{i_k, i_{k+1}} = 1$, for $k = 1, \ldots, n-1$. If we stack the glasses in that order (glass i_{k+1} is put into glass i_k) then the stack will have height $10 + (n-1)1$ cm, and thus fit into the cupboard. □

Theorem 4.42 *The problem* POTATO SOUP *is $\mathcal{NP}$-complete.*

Proof. The proof is by re-interpretation. This is just a reformulation of SUBSET SUM. □

Theorem 4.43 *The problem* TRAVELING SALESPERSON *is $\mathcal{NP}$-complete.*

Proof. It is an exercise to show that TRAVELING SALESPERSON is in $\mathcal{NP}$. Now,

$$\text{HAMILTONCYCLE} \leq_p \text{TRAVELING SALESPERSON}$$

via the following transformation, which can be considered a restriction: Let $G = (V, E)$ be the input graph for HAMILTONCYCLE. The transformation constructs the cost matrix C for TRAVELING SALESPERSON as follows:

$$C(i, j) = \begin{cases} 1 & \text{if } \{i, j\} \in E \ , \\ 2 & \text{otherwise} \ . \end{cases}$$

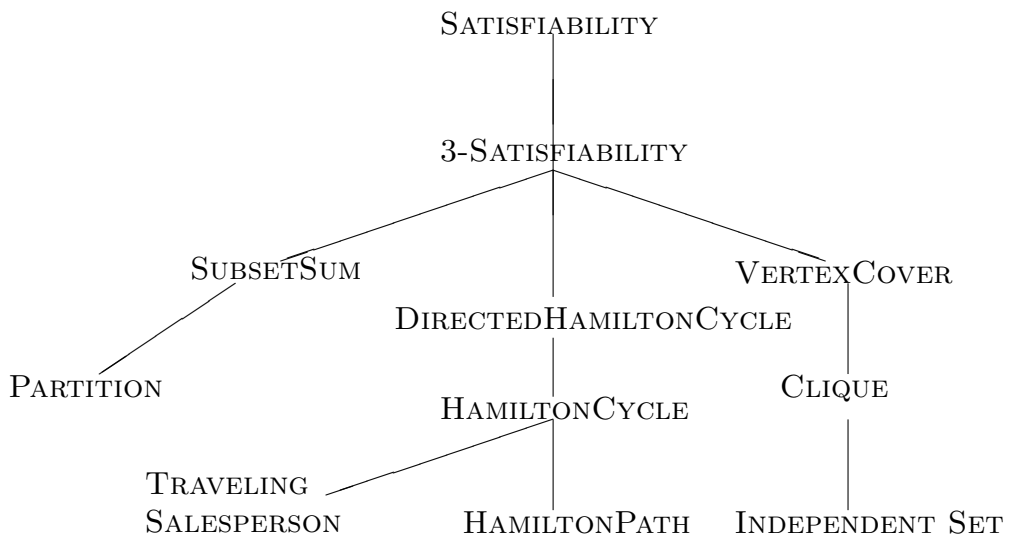


Figure 4.6: Overview of the reductions in this chapter.

That is, the edges of G have cost 1 the edges not in G correspond to edges in the graph for Traveling Salesperson which have cost 2. The cost bound for Traveling Salesperson is set to the number of vertices: $B = |V|$.

Now, if G has a Hamilton cycle then the edges of the cycle have cost 1 in the transformed input for Traveling Salesperson. Thus there is tour with costs B. If, on the other hand, the transformed input for Traveling Salesperson has a solution with cost at most B then all the edges on the tour (of length B) must have cost 1, i. e., they correspond to edges which are present in G, and these form a Hamiltonian cycle in G. □

The polynomial reductions which were proved in this chapter are visualized in Figure 4.6. Note that all reductions go back to the $\mathcal{NP}$-completeness of Satisfiability, which was the first problem known to be $\mathcal{NP}$-complete. Examples for the different reduction techniques (component design, local replacement, restriction and modelling) are listed in Table 4.1.

Component Design	3-SATISFIABILITY $\leq_p$ SUBSETSUM 3-SATISFIABILITY $\leq_p$ DIRECTEDHAMILTONCYCLE 3-SATISFIABILITY $\leq_p$ VERTEXCOVER SUBSETSUM$\leq_p$ PARTITION
Local Replacement	SATISFIABILITY $\leq_p$ 3-SATISFIABILITY VERTEXCOVER $\leq_p$ CLIQUE CLIQUE $\leq_p$ INDEPENDENT SET
Restriction	PARTITION $\leq_p$ SUBSETSUM DIRECTEDHAMILTONCYCLE $\leq_p$ TRAVELING SALESPERSON
Modelling	POTATOSOUP $\leq_p$ PARTITION

Table 4.1: Examples for the different reduction techniques used in this chapter.

4.4 List of Further $\mathcal{NP}$-complete Problems

We continue by giving further $\mathcal{NP}$-complete problems that can be good reference problems in $\mathcal{NP}$-completeness proofs.

Problem 4.44 [BINPACKING]
Input: A sequence $s_1, s_2, , \ldots, , s_n$ of positive rational numbers and a natural number B.
Interpretation: We want to pack n objects with sizes $s_1, s_2, , \ldots, , s_n$ into as few bins as possible. Every bin has a capacity of 1. The objects can be be packed into the bins such that there is no space between them. The objects cannot be divided. The sum of the sizes of the objects in a certain bin cannot exceed 1.
Output: YES if the objects can be packed into at most B bins and NO otherwise.

Problem 4.45 [KNAPSACK]
Input: Sequences $w_1, w_2, \ldots, w_n$ and $s_1, s_2, \ldots, s_n$ of natural numbers and natural numbers B, K.
Interpretation: We have n objects $a_1, \ldots, a_n$. The i-th object has weight $w_i \leq B$ and value s_i. We want to pack objects into a knapsack which has weight limit B such that the value of the objects in the knapsack is maximum.
Output: YES if there is a set $A \subseteq \{1, 2, , \ldots, , n\}$ such that:

$$\sum_{i \in A} w_i \leq B \qquad \text{and} \qquad \sum_{i \in A} s_i \geq K \ .$$

Output for the optimizing version: A subset $A \subseteq \{a_1, \ldots, a_n\}$ of the objects such that the value $V = \sum_{a_i \in A} s_i$ is maximum and the capacity is not exceeded, that is, $\sum_{a_i \in A} w_i \leq B$.

Problem 4.46 [MaximumSatisfiability]
Input: A set of clauses over n boolean variables and a natural number k.
Output: YES if there is an assignment of truth values to the variables which satisfies at least k clauses and NO otherwise.

Problem 4.47 [IntegerProgramming]
Input: Parameters $c_1, \ldots, c_m \in \mathbb{Z}$, $a_{j1}, \ldots, a_{jm}, b_j \in \mathbb{Z}$, $j = 1, \ldots, k$ and $B \in \mathbb{Z}$.
Output: YES it there are $x_1, \ldots, x_m \in \mathbb{Z}$ such that the following holds:

$$\begin{aligned} c_1 x_1 + \cdots + c_m x_m &\geq B \\ a_{j1} x_1 + \cdots + a_{jm} x_m &\leq b_j, \quad j = 1, \ldots, k \end{aligned}$$

and NO otherwise.

If the numbers x_i are allowed to be reals then the problem is efficiently solvable.

Problem 4.48 [MaximumCut]
Input: An undirected graph $G = (V, E)$ and a constant $k \in \mathbb{N}$.
Output: YES if there is a partition of V into two sets V_1, V_2 such that there are at least k edges between V_1 and V_2, and NO otherwise.

Problem 4.49 [GraphColoring]
Input: An undirected graph $G = (V, E)$ and a constant $k \in \mathbb{N}$.
Output: YES if there is a k-coloring of G and NO otherwise. A *k-coloring* assigns every vertex one of k colors such that adjacent vertices have different colors. The problem can be solved in polynomial time for $k = 2$ and is $\mathcal{NP}$-complete for $k \geq 3$.

Problem 4.50 [OnеProcessorScheduling]
Input: A set T of tasks. Every task $t \in T$ has a length $l(t) \in \mathbb{Z}^+$, an earliest release time $r(t) \in \mathbb{Z}_0^+$, and a deadline $d(t) \in \mathbb{Z}^+$.
Output: YES if there is a schedule σ for a single processor. That is σ assigns a start time $\sigma(t) \in \mathbb{Z}_0^+$ to each job such that for all $t, t' \in T$

$$\begin{aligned} \sigma(t) &\geq r(t) \ , \\ \sigma(t) + l(t) &\leq d(t) \ , \\ \sigma(t) > \sigma(t') &\implies \sigma(t) \geq \sigma(t') + l(t') \ . \end{aligned}$$

Problem 4.51 [MultiProcessorScheduling]
Input: A set T of tasks, m processors and a deadline $D \in \mathbb{Z}^+$. Every task $t \in T$ has a length $l(t) \in \mathbb{Z}^+$.
Output: YES if there is a schedule σ which meets the deadline D. That is σ assigns a start time $\sigma(t) \in \mathbb{Z}_0^+$ to each job such that

- for all times $u \in \{0, 1, \ldots, D\}$ the number of active tasks t ($\sigma(t) \leq u < \sigma(t) + l(t)$) is at most m,
- for all tasks $t \in T$ it holds that $\sigma(t) + l(t) \leq D$.

4.5 Pseudo-polynomial Algorithms and Approximation Schemes

For some $\mathcal{NP}$-complete problems, more precisely, their optimization versions, one can find algorithms that seem to solve the problem in polynomial time. Often, these algorithms are based on a dynamic programming approach. In fact, these algorithms are not polynomial in the input size but only polynomial "in some parameter of the problem" and are therefore called *pseudo-polynomial*. When the value of this parameter is proportional to the input size then the algorithms run in polynomial time on this

input. However, the value of the parameter can be exponential in the input size, and the algorithm then runs in exponential time. We demonstrate this using the KNAPSACK problem, see Page 60. In fact, we will present two similar algorithms that are pseudo-polynomial.

Let us recall the KNAPSACK problem. The problem components are objects $a_1, a_2, \ldots, a_n$ with weights $w_1, w_2, \ldots, w_n$ and values $s_1, s_2, \ldots, s_n$ and B, the capacity of the knapsack. The aim is to select objects to pack into the knapsack such that it is not overloaded and the value is maximal.

4.5.1 Algorithm for Small Object Weights

We present the first algorithm. Here the capacity B is the parameter of the problem which determines the running time.

As most Dynamic Programming algorithms, our algorithm can be visualized as a table-filling method. For details on Dynamic Programming, the reader is referred to the literature on Algorithms and Data Structures.

We will use the following problem instance as an example.

$$\begin{aligned} n &= 6 \\ W &= \{4, 1, 7, 5, 2, 2\} \\ S &= \{6, 8, 4, 9, 3, 5\} \\ B &= 11 \end{aligned}$$

We initialize a 2-dimensional $(n+1) \times (B+1)$ table V, for the example this is a 7×12 table.

–	0	1	2	3	4	5	6	7	8	9	10	11
0	–	–	–	–	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–	–	–	–

Let $A(i, w) \subseteq \{a_1, a_2, \ldots, a_i\}$ be a subset of the first i-objects with maximal value and weight at most w. For the example $A(3, 4) = \{2\}$ because of the first three objects

a_1, a_2, a_3 only a_1, a_2 meet the weight limit of 4. Not both can be chosen as their accumulated weight is larger than 4. As a_2 has the larger value, the solution is a_2, i. e., $A(3,4) = \{2\}$.

Let $V(i,w) = \sum_{a \in A(i,w)} s_i$ be the value of $A(i,w)$. In other words, $V(i,w)$ is the maximum value that can be composed with weight at most w using only objects up to i. The final result is in cell $V(n,B)$, the maximum value achievable with weight limit B when all n objects are allowed. For the example $V(3,4) = 8$. We only compute the optimal value here. In order to select a set of objects which attain this value, some more bookkeeping is needed.

We now initialize the table. One has $V(0,j) = 0$ for $j = 0, \ldots, B$ because with no objects one can only get a value of 0. One also has $V(i,0) = 0$ for $i = 0, \ldots, n$ because one cannot pack anything into a knapsack with capacity 0.

–	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	–	–	–	–	–	–	–	–	–	–	–
2	0	–	–	–	–	–	–	–	–	–	–	–
3	0	–	–	–	–	–	–	–	–	–	–	–
4	0	–	–	–	–	–	–	–	–	–	–	–
5	0	–	–	–	–	–	–	–	–	–	–	–
6	0	–	–	–	–	–	–	–	–	–	–	–

In order to fill the rest of the table, we use the following recurrence, which we only briefly explain.

The value $V(i+1,w)$ is computed by distinguishing

- If $w_{i+1} \leq w$ then object a_{i+1} might be used and therefore

$$V(i+1,w) = \max\{V(i,w)\,;\, s_{i+1} + V(i, w - w_{i+1})\}\ .$$

- If $w_{i+1} > w$ then object a_{i+1} cannot be used and

$$V(i+1,w) = V(i,w)\ .$$

Using the above recurrences, one fills the table row-wise from left to right. Note that the indices on the right-hand sides of the recurrences refer to table cells where the values are already computed. The time to fill the table is $\Theta(nB)$, because we **have to** compute nB entries and every entry can be computed in constant time. The resulting

table looks like this:

–	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	6	6	6	6	6	6	6	6
2	0	8	8	8	8	14	14	14	14	14	14	14
3	0	8	8	8	8	14	14	14	14	14	14	14
4	0	8	8	8	8	14	17	17	17	17	23	23
5	0	8	8	11	11	14	17	17	20	20	23	23
6	0	8	8	13	13	16	17	19	22	22	25	25

The entry $V(n, B)$ ($V(6, 11)$ in the example) is 25, hence this is the optimal value we can pack into the knapsack.

In addition to the value of an optimal solution, an optimal solution itself can be recovered if we additionally store whether the expression $\max\{V(i, w)\,;\, s_{i+1}+V(i, w-w_{i+1})\}$ explained above takes its maximum on the first term (which means that object a_{i+1} should not be packed in) or the second term (which means it should be packed in). Doing this for all cells, we store which packing leads to the value computed in the cell. The optimal solution in our example is $A = \{a_2, a_4, a_5, a_6\}$ with weight 10.

This looks like nice and fast solution method but let us look at the following problem instance.

$$\begin{aligned} n &= 6 \\ W &= \{402341, 128834, 712323, 5232838, 228292, 211827\} \\ S &= \{6, 8, 4, 9, 3, 5\} \\ B &= 1173917 \end{aligned}$$

The input size (that is the number of symbols) has not increased by much. In fact, we still have 6 objects. To use the dynamic programming approach, we now need a 7×1173918 table. Although the problem size has only increased by a factor of 4, the running time has increased by more than 100000. As the running time depends on B, increasing this parameter will increase the running time even further. Note that it makes no sense to consider values of B which are larger than the sum of the weights of all objects; this is why we also increased the weights in the last example.

Altogether, we can only guarantee this algorithm to be efficient if the sum of the weights is small.

4.5.2 Algorithm for Small Object Values

In the second algorithm, the sum $S := s_1 + \cdots + s_n$ of the object values is the parameter that determines the running time. We use a 2-dimensional $(n + 1) \times (S + 1)$ table V,

where $V(i,s)$ contains the *minimum* weight that allows a total value of *exactly* s if only the first i objects are allowed to be selected. Formally,

$$V(i,s) \;=\; \min\left\{ \sum_{a_j \in A} w_j \mid A \subseteq \{a_1,\ldots,a_i\}, \sum_{a_j \in A} s_j = s \right\}$$

if there is such a selection A; otherwise $V(i,s) = \infty$.

We initialize the table as follows. We have $V_{0,0} = 0$ since value 0 can be achieved without selecting objects. Moreover, $V_{0,s} = \infty$ for $s > 0$ since we cannot achieve positive value without selecting objects.

The value $V(i+1,s)$ is computed by distinguishing

- If $s_{i+1} \leq s$ then object a_{i+1} might be used and
$$V(i+1,s) = \min\{V(i,s)\,;\, w_{i+1} + V(i, s - s_{i+1})\}\ .$$
- If $s_{i+1} > s$ then object a_{i+1} cannot be used to achieve a total value of exactly s and therefore
$$V(i+1,s) = V(i,s)\ .$$

Using the above recurrences, one fills the table row-wise from left to right as in the first algorithm. Note again that the indices on the right-hand sides of the recurrences refer to table cells where the values are already computed. We finally output the value in the rightmost column where there is still an entry of at most B.

As an example, consider the following problem instance.

$$\begin{aligned} n &= 3 \\ W &= \{6,2,2\} \\ S &= \{4,4,2\} \\ B &= 6 \end{aligned}$$

Then the final table would look like this and the output would be 6 since column 6 contains the entry 4 but not column to its right has an entry of size at most B. Analogously to the first algorithm, an optimal solution itself can be recovered by storing for each cell whether using object a_{i+1} minimizes the expression for $V(i+1,s)$ or not.

–	0	1	2	3	4	5	6	7	8	9	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	∞	∞	6	∞	∞	∞	∞	∞	∞
2	0	∞	∞	∞	2	∞	∞	∞	8	∞	∞
3	0	∞	2	∞	2	∞	4	∞	8	∞	10

The time to fill the table is $\Theta(nS)$, because we **have to** compute nS entries and every entry can be computed in constant time. Altogether, this algorithm can only be efficient if the sum of the object values is small.

4.5.3 An Approximation Scheme

In this section, we show that the previous dynamic programming algorithm can be used as a subroutine in order to compute solutions of good quality in polynomial time even if both weights and values of the objects are very large. The idea is to use divide all object values by some large enough number k and round them afterwards down to the nearest integer (note that integrality is required). Then the algorithm from the previous subsection is run. If k is chosen appropriately, the sum of the values is small enough to allow for a polynomial-time algorithm. At the same time, if k is not chosen too large, the errors introduced by rounding are small enough and the quality of the solution is "good".

More precisely, let an instance to the knapsack problem consisting of n objects $a_1, \ldots, a_n$ with weights $w_1, \ldots, w_n$ and values $s_1, \ldots, s_n$ and knapsack capacity B be given. The aim is to compute a solution of a value s that is only by a factor of $1+\epsilon$ worse than the optimum value OPT, i. e., $\mathrm{OPT}/s \leq 1+\epsilon$. Here $\epsilon > 0$ is chosen sufficiently small and denotes the error threshold. For example, if we choose $\epsilon = 0.01$, then the solution will be only 1% off from the optimum, so if the best solution value is 100, our algorithm is guaranteed to compute a solution of value at least $100/1.01 > 99$. Choosing a very small ϵ, however, will be at the expense of increasing the runtime. In the theory of algorithms, an algorithm like ours with adjustable error threshold ϵ is often called an *approximation scheme.*

Our approximation scheme works as follows. We define

$$k := \frac{\epsilon \cdot \max\{s_1, \ldots, s_n\}}{(1+\epsilon)n}$$

and create a modified instance to the knapsack problem with the original weights $w_1, \ldots, w_n$ but modified values $\tilde{s}_1, \ldots, \tilde{s}_n$, where

$$\tilde{s}_i := \left\lfloor \frac{s_i}{k} \right\rfloor = \left\lfloor \frac{(1+\epsilon) \cdot n \cdot s_i}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor$$

for $1 \leq i \leq n$. Moreover, the capacity B is the same as in the original instance.

Clearly, since the weights and capacity of original and modified instance are the same, every feasible solution (i. e., a solution that does not exceed the capacity bound) to the modified instance is also a feasible solution to the original instance. We therefore run the algorithm from Section 4.5.2 on the modified instance and use the solution it outputs as a solution to the original problem. That completes the description of the

approximation scheme. We now have to analyze the running time of our algorithm and the quality of the solution obtained with respect to the original instance.

Creating the modified instance can be done in linear time. The running time of the algorithm is therefore dominated by the running time of the algorithm from Section 4.5.2. This is bounded from above by

$$O(n \cdot (\tilde{s}_1 + \cdots + \tilde{s}_n)) = O(n \cdot n \cdot \max\{\tilde{s}_1, \ldots, \tilde{s}_n\}),$$

which, by definition of the $\tilde{s}_i$, is

$$O\left(n^2 \cdot \max\left\{\left\lfloor \frac{(1+\epsilon)\cdot n \cdot s_1}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor, \ldots \left\lfloor \frac{(1+\epsilon)\cdot n \cdot s_n}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor\right\}\right),$$

which, as the arguments to the maximum are linear in the s_i, is the same as

$$O\left(n^2 \cdot \frac{(1+\epsilon)\cdot n}{\epsilon}\right) = O\left(\frac{n^3}{\epsilon} + n^3\right).$$

Hence, the running time is polynomial in the input length n and in $1/\epsilon$. If ϵ is a constant, then the running time is $O(n^3)$.

We denote the solution obtained for the modified instance by A. Note that the algorithm computes an optimal solution with respect to the modified instance. Moreover, we denote an optimal solution with respect to the original instance by A' (only for the sake of comparison; note that we did not compute this by our algorithm). Both A' and A do not exceed the capacity bound B. Since A is optimal for the modified instance, A' cannot be better than A here. Formally,

$$\sum_{a_i \in A'} \tilde{s}_i \leq \sum_{a_i \in A} \tilde{s}_i.$$

(Analogously, $\sum_{a_i \in A} s_i \leq \sum_{a_i \in A'} s_i$, which is not needed here, though).

Since $\tilde{s}_i = \lfloor s_i/k \rfloor$, we have

$$\frac{s_i}{k} - 1 \leq \tilde{s}_i \leq \frac{s_i}{k}$$

for $1 \leq i \leq n$. This bracketing of the $\tilde{s}_i$ is used to estimate the quality of the solution A in terms of the original instance. Let OPT denote the value of an optimal solution to the original instance, i. e., $\mathrm{OPT} = \sum_{a_i \in A'} s_i$. We have

$$\sum_{a_i \in A} s_i \;\geq\; k \cdot \sum_{a_i \in A} \tilde{s}_i \;\geq\; k \cdot \sum_{a_i \in A'} \tilde{s}_i \;\geq\; k \cdot \sum_{a_i \in A'} \left(\frac{s_i}{k} - 1\right) \;\geq\; \left(\sum_{a_i \in A'} s_i\right) - nk,$$

where the last inequality follows since there are at most n elements in A'. We recognize OPT and plug in the definition of k. Hence, the last expression equals

$$\mathrm{OPT} - n \cdot \frac{\epsilon \cdot \max\{s_1, \ldots, s_n\}}{(1+\epsilon)n} = \mathrm{OPT} - \frac{\epsilon}{1+\epsilon} \cdot \max\{s_1, \ldots, s_n\}.$$

Now, $\mathrm{OPT} \geq \max\{s_1, \ldots, s_n\}$ since it is always possible to put only the most valuable object into the knapsack (note that $w_i \leq B$ for all i). Hence, the last expression is at least

$$\mathrm{OPT} - \frac{\epsilon}{1+\epsilon} \cdot \mathrm{OPT} = \mathrm{OPT} \cdot \left(1 - \frac{\epsilon}{1+\epsilon}\right) = \frac{\mathrm{OPT}}{1+\epsilon}.$$

We have proved

$$\frac{\mathrm{OPT}}{\sum_{a_i \in A} s_i} \leq 1 + \epsilon,$$

which means that the solution computed for the modified instance is only by a factor of $1 + \epsilon$ worse than an optimal solution to the original instance.

5 Design and Analysis of Randomized Algorithms

5.1 Examples

In Chapter 4, we used random algorithms to classify (probably) hard problems by showing that they are $\mathcal{NP}$-complete. We now turn to a more constructive use of randomization. We know from the discussions in Chapter 3 that randomized algorithms can show a strange behaviour: Their running time can be random, their output can be random or even both. For some problems, there are algorithms for which the random behaviour is somewhat controlled. At the end of Chapter 3, we defined complexity classes for some possible combinations of probabilistic guarantees on running time or correctness of the result.

In this chapter, we will also look at the optimization versions of the problems. That is, we are interested in finding an optimal solution or at least one that is close to the optimum.

5.2 A Randomized Algorithm for Independent Set

The problem of finding a maximum independent set in a graph is $\mathcal{NP}$-complete. Hence, we cannot hope to find an efficient algorithm for solving it. We now look at the problem of finding a *large* independent set instead and present a randomized algorithm for it. The analysis is a simple example for the use of expectations. Let $G = (V, E)$, $|V| = n$, $|E| = m$, be an undirected graph. Let d_i be the (number of adjacent edges) of vertex i, and let

$$d_{avg} = \frac{1}{n} \sum_{i=1}^{n} d_i = \frac{2m}{n}$$

be the average degree of G.

Theorem 5.1 *Let $G = (V, E)$, $|V| = n$, $|E| = m$, be an undirected graph with $d_{avg} > 1$. There is a randomized algorithm A_{ip} which computes an independent set of* ***expected*** *size at least $n/(2\,d_{avg})$*

Proof. Let $p_1, p_2, \ldots, p_n \in [0, 1]$. The p_i do not have to be a probability distribution, i. e., they do not have to sum to 1! We shall later see how to define the p_i. Now algorithm A_{ip} constructs a set $I \subseteq V$ as follows: Initially, $I = \emptyset$. Then for every node i, $i = 1, \ldots, n$, it adds i to I with probability p_i (and does not add it with probability $1 - p_i$). Then we check for every pair $i, j \in I$, whether $\{i, j\} \in E$ and, if so, remove one of the nodes. The decision whether to remove i or j is made at random. Let I^* be the resulting set of nodes. Clearly I^* is an independent set, but might be empty.

We want to compute the expected size $\boldsymbol{E}[\,|I^*|\,]$ of I^*. To this end, let us first compute the expected size $\boldsymbol{E}[\,|I|\,]$ of I. As node i is a member of I with probability p_i we are in situation of Equation A.19 and get

$$\boldsymbol{E}[\,|I|\,] = \sum_{i=1}^{n} p_i \ .$$

The next step is to determine the expected number of nodes which are removed from I when constructing I^*. Let F be the set of edges between nodes of I. We only remove node i if there is an edge $\{i, j\} \in F$ (for some $j \in I$). Hence, removing i also removes at least this edge. Note that even more edges could be removed from F. Hence, the number of edges removed is no less than the number of nodes removed. In the worst case, all edges in F are removed individually. Let us compute the expected number of edges in F. An edge $\{i, j\} \in E$ is in F if both endpoints i and j are in I. Because the nodes of I are selected independently, this happens with probability $p_i p_j$. As above, an edge contributes 1 to $|F|$ if both endpoints are in I and 0 otherwise. Again using Equation A.19, we get

$$\boldsymbol{E}[\,|F|\,] = \sum_{\{i,j\} \in E} p_i p_j \ .$$

Now we are putting things together to determine $\boldsymbol{E}[\,|I^*|\,]$:

$$\begin{aligned}
\boldsymbol{E}[\,|I^*|\,] &= \boldsymbol{E}[\,|I| - \text{number nodes removed from } I\,] \\
&\geq \boldsymbol{E}[\,|I| - \text{number edges removed from } F\,] \\
&= \boldsymbol{E}[\,|I| - |F|\,] \\
&= \boldsymbol{E}[\,|I|\,] - \boldsymbol{E}[\,|F|\,] \\
&= \sum_{i=1}^{n} p_i - \sum_{\{i,j\} \in E} p_i p_j \ . \qquad (5.1)
\end{aligned}$$

We now have to find a choice for the probabilities p_i which makes $\sum_{i=1}^{n} p_i - \sum_{\{i,j\} \in E} p_i p_j$ – and hence $\boldsymbol{E}[\,|I^*|\,]$ – large. Let us try to set all equal: $\forall i \colon p_i = p$. Then (5.1) becomes

$$\boldsymbol{E}[\,|I^*|\,] \geq np - mp^2 \ . \qquad (5.2)$$

We want to choose p in order to maximize the expression. Differentiating $np - mp^2$ for p gives $n - 2pm$. Setting this equal to zero and solving for p gives $p = n/(2m) = 1/d_{\text{avg}}$. As the second derivative $-2m$ is negative, this is a maximum. With $p = n/(2m) = 1/d_{\text{avg}}$, Equation (5.2) becomes (using $n/(2m) = 1/d_{\text{avg}}$):

$$\boldsymbol{E}[\,|I^*|\,] \geq n\frac{1}{d_{\text{avg}}} - m\left(\frac{1}{d_{\text{avg}}}\right)^2 = \frac{n}{d_{\text{avg}}} - m\frac{n^2}{4m^2} = \frac{n}{d_{\text{avg}}} - \frac{n^2}{4m} = \frac{n}{d_{\text{avg}}} - \frac{n}{2d_{\text{avg}}} = \frac{n}{2d_{\text{avg}}}\ ,$$

which finishes the proof. □

As in Section 4.5.3, we want to relate the value of the solution given by the algorithm to the value of an optimal solution, noting that the aim is maximization. However, unlike Section 4.5.3, we are dealing with randomized algorithms here. Let $\text{OPT}(\boldsymbol{X})$ be the value of an optimal solution to the given problem (i. e., the size of a maximal independent set) and let $A(\boldsymbol{X})$ be the value of the solution output by the algorithm.

If A is a deterministic algorithm for a maximization problem, the *approximation ratio* $R_A(\boldsymbol{X})$ of A on $\boldsymbol{X}$ is defined by

$$R_A(\boldsymbol{X}) = \frac{\text{OPT}(\boldsymbol{X})}{A(\boldsymbol{X})}\ .$$

For minimization problems, the approximation ratio is defined by

$$R_A(\boldsymbol{X}) = \frac{A(\boldsymbol{X})}{\text{OPT}(\boldsymbol{X})}\ ,$$

which is the reciprocal of the definition for maximization problems. In this way, always $R_A(\boldsymbol{X}) \geq 1$. The closer the value is to 1, the better the approximation is. If A is randomized then $R_A(\boldsymbol{X})$ for maximization problems is defined to be the **expected** approximation ratio:

$$R_A(\boldsymbol{X}) = \frac{\text{OPT}(\boldsymbol{X})}{\boldsymbol{E}[\,A(\boldsymbol{X})\,]}\ .$$

and for minimization problems, it is defined as the reciprocal of this value.

Since $\text{OPT}(\boldsymbol{X}) \leq n$, the expected approximation ratio of our algorithm for INDEPENDENT SET is no worse than $n/\frac{n}{2d_{\text{avg}}} = 2d_{\text{avg}}$.

Of course, other choices for p are possible. It might be a good idea to prefer nodes with low degree because we are looking for an independent set. A way of achieving this is to set $p_i = 1/(d_i + 1)$.

5.3 Randomized Algorithms for MaximumSatisfiability

Let us now look at the problem MaximumSatisfiability. As this is $\mathcal{NP}$-complete, we can only hope for efficient approximation algorithms. We start by giving a probabilistic proof for a lower bound on the number of clauses that can be satisfied.

Theorem 5.2 *Let $C = \{c_1, c_2, \ldots, c_m\}$ be a set of clauses over the Boolean variables $x_1, x_2, \ldots, x_n$. Then there is an assignment of truth values to the x_i which satisfies at least $m/2$ clauses.*

Proof. For all i, independently set x_i to **true** or **false** with probability 1/2. For $j = 1, 2, \ldots, m$, let $s_j = 1$ if c_j is satisfied and $s_j = 0$ otherwise. The s_j are random variables. If c_j has ℓ literals then the probability that c_j is not satisfied is $2^{-\ell}$ (by independence, and the fact that c_j is not satisfied iff all literals are **false**). Hence, the probability that c_j is satisfied is $1 - 2^{-\ell} \geq 1/2$ and it follows that $\boldsymbol{E}[\, s_j \,] \geq 1/2$ for all j. Let N denote the number of satisfied clauses, N is a random variable. We have $\boldsymbol{E}[\, N \,] = \sum_{j=1}^{m} \boldsymbol{E}[\, s_j \,]$ by (A.19). Thus $\boldsymbol{E}[\, N \,] \geq m \cdot (1/2) = m/2$. By Fact A.1, we have that there is an assignment for which $\sum_{j=1}^{m} s_j \geq m/2$, finishing the proof. □

The last proof is *non-constructive* as it only establishes the **existence** of a truth assignment with certain properties but does not produce such an assignment. It is an exercise to come up with an algorithm that constructs such an assignment. In any case, these simple algorithms have an approximation ratio that is only bounded by 2.

We shall proceed by developing a randomized algorithm with expected performance ratio no worse than 4/3. In fact, the approximation ratio for the algorithm implicit in Theorem 5.2 is already bounded by 4/3 if the clauses have all length 2 or more; the details are left as an exercise. Hence, a problem in achieving a performance ratio of 4/3 in general is the existence of clauses of a single literal.

Let $C = \{c_1, c_2, \ldots, c_m\}$ still be a set of clauses over the Boolean variables $x_1, x_2, \ldots, x_n$. We formulate the problem MaximumSatisfiability as an integer linear optimization problem with *binary* variables y_i, $i = 1, 2, \ldots, n$, and z_j, $j = 1, 2, \ldots, m$. We write $x_i \in c_j$ if x_i is a literal in clause c_j and $\bar{x}_i \in c_j$ if $\bar{x}_i$ is a literal in clause c_j. The interpretation is that $z_j = 1$ iff c_j is satisfied and that y_i is the truth value assigned to

x_i. Then the linear optimization problem is:

$$\textbf{max!} \quad \sum_{j=1}^{m} z_j \qquad \text{subject to} \tag{5.3}$$

$$y_i, z_j \in \{0,1\} \qquad i = 1,2,\ldots,n, \qquad j = 1,2,\ldots,m\ , \tag{5.4}$$

$$\sum_{x_i \in c_j} y_i + \sum_{\bar{x}_i \in c_j} (1 - y_i) \geq z_j, \qquad j = 1,2,\ldots,m\ . \tag{5.5}$$

The expression in (5.3) is called the *target function*. Formula (5.5) counts how many literals in clause c_j are satisfied if x_i is set to y_i. As we are maximizing $\sum_{j=1}^{m} z_j$, clause c_j is satisfied if $z_j = 1$ and not satisfied if $z_j = 0$. As MAXIMUMSATISFIABILITY is $\mathcal{NP}$-complete, it is also hard to solve the above integer program. We *relax* the program by replacing Condition (5.4) by $y_i, z_j \in [0,1]$. The relaxation makes the value of the target function only greater. There is an exercise that this can happen. For the relaxed linear program, efficient solution algorithms are known. They will assign real numbers to the variables y_i and z_j. These have no meaning for the boolean case but can be interpreted like "the clause is 75% satisfied".

Now let $\hat{y}_i$, $i = 1,2,\ldots,n$, and $\hat{z}_j$, $j = 1,2,\ldots,m$, $\hat{y}_i, \hat{z}_j \in [0,1]$ be a solution of the relaxed linear program. We have to assign Boolean values to the x_i. To do so, we use a technique called *randomized rounding*:

With probability $\hat{y}_i$ set $x_i = 1$, and with probability $1 - \hat{y}_i$ set $x_i = 0$.

We now analyze how many clauses are satisfied by randomized rounding (of course, in expectation). We formulate the basic property in the following lemma:

Lemma 5.3 *Let c_j consist of k literals. Then the probability that c_j is satisfied by randomized rounding is at least*

$$\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \hat{z}_j\ .$$

Proof. Let us for simplicity assume that $c_j = x_1 \vee x_2 \vee \cdots \vee x_k$. Then c_j is not satisfied if all x_i are set to zero. This happens with probability $\prod_{i=1}^{k}(1 - \hat{y}_i)$, and c_j is satisfied with the complementary probability $1 - \prod_{i=1}^{k}(1 - \hat{y}_i)$. From $\hat{y}_1 + \hat{y}_2 + \cdots + \hat{y}_k \geq \hat{z}_j$, we have

$$(1 - \hat{y}_1) + (1 - \hat{y}_2) + \cdots + (1 - \hat{y}_k) \leq k - \hat{z}_j\ .$$

Using Lemma B.1 with $a_i = (1 - \hat{y}_i)$, we get

$$\prod_{i=1}^{k}(1 - \hat{y}_i) \leq \left(\frac{(1 - \hat{y}_1) + \cdots + (1 - \hat{y}_k)}{k}\right)^k \leq \left(\frac{k - \hat{z}_j}{k}\right)^k\ .$$

Thus

$$1 - \prod_{i=1}^{k}(1 - \hat{y}_i) \geq 1 - \left(\frac{k - \hat{z}_j}{k}\right)^k \ . \tag{5.6}$$

By Lemma B.2, the right-hand side of (5.6) is at least $\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \hat{z}_j$. □

Using Equation (B.2) of Lemma B.3, we have that $\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \geq 1 - \frac{1}{e} \sim 0.632$.

Theorem 5.4 *Let $c_1, c_2, \ldots, c_m$ be clauses over $x_1, x_2, \ldots, x_n$. Let k_j be the number of literals in c_j. Let $N_{\max}$ be the maximum number of clauses that can be satisfied. Let $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ be the solution of the relaxed linear program. Then randomized rounding will satisfy at least $\left(1 - \frac{1}{e}\right) \cdot N_{\max}$ clauses in expectation.*

Proof. In the relaxed linear program we have $N_{\max} \leq \sum_{j=1}^{m} \hat{z}_j$. The expected number of satisfied clauses is

$$\sum_{j=1}^{m}\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \hat{z}_j \geq \left(1 - \frac{1}{e}\right) \cdot \sum_{j=1}^{m} \hat{z}_j \geq \left(1 - \frac{1}{e}\right) \cdot N_{\max} \ .$$

□

We can think of the algorithm in the proof of Theorem 5.2 as using randomized rounding with $\hat{y}_i = 1/2$, for all i. Now our final algorithm MIX is as follows.

Algorithm 5.5 [MIX] Let $\hat{y}'_i = 1/2$, for all i. Compute an assignment α_1 of truth values to the variables $x_1, x_2, \ldots, x_n$ by randomized rounding using $\hat{y}'_i$.

Then let $\hat{y}_i$ be the solution of the relaxed linear program. Let α_2 of truth values to the variables $x_1, x_2, \ldots, x_n$ by randomized rounding using these $\hat{y}_i$.

Output that of the two solutions that satisfies more clauses.

Theorem 5.6 *Let $c_1, c_2, \ldots, c_m$ be clauses over $x_1, x_2, \ldots, x_n$. Let $N_{\max}$ be the maximum number of clauses that can be satisfied. Then the solution of algorithm MIX satisfies at least $3/4 \cdot N_{\max}$ clauses in expectation.*

Proof. We know from above that randomized rounding with $\hat{y}'_i = 1/2$ satisfies a clause with k literals with probability $1 - 2^{-k}$ and that randomized rounding with $\hat{y}_i$

from the relaxed linear program satisfies a clause with k literals with probability $\left(1-\left(1-\frac{1}{k}\right)^k\right)\cdot\hat{z}_j$. To gain some insight, let us look at Table 5.1 for these values: Let $A_k := 1-2^{-k}$ and $B_k := \left(1-\left(1-\frac{1}{k}\right)^k\right)$. Let $C_k := (A_k+B_k)/2$ be their average. The table supports our idea that the algorithm of Theorem 5.2 is superior for long

k	A_k	B_k	C_k
1	0.5	1.0	0.75
2	0.75	0.75	0.75
3	0.875	0.704	0.7895
4	0.938	0.684	0.811
5	0.969	0.672	0.8205

Table 5.1: The first values for $A_k = 1-2^{-k}$, $B_k = \left(1-\left(1-\frac{1}{k}\right)^k\right)$ and their average C_k

clauses. The last column is the expectation of the MIX algorithm. We now show that $C_k \geq 3/4$ for all k.

Let N_1 be the expected number of clauses satisfied by the algorithm of Theorem 5.2, i. e., by randomized rounding with $\hat{y}_i = 1/2$, for all i. Let N_2 be the expected number of clauses satisfied by randomized rounding with the $\hat{y}_i$ being a solution of the relaxed linear program. Due to the relaxation, we know that

$$N_{\max} \leq \sum_{j=1}^{m} \hat{z}_j \ .$$

We now show that $N_{\text{MIX}} = \max\{N_1, N_2\} \geq (3/4)\cdot\sum_{j=1}^{m}\hat{z}_j$, whence

$$r_{\text{MIX}} = N_{\text{MIX}}/N_{\max} \geq (3/4) \ .$$

By the well-known identity $\max\{N_1,N_2\} \geq (N_1+N_2)/2$, it suffices to show

$$\frac{N_1+N_2}{2} \geq \frac{3}{4}\sum_{j=1}^{m}\hat{z}_j \ .$$

Let k_j be the number of literals in clause c_j. Then (as $\hat{z}\in[0,1]$)

$$N_1 = \sum_{j=1}^{m}\left(1-2^{-k_j}\right) \geq \sum_{j=1}^{m}\left(1-2^{-k_j}\right)\cdot\hat{z}_j = \sum_{j=1}^{m} A_{k_j}\cdot\hat{z}_j$$

and

$$N_2 = \sum_{j=1}^{m}\left(1-\left(1-\frac{1}{k_j}\right)^{k_j}\right)\cdot\hat{z}_j = \sum_{j=1}^{m} B_{k_j}\cdot\hat{z}_j \ .$$

We show that $2C_k = A_k + B_k = \left(1 - 2^{-k}\right) + \left(1 - \left(1 - \frac{1}{k}\right)^k\right)$ is at least $3/2$ for all $k \geq 1$ (writing k instead of k_j for notational convenience). For $k = 1$ and $k = 2$ we have $A_k + B_k = (3/2)$. For $k \geq 3$, we have $A_k = \left(1 - 2^{-k}\right) \geq 7/8$ and $B_k = \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \geq 1 - \frac{1}{e}$. Hence,

$$\frac{7}{8} + 1 - \frac{1}{e} = 1.507\ldots \geq \frac{3}{2} \ .$$

This completes the proof. □

The following is an example of a relaxed linear program corresponding to a set of clauses.

Example 5.7 Let the Boolean variables x_1, x_2 and the 3 clauses $c_1 = x_1 \vee \bar{x}_2$, $c_2 = \bar{x}_1 \vee \bar{x}_2$ and $c_3 = x_2$ be given. One can easily check that at most two clauses can be satisfied. The relaxed linear program is:

$$\begin{array}{rcll} \textbf{max!} & & z_1 + z_2 + z_3 & \text{subject to} \\ y_1 + (1 - y_2) & \geq & z_1 & \\ (1 - y_1) + (1 - y_2) & \geq & z_2 & \\ y_2 & \geq & z_3 & \\ y_i & \geq & 0 \quad i = 1, 2 & \\ y_i & \leq & 1 \quad i = 1, 2 & \\ z_i & \geq & 0 \quad i = 1, 2, 3 & \\ z_i & \leq & 1 \quad i = 1, 2, 3 & \end{array}$$

The optimal solution for this program is

$$\hat{y}_1 = 0.5,\ \hat{y}_2 = 0.5,\ \hat{z}_1 = 1.0,\ \hat{z}_2 = 1.0,\ \hat{z}_3 = 0.5 \ ,$$

with value 2.5 for the target function. Randomized rounding will satisfy c_1 with probability $3/4$, c_2 with probability $3/4$, and c_3 with probability $1/2$. The expected number of satisfied clauses then is $3/4 + 3/4 + 1/2 = 2$.

5.4 A Randomized Algorithm for 3-Satisfiability

In this section, we reconsider the classical variant of 3-Satisfiability and state a randomized algorithm that solves the optimization problem (i. e., decides whether there is an assignment that satisfies *all* clauses and, if so, computes such an assignment) in

exponential time. However, the running time will be much lower than the 2^n bound we would obtain by the brute-force approach of testing all assignments. More precisely, the running time will be roughly 1.333^n, which, as we will see, makes a big difference.

Before we analyze the algorithm for 3-SATISFIABILITY, we look into the following variant of SATISFIABILITY called 2-SATISFIABILITY. We do this because many ideas of the 3-SATISFIABILITY algorithm can be understood from an algorithm for 2-SATISFIABILITY.

Problem 5.8 [2-SATISFIABILITY]
Input: A set of clauses $C = \{c_1, \ldots, c_k\}$ over n boolean variables $x_1, \ldots, x_n$, where every clause contains exactly two literals.
Output: YES if there is a satisfying assignment, i. e., if there is an assignment

$$a\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$$

such that every clause c_j is satisfied, and NO otherwise.

As the name suggests, 2-SATISFIABILITY works with clauses of length 2. In contrast to 3-SATISFIABILITY, there are several deterministic polynomial-time algorithms known for 2-SATISFIABILITY. That is: 2-SATISFIABILITY $\in \mathcal{P}$ and 2-SATISFIABILITY is NOT $\mathcal{NP}$-complete (unless $\mathcal{P} = \mathcal{NP}$). However, these deterministic algorithms are quite complicated. We present an amazingly simple randomized approach, see Algorithm 5.9. Its input is an instance to SATISFIABILITY, a parameter $T \in \mathbb{N} \cup \{\infty\}$ that determines the maximum number of trials for each initial guess, and a parameter $S \in \mathbb{N} \cup \{\infty\}$ that describes how often the algorithm should be rerun from a fresh guess.

As we can see, the initial guess is chosen uniformly at random from all 2^n assignments. The algorithm then repeatedly flips (until $t = T$ or a satisfying assignment has been found) the assignments to variables that appear as literals in unsatisfied clauses. Obviously, after flipping the assignment to such a variable, the previously unsatisfied clause will be satisfied. However, at the same time other clauses that contain the negation of the literal might become unsatisfied. How should the algorithm ever find a satisfying assignment to *all* clauses (assuming such an assignment exists)?

The answer is hidden in the analysis of a random process called *fair random walk*. Intuitively, the process can be considered as a token that travels on the set of integers within an interval between 0 and N. If it is not at a border (0 and N), then the token takes one step to the left or one step to the right, each with probability 1/2. In expectation, it therefore stays at the same place. If the token reaches a border, it is reflected back to the previous position. Interestingly, every state of the interval is reached in expected polynomial time.

Algorithm 5.9 [Randomized Satisfiability]
for $s \leftarrow 1, \ldots, S$ **do**
  for $i \leftarrow 1, \ldots, n$ **do**
    $x_i \leftarrow 1$ with probability 1/2, otherwise $x_i \leftarrow 0$.
    (Do this independently for each i.)
  end for
  for $t \leftarrow 1, \ldots, T$ **do**
    If $x = (x_1, \ldots, x_n)$ satisfies all clauses, output x. STOP.
    Uniformly select a clause c_j that is not satisfied by x.
    Choose uniformly a literal z from c_j. Let x_i be the variable that z is a literal of.
    $x_i \leftarrow 1 - x_i$.
  end for
end for
Output: "There is probably no satisfying assignment."

Theorem 5.10 (Fair random walk) *Consider a random process on the set* $\{0, \ldots, N\}$, *where* $N \in \mathbb{N}$, *and denote by* X_t *its state at time* t. *Suppose that for every* $i \in \{1, \ldots, N-1\}$ *and every* $t \geq 0$ *it holds that*

$$\boldsymbol{P}[\, X_{t+1} = i+1 \mid X_t = i \,] = \boldsymbol{P}[\, X_{t+1} = i-1 \mid X_t = i \,] = 1/2$$

and that $\boldsymbol{P}[\, X_{t+1} = 1 \mid X_t = 0 \,] = 1$ *as well as* $\boldsymbol{P}[\, X_{t+1} = N-1 \mid X_t = N \,] = 1$. *Then for any pair of states* $s, d \in \{0, \ldots, N\}$ *the expected number of steps to reach* d *starting from* s *is at most* N^2.

The proof of this theorem can be found in the literature of randomized algorithms. It allows us prove the following theorem on the performance of our randomized algorithm for the case of 2-Satisfiability.

Theorem 5.11 *If it is given a satisfiable* 2-Satisfiability *instance, then Algorithm 5.9 with* $T = \infty$ *and* $S = 1$ *outputs a satisfying assignment after an expected number of at most* n^2 *iterations of the inner loop over* t.

Proof. Since we assume a satisfiable instance, we can consider an arbitrary satisfying assignment, say $x^* = (x_1^*, \ldots, x_n^*)$ and analyze how long it takes the algorithm to find this assignment. Hereby we pessimistically ignore that it might find a different satisfying assignment before.

Let $d(x, x^*) := \sum_{i=1}^{n} |x_i - x_i^*|$ denote the distance of the current assignment x from the satisfying one. More precisely, $d(x, x^*)$ counts the number of variables that are differently assigned in x and x^*. Obviously, $0 \leq d(x, x^*) \leq n$ (i. e., the d-value is a random process on $\{0, \ldots, n\}$), and $d(x, x^*) = 0$ means $x = x^*$.

The crucial observation is that the d-value approaches 0 at least as fast as the fair random walk from Theorem 5.10. Let us consider the effect of choosing an unsatisfied clause c_j and flipping the assignment corresponding to a literal in it. Since x^* satisfies the clause but x does not, at least one of the two literals in c_j must have been assigned a different value in x^* and x. With probability 1/2, we choose this literal and decrease the d-value. Hereby we pessimistically ignore that also the other literal might have different assignments in x and x^*. Only with probability at most 1/2, we increase the d-value. According to Theorem 5.10, the expected number of iterations of the inner loop to reach a d-value of 0 is at most n^2, no matter what the initial d-value is. □

If there is no satisfying assignment, we should not set $T = \infty$ in Algorithm 5.9. Instead, we invoke it with $T = 2n^2$. Then a satisfying assignment will be found with probability at least 1/2 if it exists, otherwise the algorithm will output that there is probably no satisfying assignment. Recalling the randomized complexity classes, we obtain an interesting characterization.

Theorem 5.12 *Algorithm 5.9 invoked with $T = 2n^2$ and $S = 1$ is an $\mathcal{RP}$-algorithmn for* 2-SATISFIABILITY.

Proof. Obviously, if it is confronted with an unsatisfiable instance, then the algorithm will output that it is unsatisfiable after $2n^2$ iterations of the loop over t. Let us consider a satisfiable instance. By Markov's inequality (Formula (A.10) in the Appendix) and Theorem 5.11, the probability of not terminating within $2n^2$ iterations is at most 1/2. Each iteration is doable in polynomial time, so the total running time is polynomial. □

Of course, we can amplify the success probability of this $\mathcal{RP}$-algorithm by running it repeatedly and can then be 99% or 99.9% etc. sure that it gives a correct answer in polynomial time.

We now turn to 3-SATISFIABILITY. There is no place in the description of Algorithm 5.9 that restricts the input to 2-SATISFIABILITY instances. If we run it on an arbitrary 3-SATISFIABILITY instance, the following theorem holds. Here we for the first time use $S > 1$, i. e., we may start from a new random initial assignment (i. e., a new initial guess) if no solution has been found so far.

Theorem 5.13 *For a satisfiable* 3-SATISFIABILITY *instance, Algorithm 5.9 with $T = 3n$ and $S = \infty$ outputs a satisfying assignment in time at most $p(n) \cdot (4/3)^n$ for some polynomial $p(n)$.*

Proof. As in the proof of Theorem 5.11, let x be a current assignment (at $t < T$), x^* be the optimal assignment, and study how $d(x, x^*)$ changes over time. Since the clauses have length 3, the probability of decreasing the d-value is only bounded from below by 1/3, while the probability of increasing it might be 2/3. We are confronted with an "unfair" random walk.

Still, there is a positive (exponentially small probability) of reaching a d-value of 0 in $T = 3n$ steps. Here the quality of the initial guess x_{in}, which is drawn uniformly at random, comes into play. Let $d^* := d(x^*, x_{\text{in}})$ denote the distance from the initial guess to the satisfying assignment. To reach a d-value of 0, it is sufficient to consider the first $3d^* \leq 3n$ assignments created in the loop over t and to inspect the event that in the process of creating these, the distance is decreased at least $2d^*$ times (because then it is increased at most d^* times, altogether a decrease by d^*). The number of decreasing steps follows a binomial distribution (see Lemma B.5 for a definition), where the number of trials is $3d^*$ and the success probability of a trial is at least 1/3. Hence, the probability of the event is at least

$$\binom{3d^*}{2d^*} \left(\frac{1}{3}\right)^{2d^*} \left(\frac{2}{3}\right)^{d^*}$$

since there are $\binom{3d^*}{2d^*}$ different ways of having $2d^*$ decreasing steps within altogether $3d^*$ steps.

A famous estimation of binomial coefficients (see Lemma B.4 for more information) says

$$\binom{\beta}{\alpha\beta} \geq \frac{1}{\beta+1} \cdot \left(\alpha^\alpha \cdot (1-\alpha)^{1-\alpha}\right)^{-\beta}.$$

Using this with $\alpha = 2/3$ and $\beta = 3d^*$ bounds our probability from below by

$$\frac{1}{3d^*+1} \cdot \left(\left(\frac{1}{3}\right)^{1/3} \cdot \left(\frac{2}{3}\right)^{2/3}\right)^{-3d^*} \cdot \left(\frac{1}{3}\right)^{2d^*} \left(\frac{2}{3}\right)^{d^*}.$$

After some manipulations, this can be proved to equal

$$\frac{1}{3d^*+1} \cdot \left(\frac{1}{2}\right)^{d^*} \geq \frac{1}{3n+1} \cdot \left(\frac{1}{2}\right)^{d^*}.$$

The last expression still contains the random variable d^*. Using the law of total probability (A.18) with respect to the 2^n different outcomes of x_{in}, the success probability (i. e., the probability of finding an optimal assignment in T trials) is bounded from below by

$$\sum_y \boldsymbol{P}[\, x_{\text{in}} = y \,] \cdot \frac{1}{3n+1} \cdot \left(\frac{1}{2}\right)^{d(x^*,y)} = \frac{1}{3n+1} E\left(\left(\frac{1}{2}\right)^{d(x^*,x_{\text{in}})}\right).$$

Note that $d(x^*, x_{\text{in}}) := X_1 + \cdots + X_n$, where $X_i \in \{0, 1\}$ are independent random variables such that $X_i = 1$ if and only if d^* and x_{in} differ in bit i. Hence,

$$E\left(\left(\frac{1}{2}\right)^{d(x^*,x_{\text{in}})}\right) = E\left(\left(\frac{1}{2}\right)^{X_1+\cdots+X_n}\right) = E\left(\prod_{i=1}^{n}\left(\frac{1}{2}\right)^{X_i}\right) = \prod_{i=1}^{n} E\left(\left(\frac{1}{2}\right)^{X_i}\right),$$

where the last equality follows since the X_i are independent (see Equation (A.8)). Now, $E((1/2)^{X_i}) = (1/2)\cdot(1/2)^0 + (1/2)\cdot(1/2)^1 = 3/4$. Putting everything together, we have that our success probability is at least

$$\frac{1}{3n+1}\prod_{i=1}^{n}\frac{3}{4} = \frac{1}{3n+1}\left(\frac{3}{4}\right)^n.$$

If an event occurs in independent trials with success probability p each, then the expected number of trials until it occurs is at most $1/p$ (see Lemma A.2 in the appendix). Here a trial consists of an iteration of the outer loop and the success probability is at least $(3/4)^n/(3n+1)$. Hence, the expected number of runs of the outer loop (over s) is bounded from above by $O(n(4/3)^n)$. Since $T = O(n)$, the expected number of iterations is altogether $O(n^2(4/3)^n)$. Each iteration takes at most polynomial time. Altogether, the expected running time is at most $p(n) \cdot (4/3)^n$ for some polynomial $p(n)$. □

Applying Markov's inequality as above yields again a characterization as an algorithm with one-sided error.

Theorem 5.14 *Algorithm 5.9 invoked with $T = 3n$ and $S = 2p(n)(4/3)^n$, where $p(n)$ is the polynomial from Theorem 5.13, is an algorithm for* 3-SATISFIABILITY *with one-sided error and maximum running time $O(p'(n)(4/3)^n)$, where $p'(n)$ is also a polynomial. Errors are only possible for satisfiable instances.*

Finally, we give an example to show that it pays off to reduce the base of the exponential component in the running time from 2^n (the brute-force approach) to $4/3^n$ (the randomized approach). For $n = 25$ the former value is 33,554,432, while the latter is only about 1,329. This means that the randomized algorithm runs in maybe less than a second, while the brute-force approach is about 25,000 times slower.

5.5 Randomized Evaluation of Game Trees

Game trees are a means to find strategies for games played by two players where the moves alternate between player 1 and player 2. Let us look at the following (stupid) game *Sum-Is-Four*.

Example 5.15 Before the game, a number B is set to 0. At every round, the active player names a number n, either 1 or 2. The number is added to B, $B \leftarrow B + n$. In the next round, the other player is active. Player 1 starts. Player i, $i \in \{1, 2\}$, wins if he is active and makes $B = 4$ or if he is inactive and the other player makes $B > 4$. The possible games can be represented by the binary tree in Figure 5.1. The root is the start node where player 1 names the first number. If it is 1, we proceed to the left child; if the number is 2 to the right child. In the nodes, we write the values of B. At the leaves, the game is over. We indicate the winner. A game then corresponds to a path in the tree originating at the root. We want to know whether one of the players has a winning strategy, i. e., one player can always win regardless of the moves of the other player.

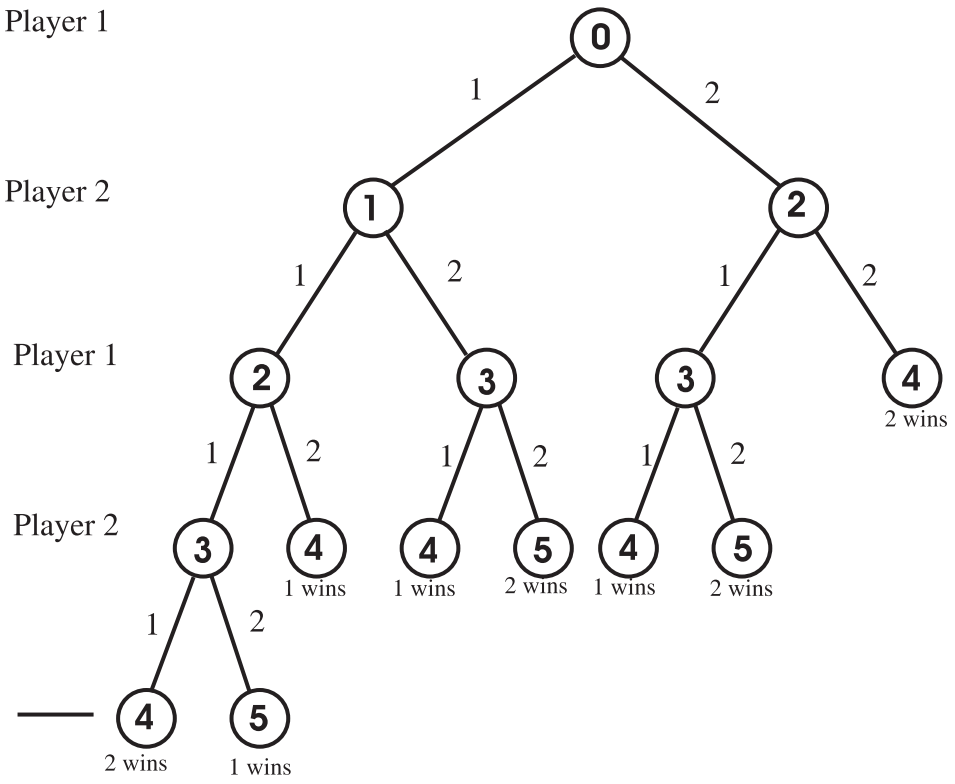


Figure 5.1: The game tree for the game described in the text. The numbers in the nodes are the values of B.

We draw the tree again (Figure 5.2) but with different labels at the leaves. A leaf gets label 1 if player 1 wins there and 0 if player 2 wins. Now consider an unlabeled internal node v whose two children are labeled. Assume first this node belongs to a level where player 1 has to make a move (name a number). If both children w_1 and w_2 are labeled 0 then both are non-wins for player 1. Regardless if he says 1 or 2, he will lose if he is in v. So v is assigned the label 0 (win for player 2). If both children w_1 and w_2 are labeled 1

then both are wins for player 1. Regardless if he says 1 or 2, he will win if he is in v. So v is assigned the label 1 (win for player 1). If w_1 is labeled 1 and w_2 labeled 0 then player 1 can make the move (name the number) which gets him into w_1. If w_1 is the left child, he says 1. Thus, he can win if he is in v and player 2 can make no move later on to avoid this. So v is assigned the label 1 (win for player 1). Analogously, v is labeled 1 if w_2 is labeled 1 and w_1 is labeled 0. Hence, on a level where it is player 1's turn, a node v is labeled by the **maximum** of the labels of its children. The reader should check that at a level where it is player 2's turn a node v is labeled by the **minimum** of the labels of its children. Thus ensuring that player 1 will not win if this can be avoided.

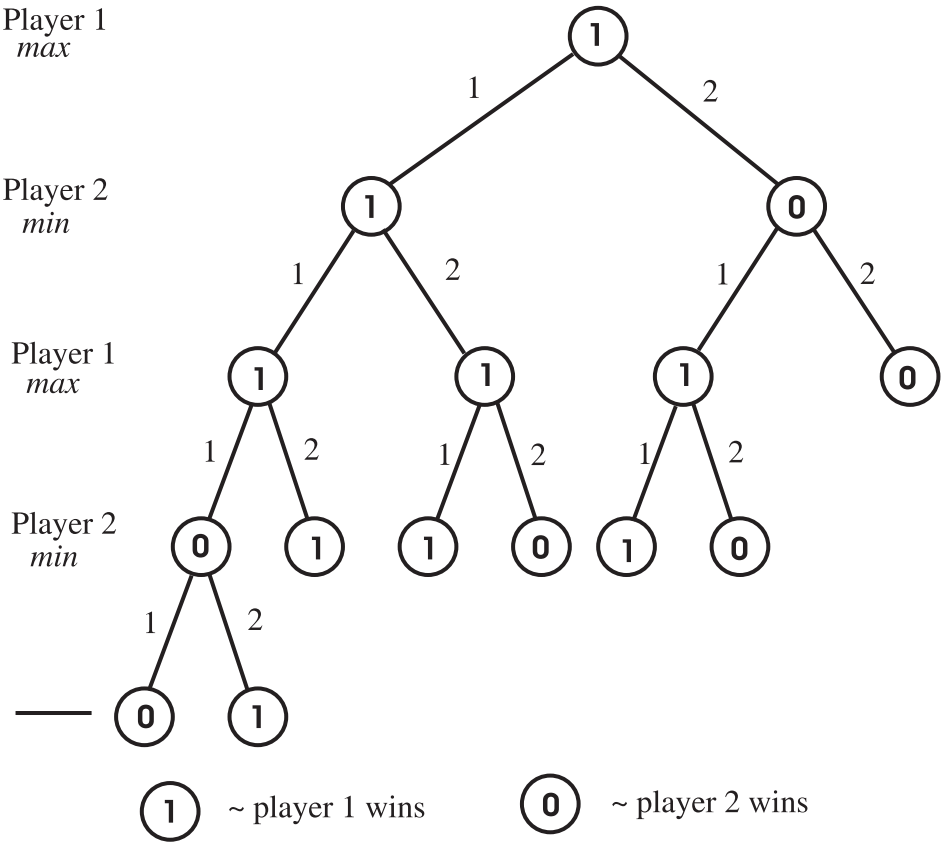


Figure 5.2: The game tree with the nodes labeled 0 (player 2 wins) or 1 (player 1 wins). The labels at the leaves are obvious from the rules of the game. Those at the internal nodes are computed by taking minima or maxima over the labels of the children. Player 1 has a winning strategy: He can play in such away that he always reaches a node labeled by 1. The possible sequences of moves are: `1-1-2` and `1-2-1`.

The labeling is done upwards starting at the nodes that have two leaves as children. If the label of the root is 1 then player 1 has a winning strategy. Otherwise, player 2 has a winning strategy. In our example, it is player 1. Here is the strategy: Player 1 starts in a node labeled 1 (the root). If it is player 1's turn and the current node is labeled 1 then (by construction of the labels) at least one child is labeled 1. The player 1 makes

a move that takes him into that node. But then it is player 2's turn in a node with label 1. By construction of the labels, both children are labeled 1 and regardless of the move of player 2, we will be in a node labeled 1 afterwards.

In general the labels of a game tree are more complicated. In chess, it is not possible to write down the complete game tree because it takes too much memory. Thus, the game tree is only constructed for a few levels (typically 6 to 10 in commercial chess programs and up to 15 in specialized super computers). At the root is the current configuration on the board. Also the tree is no longer binary as many moves are possible in chess. Then it is not clear whether a leaf is a winning or losing configuration for player 1, because the game is not over at the leaves. By looking at the configuration in the leaves, an expert might be able to say whether it "looks better" for player 1 or player 2. This can be done by assigning a real number $r \in [0, 1]$ as a label. Values close to 0 indicate that it looks better for player 2. In practice, some heuristic is used to evaluate the configurations at the leaves and compute the labels. Once that is done, we use the strategy of the example. If a node v is at an even level $0, 2, 4, \ldots$ (player 1 levels) and has children $w_1, w_2, \ldots, w_k$ then we set

$$\ell(v) = \max\{\ell(w_1), \ell(w_2), \ldots, \ell(w_k)\} ,$$

where $\ell(v)$ is the label of v. Add odd levels (player 2 levels) we set

$$\ell(v) = \min\{\ell(w_1), \ell(w_2), \ldots, \ell(w_k)\} .$$

Let us now consider the special case where

- the labels are from $\{0, 1\}$ and
- all leaves are at level $2k$ (which implies that the depth of the tree is $2k$),
- every internal node has exactly 2 children.

Such a game tree has $N = 2^{2k}$ leaves. The problem now is:

Problem 5.16 [GameTreeEvaluation]
Input: A game tree with the properties just stated and an assignment of $\{0, 1\}$-labels to the leaves.
Output: The label of the root.

Let A be an algorithm for this problem. We use the number of leaves which A looks at as the measure of complexity. One can show that for every deterministic algorithm A there is an assignment of labels to the leaves such that A has to look at all leaves. Hence the worst-case complexity of this problem is N.

Here is how we can save time by randomization: Let v be a node at an even level, i. e., a max-node. If we look at only one child and that is labeled 1 then v will be labeled 1 regardless of the other child. If v is at an odd level (min-node) then if one child is 0 the label of v is zero. The algorithm starts at the root and recursively evaluates the nodes. It chooses one child at random and only if that does not determine the label, the other child is also evaluated. We assume at this point that a node has none or two children.

Algorithm 5.17 [Evaluate]

```
v ← root
result ← evaluate(v)

proc evaluate(v)
if v is a leaf then
  return(ℓ(v))
else
  let w1 and w2 be the children of v
  pick one child with probability 1/2 at random; call this a and the other b
  t ← evaluate(a)
  if (v is max-node) ∧ (t == 1) then
    return(1)
  else if (v is min-node) ∧ (t == 0) then
     return(0)
  else
    return(evaluate(b))
  end if
end if
end proc
```

Let T be a game tree as described above (binary of depth $2k$) and let $M(T)$ be the expected number of leaves that algorithm *Evaluate* looks at. Let $M_{\max}(k) := \max\{M(T) \mid T \text{ is a game tree as above of depth } 2k.\}$.

Consider a tree T of depth $2k+1$ with a min-node v at the root. Node v has subtrees T_1 and T_2 both of depth $2k$. Three cases are possible.

Case 1 If both subtrees have a 0-label at their roots then:

$$M(T) = \frac{1}{2}M(T_1) + \frac{1}{2}M(T_2) \leq M_{\max}(k).$$

This is because T_1 is chosen with probability $1/2$ to be evaluated first; same for T_2. Independently of which tree is chosen, its root value 0 determines the minimum to be 0. The other sub-tree of v is not evaluated. Hence, the chances to have the time $M(T_1)$ for evaluation T_1 is $1/2$, same for T_2.

Case 2 If one subtree – say T_1 – computes a 0 and the other a 1 then:

$$M(T) = \frac{1}{2}M(T_1) + \frac{1}{2}\left(M(T_2) + M(T_1)\right) \leq 1.5\, M_{\max}(k).$$

Because with probability $1/2$ the tree T_1 with root label 0 is chosen and only that one has to be evaluated. With probability $1/2$ the tree T_2 with root label 1 is chosen first. After T_2 is evaluated and it is clear that its root label is 1, we have to evaluate T_1 in addition to determine the minimum to be 0. In all cases $M(T) \leq 1.5 M_{\max}(k)$.

Case 3 Both subtrees have root label 1. Then after evaluating one of them, the minimum cannot be determined and the other one has to be evaluated as well. The minimum is 1.

$$M(T) = M(T_2) + M(T_1) \leq 2\, M_{\max}(k).$$

Altogether we have

Fact 1 If the label of v is 0, the time is at most $1.5\, M_{\max}(k)$

Fact 2 If the label of v is 1, then $2\, M_{\max}(k)$ are sufficient.

The above considerations apply in the case that v is max-node with the obvious changes.

Claim 5.18 *For all game trees as above with depth $2k$, $M(T) \leq N^{0.793}$ and $M_{\max}(k) \leq N^{0.793}$.*

Proof. The proof is by induction on k and shows that $M_{\max}(k) \leq 3^k$. The claim then follows as

$$3^k = 2^{k \log_2(3)} = 4^{(k/2)\log_2(3)} = \left(4^k\right)^{\log_2(3)/2} = N^{\log_2(3)/2} = N^{0.793}\ .$$

In one step of the induction, we consider two levels of the tree, the top-most being a max-level. The induction starts with $k = 0$. Then the tree T is just a leaf. Then $M(T) = 1 \leq 3^0$.

Now consider the situation as in Figure 5.3 with 4 depth-$2k$ sub-subtrees T_1, T_2, T_3, T_4. The whole tree T has depth $2k + 2$.

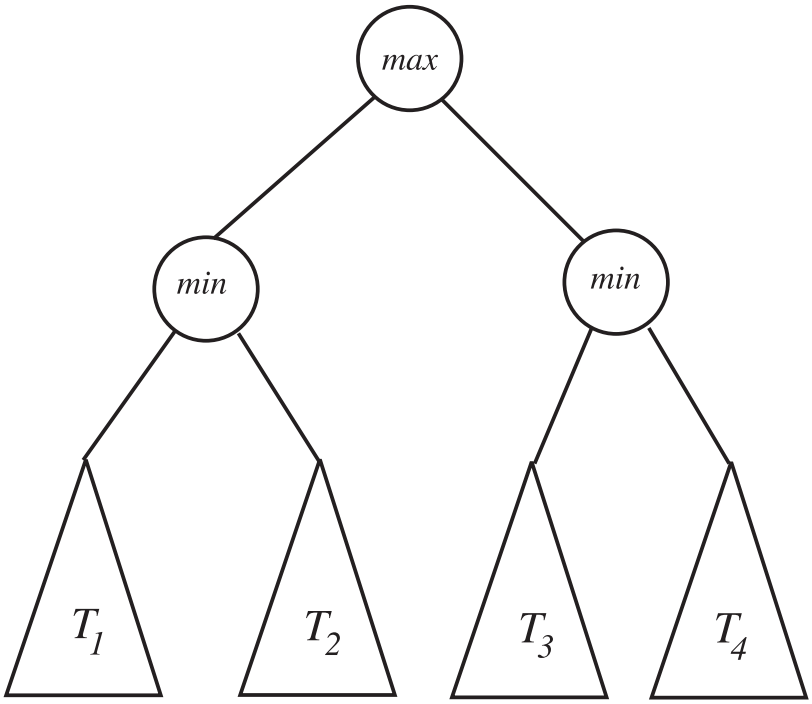


Figure 5.3: Example for the randomized evaluation of a game tree.

We consider 3 cases:

Case 1 Both min-nodes compute a 0. Then the algorithm evaluates both. By Fact 1, we have

$$M(T) \leq 1.5M_{\max}(k) + 1.5M_{\max}(k) = 3M_{\max}(k) \ .$$

Case 2 One min-node computes a 0 (w. l. o. g., the left one) and the other a 1. The right one is chosen at first (as node a in the algorithm) with probability $\frac{1}{2}$. Then the (in expectation) $2M_{\max}(k)$ leaves are considered (in T_3, T_4). If the left node is chosen first then by Fact 1 and Fact 2 (in expectation) $1.5M_{\max}(k) + 2M_{\max}(k) = 3.5M_{\max}(k)$ leaves are considered because the other tree has to be evaluated as well. Altogether:

$$M(T) \leq \frac{1}{2}\left(2M_{\max}(k) + 3.5M_{max(k)}\right) < 3M_{\max}(k) \ .$$

Case 3 Both min-nodes compute a 1 . Then only one is evaluated because it determines the max-node to be 1. By Fact 2 we have

$$M(T) \leq 2M_{\max}(k) < 3M_{\max}(k) \ .$$

In any case, we have $M(T) \leq 3M_{\max}(k)$. Now the induction hypothesis is that $M(k) \leq 3^k$. Using this, we derive

$$M_{\max}(k+1) \leq 3M_{\max}(k) \leq 3 \cdot 3^k = 3^{k+1}.$$

□

5.6 Introduction to Randomized Search Heuristics (RSH)

This section and the two following ones deal with randomized search heuristics for optimization problems, more precisely bio-inspired randomized search heuristics such as evolutionary algorithms, simulated annealing etc. Such general heuristics are applied "off-the-shelf" when it is not possible to apply an algorithm tailored to the problem at hand. Reasons for this are lack of resources such as time, money, knowledge etc. We will describe how randomized search heuristics can both be applied to solve optimization problems, and how they can be analyzed with respect to their running time.

5.6.1 Evolutionary Algorithms

There are numerous variants of randomized search heuristics, with evolutionary algorithms (EAs) still being the most prominent example. In general, an evolutionary algorithm is centered around the following concepts and modules:

Search space and fitness function. Generally, the aim is to find an optimum (minimum or maximum) of a so-called *fitness function* (also called objective function) $f\colon S \to \mathbb{R}$ mapping points from a *search space* S to a real number. The search space is also called domain and represents all possible solutions to the problem. We consider the traveling salesman problem as an example. Then S could be the set of all permutations on $\{1, \ldots, n\}$, corresponding to a Hamilton circuit through the underlying weighted graph. The fitness function f, applied to a concrete permutation $(i_1, \ldots, i_n)$, would then give the total length of the tour represented by the permutation. We are then interested in finding a search point of minimum value.

A simple search space is $\{0,1\}^n$, the space of all bit strings of length n. For example, the assignment of n boolean variables for the MAX-SAT problem can be described in this way. The fitness function could return the number of satisfied clauses under a given assignment.

Population and generational procedure. Usually, an EA maintains several search points from the search space S simultaneously. This collection is called a *population* and its elements are called *individuals*. Formally, the population P is a multi-set of search points from S as we do not rule out duplicates. The size of P is typically kept fixed and denoted by μ. An EA typically proceeds in rounds called *generations*. A generation takes the current population P and creates from it a new population P' for the next generation by applying *selection*, *mutation* and *crossover* (all explained below) in some way.

Intialization. It has to be decided what the initial population has to look like. If no prior knowledge on the structure of good solution is present, than the initial population consists of μ individuals that are drawn uniformly at random from S.

Selection. As a first step in creating the new population P', one or several individuals have to be selected from the current population P. Typically, individual of better fitness value are more likely to be selected than individuals of bad fitness. However, selection could also choose individuals uniformly at random without taking their fitness into account.

A well-known selection operator called *tournament selection* with tournament size k works as follows: it temporarily chooses k individuals from the current population uniformly at random and outputs the one of best fitness as result of the selection process. Note that tournament size 1 would result in a completely uniform choice, whereas tournament size $|P|$ corresponds to selecting the fittest individual for sure.

Usually, the selection operators of an EA crucially depend on the fitness values of the individuals whereas the following operators do not use the fitness values.

Mutation. This is an operator (formally, a stochastic mapping) transforming an individual $x \in S$ into a new individual $x' \in S$. Often, x' is obtained by only changing x' to a minor extent but in a randomized fashion. For example, assuming x is a permutation $(i_1, \dots, i_n)$, the so-called swap mutation picks two indices $k, \ell \in \{1, \dots, n\}$ uniformly at random and swaps the elements at these indices, resulting in $x' = (i_1, \dots, i_{k-1}, i_\ell, i_{k+1}, \dots, i_{\ell-1}, i_k, i_{\ell+1}, \dots, i_n)$ if $k < \ell$. Another example is the so-called bit-flip mutation: given a bit string $(x_1, \dots, x_n)$, the operator chooses one index $k \in \{1, \dots, n\}$ and flips bit k. Formally, $x' = (x_1, \dots, x_{k-1}, 1 - x_k, x_{k+1}, \dots, x_n)$.

Crossover. This is a operator taking two individuals (so-called parents) $x, y \in S$ as input and producing a new individual (child) $z \in S$ (or two children z_1, z_2) as output. Typically, the idea is that z combines x and y in a randomized fashion. For example, assuming $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_n)$ are bit strings of length n, the so-called *uniform crossover* would choose each bit of z uniformly from either x and y; formally, $z_i = x_i$ with probability $1/2$ and $z_i = y_i$ with probability $1/2$, uniformly at random for all $i \in \{1, \dots, n\}$.

It is not obvious to design crossover operators for the search space of permutations. Several different approaches are discussed below in Section 5.6.2.

We present a generic pseudo-code of an EA in Algorithm 5.19. Although this scheme can be considered typical, it does not cover all variants of EAs. For example, EAs with varying population size or multiple populations are outside the scope of the scheme. We also remark that often mutation and crossover are only applied with a certain probability. For example, z' could be a copy of z with probability 2/3 and only with probability 1/3 mutation is applied. If crossover is only applied with a certain probability, then z would be set to either x or y if the step omits crossover.

Algorithm 5.19 [Generic scheme of an evolutionary algorithm]

Initialize population P_0 of size μ.
$t \leftarrow 0$.
while stopping criterion not fulfilled **do**
  $P_{t+1} := \emptyset$.
  for $i \leftarrow 1, \ldots, \mu$ **do**
    Choose two individuals x and y from P_t by applying some selection operator.
    Create z by applying some crossover operator to x and y.
    Create z' by applying some mutation operator to z.
    Add z' to P_{t+1}.
  end for
  $t \leftarrow t + 1$.
end while

5.6.2 Evolutionary Algorithms for the Traveling Salesman Problem

We describe possible implementations of EAs for the TSP (see Problem 4.6). The search space is naturally given by the space of all permutation on $\{1, \ldots, n\}$, where n is the number of vertices of the underlying graph and the fitness function is given by the length of the tour a permutation represents. The performance of an EA for the TSP still depends crucially on the choice of selection, crossover and mutation operators. As selection operator tournament selection (mentioned above) has turned out as good choice.

Well-established mutation and crossover operators for the TSP are discussed in the following. Although all these operators are defined with respect to arbitrary distance matrices, many of them implicitly assume that the instance has some properties that are beneficial for the operator. For example, the 2-OPT mutation operator and PX crossover try to exploit that many TSP instances are symmetric, i. e., the cost of edge (i, j) is the same as the cost of edge (j, i).

Mutation operators

Swap mutation This is the simple mutation operator described in Section 5.6.1.

Jump mutation Similarly to the swap mutation, two indices $k, \ell \in \{1, \dots, n\}$ are chosen uniformly at random. Without loss of generality, $k < \ell$. The element at position k jumps to position ℓ and all elements from positions $k+1, \dots, \ell-1$ are shifted one position to the left. Formally, $x' = (i_1, \dots, i_{k-1}, , i_{k+1}, \dots, i_{\ell-1}, i_\ell, i_k, i_{\ell+1}, \dots, i_n)$.

2-OPT This is the most established and often best-performing operator. In contrast to Swap and Jump, it directs its attention on the edges traversed by the tour. Given a tour $(i_1, \dots, i_n)$, it chooses two indices $k, \ell \in \{1, \dots, n\}$, where $k < \ell + 1$, uniformly at random and takes out the edges (i_k, i_{k+1}) and $(i_\ell, i_{\ell+1})$ (here and in the following notation wraps around after index n, i. e., $i_{n+1} = i_1$). Since the two edges do not share a common endpoint, this breaks the tour into two subtours $(i_{k+1}, \dots, i_\ell)$ and $(i_{\ell+1}, \dots, i_k)$. Then the edges (i_k, i_ℓ) and $(i_{\ell+1}, i_{k+1})$ are introduced to reconnect the subtours across. The new tour is $(i_1, \dots, i_k, i_\ell, i_{\ell-1}, \dots, i_{k+1}, i_{\ell+1}, \dots, i_n)$. See Figure 5.4 for an illustration. Note that the order of the segment $(i_{k+1}, \dots, i_\ell)$ is reversed by the operator.

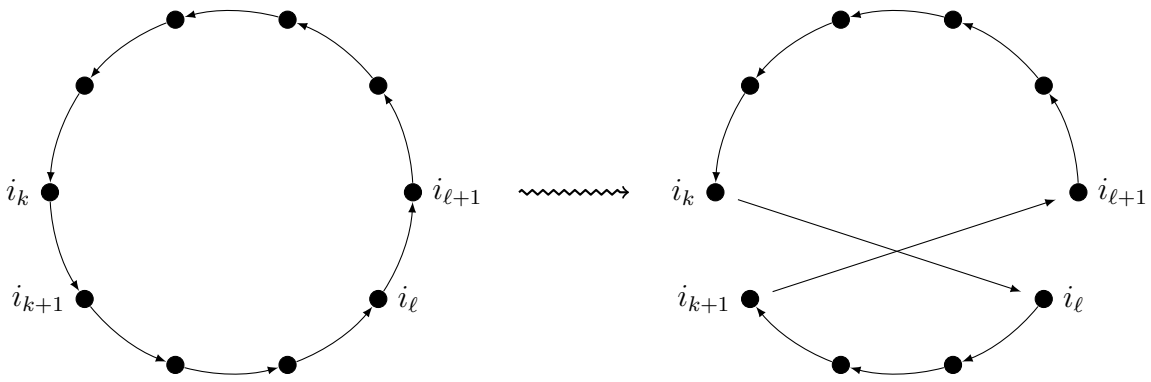


Figure 5.4: Example of 2-OPT mutation operator

A generalization called k-OPT exists that takes out k edges and then reconnects the subtours in some crossing fashion.

Crossover operators

Designing crossover operators for permutations is not trivial. The challenge is to combine to parents $(x_1, \dots, x_n)$ and $(y_1, \dots, y_n)$ by taking elements from both parents while

ensuring that the result is still a permutation on $\{1, \ldots, n\}$. For example, if $x = (1, 2)$ and $y = (2, 1)$ then uniformly taking one of the elements of x and y could result in $(1, 1)$ or $(2, 2)$, neither of which are permutations.

Several crossover operators for permutations have been designed. We present some well-known examples, both classical and more recent ones.

Order crossover (OX) This operator tries to preserve the order in which cities are visited in one of the parents as much as possible. It chooses two indices $k, \ell \in \{1, \ldots, n\}$, where $k < \ell$, uniformly at random. The child z takes the segment $(x_k, \ldots, x_\ell)$ from the first parent. The remaining positions, starting from $\ell + 1$ and wrapping around, are filled consecutively by taking elements from the second parent (also starting at position $\ell + 1$ and wrapping around), skipping any occurrences of the already copied elements $x_k, \ldots, x_\ell$.

For example, if the parents are

$$x = (1, 2, 3, 5, 4, 6, 7, 8, 9)$$

and

$$y = (4, 5, 2, 1, 8, 7, 6, 9, 3)$$

and $k = 4$, $\ell = 7$, then the child will be

$$(2, 1, 8, \mathbf{5}, \mathbf{4}, \mathbf{6}, \mathbf{7}, 9, 3).$$

The elements inherited from the first parent are printed in bold face.

If two children have to be created, then the second one is created in the same way but with the roles of x and y swapped.

Partially Matched Crossover (PMX) The intuition for this crossover is to preserve the absolute positions in which cities are visited to a large extent. As in order crossover, two indices are chosen and the child z takes the segment $(x_k, \ldots, x_\ell)$ from the first parent. The remaining elements are added from the second parent by preserving absolute positions as far as possible. The elements j from the second parent that cannot be put at the same position are filled with the elements that appear at the same position in y as j in x, possibly repeating this lookup until a valid element is found.

Reusing the example from above, the child starts out as

$$(*, *, *, \mathbf{5}, \mathbf{4}, \mathbf{6}, \mathbf{7}, *, *)$$

All elements but 4 and 5 can be taken from y and put at the same position. The intermediate result is

$$(*, *, 2, \mathbf{5}, \mathbf{4}, \mathbf{6}, \mathbf{7}, 9, 3)$$

The element 4 from the first position of y shows up at x_5. Hence, the first element of z becomes $y_5 = 8$. As the element 5 is in x_4, the second position becomes $y_4 = 1$. The final result is

$$(8, 1, 2, \mathbf{5}, \mathbf{4}, \mathbf{6}, \mathbf{7}, 9, 3)$$

The second child can again be obtained by swapping x and y.

There are variations of PMX in the literature that are defined slightly differently to the above.

Cycle Crossover (CX) The idea of this operator is to maintain many absolute positions of the first parent by identifying a cycle through the parents. We consider the two parents to define a mapping f on the indices: $f(x_i) := y_i$ for $i \in \{1, \ldots, n\}$. We start out by copying x_1 to z_1. Then the element of x that equals $y_1 = f(x_1)$, say x_j, is copied to z at the same position j. We proceed by finding the element of x equaling $f(x_j)$ etc. until we hit an element that has already been copied to z. The remaining positions, starting to the right from the last copied element, are filled with the missing elements in the order they appear in y.

For example, again assuming the parents

$$x = (1, 2, 3, 5, 4, 6, 7, 8, 9)$$

and

$$y = (4, 5, 2, 1, 8, 7, 6, 9, 3),$$

the child starts out as $(1, *, \ldots, *)$. As $y_1 = 4$, the next element added is the 4 at position 5: $(1, *, *, *, 4, *, *, *, *)$. Now $y_5 = 8$, hence, 8 is added at position 8. Proceeding in this way, $(1, 2, 3, 5, 4, *, *, 8, 9)$ is obtained. As 5 maps to 1, which is already contained in y, we finally fill the remaining two positions with entries from y, resulting in $(\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{4}, 7, 6, \mathbf{8}, \mathbf{9})$.

Partition Crossover (PX) and Generalized Partition Crossover (GPX) Unlike the previously described crossover operators, which focused on the cities visited in the parents, partition crossover focuses on the *edges* and derives a partition of the cities based on common edges. More precisely, each of the two parents tours x and y is identified with an undirected Hamilton cycle C_x and C_y, respectively, on a graph with n vertices. Note that the same cycle C_x is represented by n different permutations obtained by cyclically shifting x by a certain amount of positions. Next the union graph G^* obtained by adding edges from the two cycles is considered. Edges in G^* that are part of both parent cycles are called common edges.

In G^*, a cut of the vertex set is sought that cuts exactly two common edges, say e and e' (see also the MaxCutproblem defined on on page 61 but note that we are looking

for a small cut). If there is no such cut, partition crossover is not possible and the child/children returned by crossover equal (one of) the parents. Otherwise, let V_1 and V_2 be a partition with the required properties. Let $C_x \cap V_i$, where $i \in \{1, 2\}$, denote the part of the cycle C_x using only vertices from V_i; analogously for C_y. Note that $C_x \cap V_i$ induces a path on V_i due to the properties of the cut. It may contain common edges except for e and e'.

The first child of partition crossover is now obtained by concatenating the paths $C_x \cap V_1$, the common edges e and e' and the path $C_y \cap V_2$; analogously with swapped roles of V_1 and V_2 for the second child. By definition, these concatenations are Hamilton paths.

For example, let the parents be

$$x = (5, 12, 11, 2, 1, 3, 4, 6, 10, 8, 7, 9)$$

and

$$y = (4, 10, 6, 7, 8, 9, 5, 11, 12, 1, 2, 3)$$

See Figure 5.5. A possible cut that cuts only two common edges is $V_1 = \{1, 2, 3, 5, 11, 12\}$ and $V_2 = \{4, 6, 7, 8, 9, 10\}$, where the common edges are $e = \{3, 4\}$ and $\{5, 9\}$. We have $C_x \cap V_1 = (5, 12, 11, 2, 1, 3)$, $C_y \cap V_1 = (5, 11, 12, 1, 2, 3)$, $C_x \cap V_2 = (9, 7, 8, 10, 6, 4)$ and $C_y \cap V_2 = (9, 8, 7, 6, 10, 4)$. Here we see that there are more common edges in the union graph than just e and e', e. g., $(11, 12)$ and $(7, 8)$. The result of crossover (more precisely, the first child) is

$$(5, 12, 11, 2, 1, 3, 4, 10, 6, 7, 8, 9),$$

or any other cyclic shift of this permutation. We note that the path $(5, 12, 11, 2, 1, 3)$ was inherited from x and the rest from y.

PX has two structural properties: it preserves the relative order in which cities are visited from one of the parents and it also preserves the absolute positions of the cities with respect to one of the Hamilton cycles, given a fixed starting city. In this sense, it combines the aims of the above described operators OX and PMX. We remark here without proof that PX can be implemented to run in time $O(n)$.

The choice of the two common edges $\{5, 9\}$ and $\{3, 4\}$ in the above example was the only possible way to obtain a partition of the above union graph by cutting only two common edges. In general, there may be more than one cut of size two that partitions the union graph. Partition crossover as described above would only consider one of these partitions. To gain from other choices of the partition, a generalization called *generalized partition crossover* (GPX) has been proposed. It partitions the graph into possibly more than two components by taking out all common edges and then determining for each component the shortest path through the component based on the tours from the two parents. The concatenation of these tours then gives rise to an offspring. See Figure 5.6 for an example. There are 6 common edges, which, after taking them out, partition three graph into three connected components. Through each

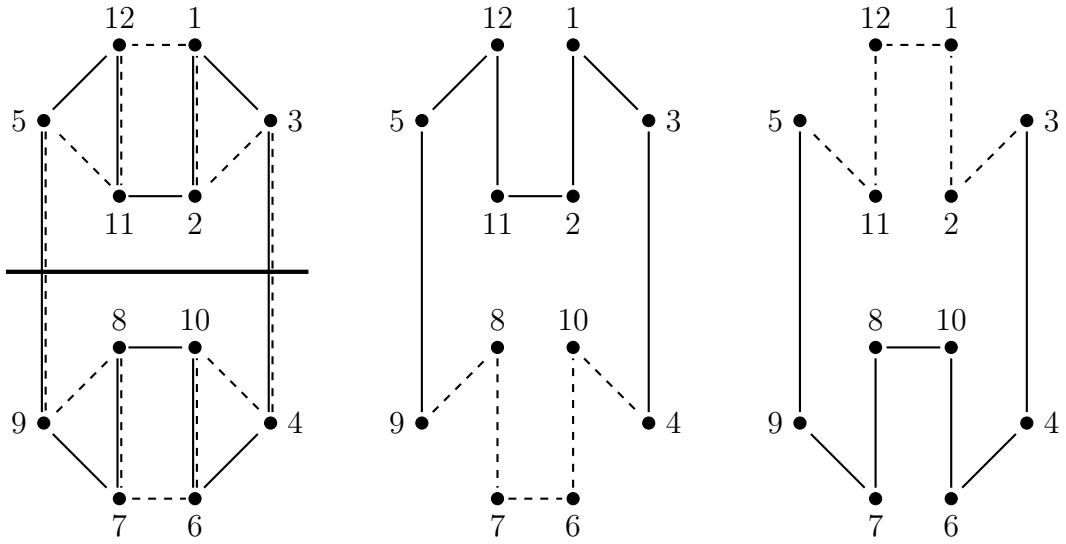


Figure 5.5: Example for partition crossover: the leftmost graph contains the union of two TSP tours (dashed and solid lines). A cut through two common edges (thick horizontal lines) is used to partition the graph, resulting in the two offspring in the middle and on the right-hand side.

connected component there are 2 possible paths, so ordinary partition crossover could produce up to $2^3 = 8$ different results depending on the choice of the two common edges to cut. GPX automatically determines the shortest of these crossover results by computing the shortest path through each connected component separately. We remark that special care has to be taken if there exists connected components entered and exited more than once in a tour. The details are outside the scope of this course, as are additional enhancements of GPX for, e. g., asymmetric distance matrices.

In general, if there are k connected components after taking out the common edges, GPX determines the shortest out of up to 2^k different crossover results. Again, without going into the details, we remark that also GPX can be implemented in time $O(n)$ using advanced data structures.

This advanced crossover operator is one of the most efficient ones known today for the TSP.

5.7 Analysis of Randomized Search Heuristics on Simple Problems

Practitioners report surprising success of randomized search heuristics on optimization problems, but a theoretical foundation has been built up only recently. The aim of this last part of the course is to contribute to the understanding of randomized search heuristics from an *algorithmic* viewpoint using methods from classical theoretical computer

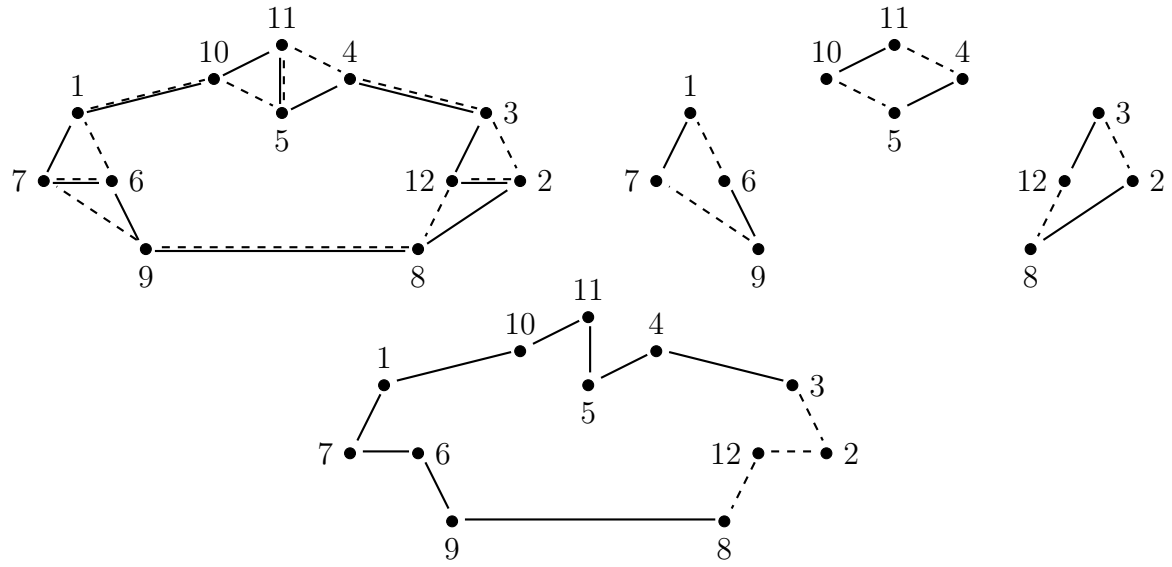


Figure 5.6: Example for generalized partition crossover: the top leftmost graph contains the union of two TSP tours (dashed and solid lines). The partition of the graph into connected components is depicted on the top right-hand side. Based on this partitioning, the edges $\{1, 10\}$, $\{8, 9\}$ and $\{3, 4\}$ are derived as cut edges and the final result of crossover is shown at the bottom.

science. We will mostly consider simple problems. The literature contains examples showing that randomized search heuristics can solve $\mathcal{NP}$-hard problems efficiently.

5.7.1 Preliminaries

We consider two very simple randomized search heuristics for the search space $\{0,1\}^n$ (i. e., bit strings of length n), where the aim is to find a bit string that optimizes a fitness function $f\colon \{0,1\}^n \to \mathbb{R}$. The search heuristics are called randomized local search (RLS) (Algorithm 5.20) and (1+1) EA (Algorithm 5.21). They only differ in Step 3, the mutation. Common to both is that they start with a uniformly at random selected search point, proceed in rounds and replace search points if and only if the new search point created by mutation is at least as good as the previous one. The mutation operator of RLS flips exactly one bit, while the number of bits flipped by the (1+1) EA is random. It may be between 0 and n and is $n \cdot (1/n) = 1$ in expectation.

The algorithms as presented here do not use a stopping criterion, which would be needed in practical implementations. For our theoretical purposes, we can analyze the search heuristics as infinite stochastic processes. We are mainly interested in the first point of time $T_{A,f}$ such that algorithm A (so far only RLS or (1+1) EA) has created a search point that is optimal for f. Each time step contains one evaluation of f, hence $T_{A,f}$ basically counts the number of f-evaluations until reaching the optimum, which is the common cost measure (sometimes called black-box measure) in the analysis of randomized search heuristics. The reason is that an evaluation of a complicated fitness

function is typically the most expensive part in experiments, while the other steps of the algorithm are cheap and computationally neglegible. Since we do not always know how the concrete fitness function works (it is a so-called black box), we always set up one time unit for an evaluation.

We call $T_{A,f}$ the *optimization time* or simply *runtime* of A on f and note that this is a random variable. As usual in the analysis of randomized algorithms, the expected value $\boldsymbol{E}[\,T_{A,f}\,]$ is of particular interest and is referred to as *expected optimization time* or, synonymously, *expected runtime*. Finally, optimization may mean that we minimize or maximize a function. The algorithms are defined for maximization problems, which is without loss of generality since maximizing f is equivalent to minimizing $-f$.

Algorithm 5.20 [Randomized Local Search (RLS)]

1. $t := 0$. Choose $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ uniformly at random.
2. $y := x$
3. Choose one bit in y uniformly at random and flip it *(mutation)*.
4. If $f(y) \geq f(x)$ Then $x := y$ *(selection)*.
5. $t := t + 1$. Continue at line 2.

Algorithm 5.21 [(1+1) Evolutionary Algorithm ((1+1) EA)]

1. $t := 0$. Choose $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ uniformly at random.
2. $y := x$
3. Independently for each bit in y: flip it with probability $\frac{1}{n}$ *(mutation)*.
4. If $f(y) \geq f(x)$ Then $x := y$ *(selection)*.
5. $t := t + 1$. Continue at line 2.

5.7.2 (1+1) EA and RLS on Example Problems

Randomized search heuristics were defined without a theoretical analysis in mind. Instead, the aim was to follow biological ideals such as evolution ("survival of the fittest"), which makes them harder to analyze than classical algorithms, where the

designer had a proof of correctness, runtime etc. already in mind when publishing the algorithm. In order to develop first results, initial research focused on so-called example problems of a simple structure. Even on these problems, it is not obvious how randomized search methods behave. Methods from probability theory and algorithmics had to be adjusted to randomized search heuristics and the stochastic processes behind them. In the course of this and the subsequent section, we develop proof techniques along with the results presented.

5.7.3 Upper Bounds

We start with a very general result, whose proof presents the first general technique.

Theorem 5.22 *Let $f\colon \{0,1\}^n \to \mathbb{R}$ be arbitrary. The expected optimization time of the (1+1) EA on f is at most n^n.*

Proof. Let x^* be a search point that has maximal f-value. For any current search point x, the probability of creating x^* by mutation of x is

$$\left(\frac{1}{n}\right)^{H(x,x^*)} \left(1-\frac{1}{n}\right)^{n-H(x,x^*)},$$

where $H(x,x^*)$ is the number of bits that differ in x^* and x (sometimes called the *Hamming distance*). The reason is that exactly those bits where x and x^* differ need to be flipped (which happens with probability $1/n$ each) and the others have to be kept (which happens with probability $1-1/n$ each). Obviously, the probability becomes smallest (assuming $n \geq 2$) if $H(x,x^*) = n$ and is $(1/n)^n$ then. In each step, we have this minimum probability, hence the waiting time argument (see Lemma A.2 in the appendix) tells us that we have to wait for an expected number of at most $1/(1/n)^n = n^n$ steps until x^* is created. □

For RLS, we do not have any general upper bound on the expected optimization time except for the trivial bound ∞. The reason is that RLS might fail to optimize functions since it would have to change 2 or more bits for an improvement, which is not possible with RLS's mutation operator; in such a case, one says that RLS is stuck in a local optimum. Theorem 5.22 tells us that the (1+1) EA cannot get stuck and will eventually optimize every function, but the bound is exponential. Later (Theorem 5.30), we will see that the bound is tight, i. e., that on certain functions $n^{\Omega(n)}$ steps are needed.

On well-structured problems, we expect randomized search heuristics to be much faster than n^n. One of the most famous example problems is

$$\textsc{OneMax}(x_1,\ldots,x_n) = x_1 + \cdots + x_n,$$

which just returns the number of one-bits in the current search point. Of course, if the randomized search heuristic knew that it was optimizing OneMax, it would be trivial to state the optimum, namely the string of all-ones. However, randomized search heuristics do not know the fitness function since they are general black-box algorithms that only evaluate the function by querying search points. Nevertheless, our two search heuristics optimize OneMax fairly efficiently, as will be shown in the following theorem.

Theorem 5.23 *The expected optimization time of RLS and the (1+1) EA on* OneMax *is* $O(n \log n)$.

Proof. We start with a general observation. The fitness function OneMax depends only on the number of one-bits, and both search heuristics do not care about the positions of the bits in the current search point; namely, RLS selects the bit to flip randomly and (1+1) EA considers each bit independently without taking the position of the bits into account. It is therefore safe to consider only the number of one-bits in the current search point, no matter where these are. More formally, we compress the state space of the search heuristic to $\{0, \dots, n\}$, where state i means that the current search point has i one-bits.

Let us first consider RLS. If the current search points has i one-bits, there are $n - i$ out of n good outcomes of the mutation, namely those that flip a zero-bit. All other mutations lead to a search point with inferior fitness, which will be rejected. Hence, the probability of improving the number of one-bits by 1 is exactly $(n - i)/n$ in this situation. The expected time for this to happen (waiting time argument) is $n/(n - i)$. Each value of i has to be considered at most once, since the fitness of the current search point will not decrease. Pessimistically assuming that every value occurs once (i. e., assuming we start with no one-bit) gives us the expected optimization time

$$\sum_{i=0}^{n-1} \frac{n}{n-i} \;=\; n \sum_{i=1}^{n} \frac{1}{i} \leq n(\ln n + 1) \;=\; O(n \log n),$$

where we have used $\sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1$ (Lemma B.6, see the appendix).

With the (1+1) EA, the argumentation is similar. However, this search heuristic might flip several bits in a step, and it is difficult to argue about such steps. Instead, we concentrate on the steps that flip only a single bit, more precisely a zero-bit in the current search point. The probability of such a step is

$$(n-i)\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} \;\geq\; \frac{n-i}{en},$$

where we used that there are $n - i$ zero bits, each of which flips with probability $1/n$, while $n - 1$ bits do not flip with probability $1 - 1/n$ each; finally, we have used the

estimate $(1-1/n)^{n-1} \geq 1/e$ (Lemma B.3 in the appendix). Continuing as in the case of RLS, the expected optimization time of (1+1) EA is at most

$$\sum_{i=0}^{n-1} \frac{en}{n-i} \;=\; en \sum_{i=1}^{n} \frac{1}{i} \;=\; O(n \log n),$$

which completes the proof. □

Remark: The argumentation for the case of RLS is well known in the theory of randomized algorithms as the "Coupon Collector's Problem" [MR95]. The scenario is that there are n initially empty bins, and a stochastic process throws at each time step one ball into a uniformly chosen bin. It takes $\Theta(n \log n)$ expected steps until each bin contains at least one ball, which result we have proved above noting that "we throw one-bits". This analogy shows how classical algorithms theory helps to analyze the stochastic processes behind randomized search heuristics. However, we also see that we cannot apply the coupon collector's problem in its pure form to the (1+1) EA. Here we had to adapt the method towards the underlying, more complicated stochastic process.

As said above, one aim behind the analysis of simple problems like OneMax is to develop an understanding of the search heuristics and general techniques for the analysis. It turns out that the proof method from Theorem 5.23 can be generalized as follows. We partition the search space into levels of "roughly" the same fitness and consider the progress of the search heuristics through the levels.

Definition 5.24 [Fitness-based partition] Let $f\colon \{0,1\}^n \to \mathbb{R}$ be given. Then $L_0, L_1, \ldots, L_k \subseteq \{0,1\}^n$ is called a *fitness-based partition* with respect to f iff

1. $\forall i \in \{0,1,\ldots,k\}\colon L_i \neq \emptyset$
2. $\forall i \neq j \in \{0,1,\ldots,k\}\colon L_i \cap L_j = \emptyset$
3. $\bigcup_{i=0}^{k} L_i = \{0,1\}^n$
4. $\forall i < j \in \{0,1,\ldots,k\}\colon \forall x \in L_i\colon\ \forall y \in L_j\colon f(x) < f(y)$
5. $L_k = \{x \in \{0,1\}^n \mid f(x) = \max\{f(x') \mid x' \in \{0,1\}^n\}\}$

The first three conditions from Definition 5.24 just specify that $L_0, \ldots, L_k$ partitions the search space into non-empty, non-overlapping sets. The fourth condition tells us that every search point in a set with a certain index has higher fitness (i. e., f-value) than all search points in all sets with smaller index. Finally, the set with largest index contains only optimal search points, i. e., those with maximal f-value. In the proof of Theorem 5.23, we implicitly considered the partition $L_i := \{(x_1, \ldots, x_n) \mid x_1 + \cdots + x_n = i\}$ for $0 \leq i \leq n$.

A fitness-based partition will be used in the following way to derive upper bounds on the expected optimization time of RLS and the (1+1) EA. Suppose we have for each level a lower bound on the probability of leaving the level. Since our search heuristics do not accept search points of lower fitness, each level can be left at most once; in fact certain levels might be left out. Summing up the waiting times on all levels, we obtain the following general upper bound.

Theorem 5.25 (Fitness-level method) *Consider the (1+1) EA or RLS on a function $f\colon \{0,1\}^n \to \mathbb{R}$ and suppose an f-based partition $L_0, L_1, \ldots, L_k$ is given. Let for all $i \in \{0,1,\ldots,k-1\}$*

$$s_i := \min_{x \in L_i} \mathbf{P}[\textit{mutate } x \textit{ into some } y \in L_{i+1} \cup \cdots \cup L_k].$$

Then the expected optimization time on f is at most $\sum_{i=0}^{k-1} \frac{1}{s_i}$.

Proof. As mentioned above, each fitness level is left as most once. As long as the search heuristic is in level i, there is a lower bound s_i on the probability of leaving the level, and the expected time for this to happen is at most $1/s_i$ by the waiting time argument. Summing up the $1/s_i$ for all levels yields the theorem. □

The proof of Theorem 5.23 used $s_i = (n-i)/n$ and $s_i = (n-i)/(en)$ for RLS and (1+1) EA, respectively. As an additional application, we consider another well-known example function, namely

$$\textsc{BinVal}(x_1, \ldots, x_n) := \sum_{i=1}^{n} 2^{n-i} \cdot x_i,$$

which returns the integer value of the bit string with x_1 being the most significant bit.

Theorem 5.26 *The expected optimization time of the (1+1) EA and RLS on* BinVal *is $O(n^2)$.*

Proof. We use

$$L_i := \{x \mid 2^{n-1} + 2^{n-2} + \cdots + 2^{n-i} \leq \textsc{BinVal}(x) < 2^{n-1} + 2^{n-2} + \cdots + 2^{n-i} + 2^{n-i-1}\},$$

where $0 \leq i \leq n$, as fitness-based partition. By studying the intervals which make up the sets L_i, it immediately follows that $L_0, \ldots, L_n$ partitions the whole search space (first three properties from Definition 5.24) and is fitness-based (last two properties). Moreover, since $2^{k-1} \leq \sum_{j=0}^{k-1} 2^j < 2^k$ holds for all $k \geq 0$, it follows that a search point is in level i if and only if the bits with index $1, \ldots, i$ are all 1 while bit $i+1$ is 0. Note

that the remaining $n-i-1$ bits are arbitrary, hence L_i ($i < n$) contains 2^{n-i-1} search points.

In order to advance from L_i to a level of higher index, it is sufficient to flip bit $i+1$ and leave the other bits unchanged. The corresponding probability is $1/n$ for RLS and $(1/n)(1-1/n)^{n-1} \geq 1/(en)$ for the (1+1) EA. (Recall the proof of Theorem 5.23 if these estimations are unfamiliar.) Hence, defining $s_i := 1/(en)$ is in accordance with Theorem 5.25 for both heuristics. Consequently, $\sum_{i=0}^{n-1} 1/s_i = en^2 = O(n^2)$ is an upper bound on the expected optimization time. □

5.7.4 Lower Bounds

Of course, it is interesting to know whether the previously proven upper bounds are tight or whether they can be improved. Therefore, we supply lower bounds on OneMax and another example function in this subsection.

Theorem 5.27 *The expected optimization time of RLS and the (1+1) EA on* OneMax *is* $\Omega(n \log n)$.

Proof. For both search heuristics, we consider the number X of one-bits in the initial search point. Since this one is drawn uniformly, each bit is set to 1 with probability 1/2, and we obtain $\boldsymbol{E}[\,X\,] = n/2$. Since the distribution is symmetrical around its expected value, we get $\mathbf{P}[X > n/2] \leq 1/2$. Let A denote the event that the initial number of one-bits is at most $n/2$. In other words, $\mathbf{P}[A] \geq 1/2$. In the following, we consider the expected optimization time under the condition of A. Using the law of total probability (see Lemma A.18 in the appendix) we obtain $\boldsymbol{E}[\,T\,] \geq \boldsymbol{E}[\,T \mid A\,]\mathbf{P}[A] \geq \boldsymbol{E}[\,T \mid A\,]\cdot(1/2)$ for the unconditional expected optimization time $\boldsymbol{E}[\,T\,]$.

With RLS, the number of one-bits increases by at most 1 in every step and the probability of improving from i to $i+1$ one-bits is exactly $(n-i)/n$. Arguing analogously to the proof of Theorem 5.23 and using the bound $\sum_{i=1}^{k} 1/i \geq \ln k$ (Lemma B.6), we obtain

$$\boldsymbol{E}[\,T_{\text{RLS,OneMax}} \mid A\,] \;\geq\; \sum_{i=n/2}^{n-1} \frac{n}{n-i} \;=\; n \sum_{i=1}^{n/2} \frac{1}{i} \;\geq\; n \ln(n/2) \;=\; \Omega(n \log n),$$

hence also $\boldsymbol{E}[\,T_{\text{RLS,OneMax}}\,] = \Omega(n \log n)$.

With the (1+1) EA, the arguments from the preceding paragraph do not work. Instead, we take a more direct approach. The aim is to show (still assuming event A) that with probability at least c_1 for some constant $c_1 > 0$ it happens that at least one of the at

least $n/2$ zero-bits from the initial search point is never subject to mutation in a period of $c_2 n \ln n$ steps for some constant $c_2 > 0$. If we manage to prove this statement, then the law of total probability still gives us a bound of $c_1 c_2 n \ln n$ on the expected optimization time (given A), altogether $\Omega(n \log n)$ for the unconditional expected optimization time.

To prove the statement, fix an arbitrary bit i. Obviously, $\mathbf{P}[i \text{ flips}] = \frac{1}{n}$ and the probability that i does not flip equals $1 - \frac{1}{n}$ for every single step of the (1+1) EA. Moreover, due to the independence of the mutations over several steps, we obtain for every $t > 0$ that $\mathbf{P}[i \text{ does not flip in } t \text{ steps}] = (1-\frac{1}{n})^t$, and the probability that i flips at least once in t steps equals $1 - (1 - \frac{1}{n})^t$. Since the mutation operator of the (1+1) EA treats all bits independenly also while performing the mutation steps, we have that $n/2$ specific bits all flip at least once in t steps with probability $(1 - (1 - \frac{1}{n})^t)^{n/2}$ and, finally,

$$\mathbf{P}\begin{bmatrix}\exists \text{ bit out of } \frac{n}{2} \text{ specific bits} \\ \text{that never flips in } t \text{ steps}\end{bmatrix} = 1 - \left(1 - \left(1 - \frac{1}{n}\right)^t\right)^{n/2}.$$

We are left with finding an appropriate value for t. This can be found having in mind the estimations $(1 - 1/n)^n \leq 1/e \leq (1 - 1/n)^{n-1}$ and using some intuition. We verify that $t := (n-1)\ln n$ is a good choice.

$$\begin{aligned}
&\mathbf{P}[\exists \text{ bit out of } \tfrac{n}{2} \text{ specific bits that never flips in } t \text{ steps}] \\
&= 1 - \left(1 - \left(1 - \frac{1}{n}\right)^{(n-1)\ln n}\right)^{n/2} \geq 1 - \left(1 - \left(\frac{1}{e}\right)^{\ln n}\right)^{n/2} \\
&= 1 - \left(1 - \frac{1}{n}\right)^{n/2} \geq 1 - \left(\frac{1}{e}\right)^{1/2} > 0.39,
\end{aligned}$$

where we used $(1 - 1/n)^{n-1} \geq 1/e$ in the first "$\geq$" and $(1 - 1/n)^n \leq 1/e$ in the second "$\geq$" (see Lemma B.3). Since $c_1 := 0.39$ does not depend on n, $c_2 = (n-1)/2$ is bounded from below by a constant, and event A occurs with probability at least $1/2$, the desired statement has been shown and the proof is complete. □

Taking the previous proof and the upper bounds from Theorem 5.23 together, we obtain the following asymptotically tight bound.

Corollary 5.28 *The expected optimization time of RLS and the (1+1) EA on* OneMax *equals* $\Theta(n \log n)$.

One might wonder whether there are functions on which the search heuristics are faster. Reconsidering the proof of Theorem 5.27, we actually have proven a much more general result, showing that the lower bound $\Omega(n \log n)$ cannot be beaten unless the function is very simple.

Theorem 5.29 *Let $f\colon \{0,1\}^n \to \mathbb{R}$ be a function with a unique optimum. Then the expected optimization time of RLS and the (1+1) EA on f is $\Omega(n \log n)$.*

Proof. Let x^* be the unique search point that optimizes f. In the initial search point, every bit is set to the corresponding value in x^* with probability exactly $1/2$, hence the event A that x differs from x^* in at least $n/2$ bits occurs with probability at least $1/2$. Consequently, with the (1+1) EA we proceed as in the proof of Theorem 5.27 by considering "wrongly" intialized bits instead of zero-bits.

With RLS, the proof may be done as an exercise. □

Finally, we investigate the general upper bound from Theorem 5.22. It turns out to be tight by virtue of the following function.

Define

$$\textsc{Trap}(x_1, \ldots, x_n) := \begin{cases} x_1 + \cdots + x_n & \text{if } x_1 + \cdots + x_n \geq 1, \\ n+1 & \text{otherwise.} \end{cases}$$

Except for the all-zeros string, the TRAP function equals ONEMAX. However, the optimal search point is the all-zeros string, not the all-ones string. Typically, a search heuristic will follow the ONEMAX-like structure in order to arrive at the all-ones string. There it will be trapped in a sense that all bits have to be changed at once in order to reach the optimum.

Theorem 5.30 *The optimization time of the (1+1) EA on* TRAP *is at least $n^{n/2}$ with probability at least $1/3 - 2^{-\Omega(n)}$. For RLS, it is infinite with probability at least $1/3$.*

Proof. The idea is to consider a so-called *typical run* of the underlying heuristic. Let A_1 be the event that the initial search point has at least $n/2$ zero-bits. Let A_2 be the event that the search heuristic optimizes ONEMAX within at most $c2^n n \log n$ steps for some constant c to be chosen later. Let A_3 be the event that the search heuristic does not flip $n/2$ or more bits in a single step of the first $c2^n n \log n$ steps. If $A^* := A_1 \cap A_2 \cap A_3$, i. e., all three events together, occur, the run is called typical. We will show later that $\mathbf{P}[A^*] \geq 1/3$.

Let us consider a typical run. Hence, after at most $c2^n n \log n$ steps the all-ones string is reached, which is the second-best point. To create the optimum, all bits have to flip at once, which has probability $(1/n)^n$ for the (1+1) EA and is impossible for RLS. Using the law of total probability to take into account $\mathbf{P}[A^*] \geq 1/3$ (see the proof of Theorem 5.27 for the first application of this technique), we already obtain the lower bound for RLS. With regard to the (1+1) EA, the probability of reaching the optimum is $(1/n)^n$ in every step t where the all-ones string is the current search point.

Therefore, the probability of reaching the optimum within $T := n^{n/2}$ steps is at most $\sum_{t=1}^{n^{n/2}} (1/n)^n = n^{-n/2}$ by the union bound (inequality (A.5)). The probability that this happens or that the run is not typical is altogether at most $n^{-n/2} + 2^{-\Omega(n)} = 2^{-\Omega(n)}$, which proves the theorem for the (1+1) EA assuming $\mathbf{P}[A^*] \geq 1/3$.

We are left with the claim on $\mathbf{P}[A^*]$. We have already argued in the proof of Theorem 5.27 that $\mathbf{P}[A_1] \geq 1/2$. With regard to A_2, recall (Theorem 5.23 that the expected optimization time of the (1+1) EA on OneMax is at most $cn \log n$ for some constant $c > 0$, hence it is at most $c2^n n \log n$ with probability at least $1 - 2^{-n}$ using Markov's inequality (inequality (A.10) in the appendix).

With regard to A_3, observe that the probability of flipping at least $n/2$ bits in a single step is at most

$$\binom{n}{n/2} \cdot \left(\frac{1}{n}\right)^{n/2} \leq \frac{n^{n/2}}{(n/2)!} \left(\frac{1}{n}\right)^{n/2} = \frac{1}{(n/2)!} \leq \left(\frac{3e}{n}\right)^{n/2},$$

where we used the inequalities $\binom{n}{k} \leq n^k/k!$ and $k! \geq (k/e)^k$ (see Lemma B.4 in the appendix). Using a union bound similarly as in the preceding paragraph, the probability that at least $n/2$ bits flip in at least one out of $c2^n n \log n$ steps is at most $(c2^n n \log n)(3e/n)^{n/2} = 2^{-\Omega(n)}$, which proves $\mathbf{P}[A_3] = 1 - 2^{-\Omega(n)}$. We conclude the proof by noting that $\mathbf{P}[A^*] = 1 - \mathbf{P}[\overline{A^*}] = 1 - \mathbf{P}[\overline{A_1} \cup \overline{A_2} \cup \overline{A_3}] \geq 1 - (1/2 + 2^{-\Omega(n)} + 2^{-\Omega(n)}) \geq 1/3$, where we applied De Morgan's law to conclude $\overline{A_1 \cap A_2 \cap A_3} = \overline{A_1} \cup \overline{A_2} \cup \overline{A_3}$ and finally used a union bound (inequality (A.5)). □

We finally remark that the upper bound obtained for BinVal from Theorem 5.26 is not tight, and indeed the lower bound from Theorem 5.29 is tight again, which means $\Theta(n \log n)$ for the expected optimization time of RLS and (1+1) EA on both OneMax and BinVal. For further reading and more example functions, the reader is referred to [DJW02].

5.8 Analysis of Randomized Search Heuristics for Minimum Spanning Trees

This section will deal with the analysis of randomized search heuristics on problems from combinatorial optimization as they are known from textbooks on algorithms. Our analyses reuse many of the techniques developed in the previous section and introduce new, more advanced ones. In order to communicate the main proof ideas and not go get bogged down in details, we omit in the following some tedious technicalities of the analysis. For full proofs, the interested reader is referred to [NW10].

Of course, when applying general randomized search heuristics to problems like minimum spanning trees, we cannot expect the search heuristic to outperform the best problem-specific algorithm, for example Kruskal's algorithm. We can, however, hope for that the heuristic solves the problem or at least important subclasses thereof in expected polynomial time. If that is the case, we have arrived at a theoretical explanation for the success of randomized search heuristis on practically relevant problems.

Many combinatorial optimization problems involve binary decisions – which edges to select for a spanning tree, which object to include in a knapsack of bounded capacity etc. It is natural to encode such problems as optimization problems over the search space $\{0,1\}^*$, and we can apply RLS and (1+1) EA as defined above to them.

In this section, we study the *minimum spanning tree problem*. Given a graph $G = (V, E)$ with a cost function $w\colon E \to \mathbb{N}$, the aim is to find a selection $E' \subseteq E$ of edges that form a spanning tree (a connected, cycle-free subgraph) such that the total cost of the edges chosen is smallest possible. The graph G is assumed to be connected, since otherwise there is no solution. We encode the problem by working with bit strings of length $m := |E|$, where every edge is represented by a bit. More precisely, we enumerate the edges as $e_1, \ldots, e_m$, and having $x_i = 1$, $1 \leq i \leq m$, in the current bit string means that e_i is chosen. Moreover, we abbreviate $w_i := w(e_i)$. Finally, we also need the number $n := |V|$ but recall that the main characteristic of the problem is now the number of edges (and bits) m.

Many search points $x \in \{0,1\}^m$ (in fact an overwhelming fraction) are invalid in the sense that they do not encode spanning trees. Hence, using $f(x) = \sum_{i=1}^m w_i x_i$ as fitness function to be minimized (!) by the (1+1) EA or RLS is not sufficient to solve the MST problem since the search heuristic would just try to find the empty edge set. Instead, we have to build in a minimum of problem-specific knowledge and to penalize unconnected subgraphs as well as connected graphs that contain cycles. The primary aim is to arrive at a selection that is connected, the secondary aim is to remove cycles. Afterwards, the search for a spanning tree of minimum cost can be started. This leads to the following definition of the fitness function for the problem:

$$f(x) := \begin{cases} w_1 x_1 + \cdots + w_m x_m & \text{if } x \text{ encodes a tree} \\ M^2 \cdot (c(x) - 1) + M \cdot ((\sum_{i=1}^m x_i) - (n-1)) & \text{otherwise,} \end{cases}$$

where $c(x)$ is the number of connected components (CCs) in the graph encoded by x, and M is a sufficiently huge number. We work with $M := n \cdot (w_1 + \cdots + w_m)$, which is sufficiently large. The term involving M is called a *penalty term* since it pushes the search heuristics towards spanning trees.

Let us recall that the search heuristic minimizes f (equivalent to maximization of $-f$). Hence, any unconnected graph has worse f-value than any connected graph with cycles. The latter ones are worse than any spanning tree since those include exactly $n-1$ edges. This hierarchy will be used in the following, when we analyze the progress of the search heuristic towards minimum spanning trees. The aim is to prove the following result.

Theorem 5.31 *Let an arbitrary instance to the MST problem, encoded as fitness function f in the way described above, be given. Then the expected time until the (1+1) EA and RLS$^{\leq 2}$ construct a minimum spanning tree is bounded by* $O(m^2(\log n + \log w_{\max}))$ $= O(n^4(\log n + \log w_{\max}))$ *where* $w_{\max} := \max\{w_1, \ldots, w_m\}$.

RLS$^{\leq 2}$ is the same as Algorithm 5.20 except for the mutation in Step 3, which now reads as

> *With probability* $1/2$*, choose one bit in* y *and flip it; otherwise choose two bits in* y *uniformly and flip them.*

The reason is that the standard RLS algorithm could get stuck at a suboptimal search point.

5.8.1 Getting from Arbitrary Graphs to Spanning Trees

The proof of Theorem 5.31 is divided into two major parts. First, we prove that the search heuristic gets from an arbitrary initial search point to a search point describing a spanning tree in an expected time that is $O(m \log n)$, i. e., much less than the bound from the theorem and thus asymptotically neglegible. The interesting part of the analysis is described in the next subsection, where it is shown how arbitrary spanning trees are improved to minimum spanning trees.

Actually, we split the analysis further and focus first on connected graphs that are allowed to contain cycles.

Lemma 5.32 *The expected time until the (1+1) EA and RLS$^{\leq 2}$ construct a connected graph is* $O(m \log n)$.

Proof. By the structure of the fitness function, the number of connected components (CCs) in the accepted search points cannot increase. For each search point describing a graph with k CCs, there are at least $k-1$ edges whose inclusion decreases the number of CCs by 1. Otherwise, the input graph G would not be connected. The probability of decreasing the number of CCs in a step of RLS$^{\leq 2}$ is at least $(1/2)(k-1)/m$ since with probability $1/2$ only one bit is flipped and there are at least $k-1$ out of m good edges. With the (1+1) EA, the probability is at least $((k-1)/m)\cdot(1-1/m)^{m-1} \geq (k-1)/(em)$. Using the argumentation for the analysis of OneMax, the expected time until the search point describes a connected graph is at most

$$em \sum_{k=2}^{n} \frac{1}{k-1} = O(m \log n)$$

since there are at most n connected components. □

Lemma 5.33 *Starting with a search point describing a connected graph, the expected time until the (1+1) EA and RLS$^{\leq 2}$ construct a spanning tree is $O(m \log n)$.*

Proof. By the structure of the fitness function, the number of connected components (CCs) cannot increase, and if there is only one CC left, the number of edges in the accepted search points cannot increase. For each search point describing a connected graph with r edges, there are at least $r - (n-1)$ edges whose exclusion decreases the number of edges; if $r = n - 1$, we have reached a spanning tree. Using the same arguments as in the proof of Lemma 5.32, the expected time to reach a spanning tree is bounded from above by

$$em \sum_{r=n}^{m} \frac{1}{r-(n-1)} = O(m \log(m-(n-1))) = O(m \log n).$$

□

In the following, we can concentrate on spanning trees.

5.8.2 Getting from Arbitrary to Minimum Spanning Tress

Given that the current search point contains $n-1$ edges describing a spanning tree, it is no longer obvious how to make progress. Analyzing the combinatorial structure, however, it turns out that arbitrary non-optimal spanning trees can be improved by mutations flipping two bits. The combinatorial argument is formalized as follows.

Theorem 5.34 *Let T be a minimum spanning tree and S be an arbitrary spanning tree of a given weighted graph $G = (V, E, w)$. Then there exists a bijection Φ from $T \setminus S$ to $S \setminus T$ such that for every edge $e \in T \setminus S$ the following two properties hold:*

- *Adding edge e to S makes $\Phi(e)$ lie on a cycle in the resulting graph.*
- *$w(\phi(e)) \geq w(e)$.*

For a proof, see [NW10], page 55. Basically, Theorem 5.34 tells us that we can exchange $\Phi(e)$ in favor of e without destroying the spanning property and without increasing cost. Obviously, our search heuristics are able to carry out such exchanges by mutations flipping two bits. It is not yet clear, however, how this finally efficiently leads

to the minimum spanning tree. Note that if one carried out for all $e \in T \setminus S$ the exchange of $\Phi(e)$ in favor of e, we would have transformed S into T. The total decrease in f-value amounts to $w(S) - w(T)$, where we by $w(R)$ denote the total weight of a tree R. If S and T differ in k edges, we are dealing with k disjoint pairs of edges $(\Phi(e_{i_1}), e_{i_1}), \dots, (\Phi(e_{i_k}), e_{i_k})$ representing the exchange operations. Each of these improves the f-value, and it holds $\sum_{j=1}^{k}(w(\Phi(e_{i_j})) - w(e_{i_j})) = w(S) - w(T)$, i. e., the improvements of the single exchanges sum up to the total gain in f-value. Averaging yields the following lemma, where f_{opt} denotes the f-value of a minimum spanning tree.

Lemma 5.35 *Let x be a search point describing a non-minimum spanning tree. Then there exist some $k \in \{1, \dots, n-1\}$ and k different accepted two-bits flips such that the average f-decrease of these flips is at least $(f(x) - f_{\text{opt}})/k$.*

There are $\binom{m}{2}$ possibilities of choosing two bits to flip, but only $k \leq n-1$ of these are guaranteed to improve. Since our search heuristics do not accept worsenings, we can take the $\binom{m}{2} - k$ non-accepted two-bit flips into consideration. On average, given that a two-bit flip occurs, the f-decrease is at least $(f(x) - f_{\text{opt}})/\binom{m}{2}$. The search heuristics choose uniformly at random from the $\binom{m}{2} \leq \frac{m^2}{2}$ possibilities, which is why the average improvement is also the expected improvement. Finally, by arguing with a binomial distribution (Lemma B.5) with parameters m and $1/m$, we get that a two-bit flip occurs with probability at least

$$\binom{m}{2} \cdot \left(\frac{1}{m}\right)^2 \left(1 - \frac{1}{m}\right)^{m-2} \geq \frac{m(m-1)}{2m^2} \cdot e^{-1} \geq \frac{1}{4e},$$

(hence constant probability) with the (1+1) EA, and with $\text{RLS}^{\leq 2}$ the probability is 1/2. Multiplying the expected improvement due to 2-bit flips with this probability, we obtain the following statement about the expected progress.

Lemma 5.36 *Let x be the current search point of RLS or the (1+1) EA and suppose x describes a non-minimum spanning tree. Then the next search point decreases the f-value in expectation by an amount of at least $\frac{f(x)-f_{\text{opt}}}{2em^2}$.*

Note that the (bound on the) expected progress depends on the difference between the current and the optimum f-value. The farther the current search point is away from optimality, the larger the expected progress is. Often, $f(x) - f_{\text{opt}}$ is called the *distance* from the current search point x to an optimal search point. Lemma 5.36 tells us that the distance is decreased by an amount which is the old distance multiplied by an expected factor of at least $1/(2em^2)$. Putting it the other way round, the expected distance after one step is at most $1 - \frac{1}{2em^2}$ times the old distance. Such a multiplicative decrease of the distance is often observed in the analysis of randomized search heuristics, and a technique called *multiplicative drift analysis* to handle such cases has been developed. It is stated in the following theorem (without proof).

Theorem 5.37 (Multiplicative Drift Analysis) *Consider a stochastic process with discrete time $t = 0, 1, \ldots$ and suppose there is an optimal state. Denote by D_t the random distance of the process from this state at time $t \geq 0$. If the D_t are nonnegative integers and $\boldsymbol{E}[\, D_{t+1} \mid D_t = d \,] \leq cd$ for some $c < 1$ then the expected time to reach distance 0 is bounded from above by $(\ln(D_0) + 1)/(1 - c)$.*

For example, if $c = 1/2$ then the distance is halved in every step. Then the expected time to reach distance 0 is $O(\log D_0)$, i. e., logarithmic in the initial distance. This may sound natural by asking the question: How often can a number be halved until it is less than 1? The proof of the more general statement is more involved since it has to take into account that we are dealing with a random progress.

We have collected all arguments to conclude the proof of the final phase of the algorithm.

Proof of Theorem 5.31. According to Lemmas 5.32 and 5.33, a (not necessarily minimum) spanning tree is created after $O(m \log n)$ steps. After that, we apply the method of multiplicative drift analysis (Theorem 5.37) with $D_t = f(x_t) - f_{\text{opt}}$, where x_t is the current search point at time t and f_{opt} the optimum function value. According to Lemma 5.36, we can work with a factor $c = 1 - 1/(2em^2)$ by which the expected distance is decreased in every step. For the initial distance D_0 we estimate $D_0 \leq m \cdot w_{\max}$ and obtain an upper bound of order $\frac{\ln(D_0)+1}{1-c}$ from Theorem 5.37. Altogether

$$\frac{\ln(D_0) + 1}{1 - c} = O(\log(m \cdot w_{\max}) \cdot m^2) = O(m^2(\log n + \log w_{\max}))$$

since $\log m = O(\log n)$. Taking all bounds together, the expected optimization time is still $O(m^2(\log n + \log w_{\max}))$. □

If $w_{\max}$ is polynomial in m, the previous bound is polynomial in n and m and simplifies to $O(n^4 \log n)$. It turns out that this is tight. The reason for the high polynomial degree is that the search heuristics try out many two-bit flips that are not accepted.

5.8.3 Lower Bound

We only sketch the ideas for a graph where RLS$^{\leq 2}$ and the (1+1) EA take $\Omega(n^4 \log n)$ steps to find the minimum spanning tree. For full details, see [NW10], page 59.

The example graph (see Figure 5.7) consists of a clique on $n/2$ nodes and $n/4$ triangles. The purpose of the clique is to make the graph dense, i. e., to achieve $m = \Omega(n^2)$. The triangles have two light and one heavy edge, and there are three ways of choosing two

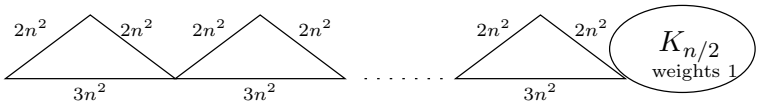


Figure 5.7: A instance to the MST problem where $\Omega(n^4 \log n)$ steps are needed

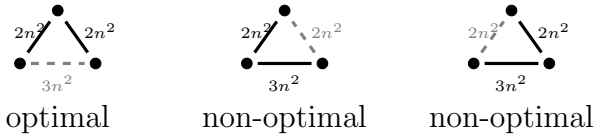


Figure 5.8: Configurations of triangles

edges of a triangle for a spanning tree. Two of these are non-optimal and only one is optimal, see Figure 5.8.

Using Chernoff bounds (inequality (A.15) in the appendix), it can be shown that $\Omega(n)$ triangles are non-optimal after initialization with high probability. This remains valid with high probability in the first spanning tree. In order to correct a non-optimal triangle, two bits (corresponding to the missing edge and the wrongly chosen edge) have to flip, which has probability $O(1/m^2) = O(1/n^4)$. Recalling that there are $\Omega(n)$ triangles to correct, an argument similar to the lower bound in Theorem 5.29 (similar to the so-called coupon collector's problem) yields the $(\log n)$-factor, resulting in an expected optimization time of $\Omega(n^4 \log n)$.

A Probability-Theoretical Background

An *event* is something that can happen or not happen. A *probability* is a real number from the interval $[0, 1]$ that is assigned to an event.

A *random variable* X is a device that can be "triggered" and every time it is triggered, it returns a numerical value out a predefined set. It cannot be determined beforehand what value will be returned. **Examples are**: A die is a random variable and is triggered by rolling it and every time it returns a value from $\{1, 2, 3, 4, 5, 6\}$. Your waiting time at the post office is a random variable. It is triggered by entering the office and returns a value from the interval $[0, \infty)$. We say that random variable X is *discrete* if the set of possible values is countable and we say it is *continuous* if the set of possible values is an interval.

Random variables define events in a canonical way. If X is a random variable and S the set of values it can take then for every $x \in S$ the following is an event "X returns value x". We write "$X = x$" for this.

Let S be a finite set $S = \{s_1, \ldots, s_n\}$. Then a *probability distribution* on S is a sequence $P = \{p_1, \ldots, p_n\}$ of real numbers such that

$$\forall i \colon p_i > 0 \qquad \text{and} \qquad \sum_{i=1}^{n} p_i = 1 \ .$$

In the countable case, we have $S = \{s_1, s_2, \ldots\}$. Then a probability distribution on S is a sequence $P = \{p_1, p_2, \ldots\}$ of real numbers such that

$$\forall i \colon p_i > 0 \qquad \text{and} \qquad \sum_{i=1}^{\infty} p_i = 1 \ .$$

Let X be a random variable which takes values in a finite (or countable) set $S = \{s_1, \ldots, s_n\}$. Then X defines a probability distribution on S in the following way:

$$p_i = \boldsymbol{P}[\, X = s_i \,] \ .$$

Example: A die takes values in $S = \{1, 2, 3, 4, 5, 6\}$. If the die is fair then all the probabilities p_i are equal and $1/6$.

Random variables $X_1, \ldots, X_k$ with values in $S_1, \ldots, S_k$ are called *independent* if for all choices $x_i \in S_i$, $i = 1, \ldots, k$ the following holds

$$\boldsymbol{P}[\,(X_1 = x_1) \wedge \cdots \wedge (X_k = x_k)\,] = \boldsymbol{P}[\,X_1 = x_1\,] \cdot \;\cdots\; \cdot \boldsymbol{P}[\,X_k = x_k\,] \;. \tag{A.1}$$

The following rules hold for probabilities of events A and B

$$\begin{aligned}
\boldsymbol{P}[\,A \wedge B\,] &= \boldsymbol{P}[\,A\,] \cdot \boldsymbol{P}[\,B\,] \qquad \textbf{iff } A \text{ and } B \text{ are independent .} && \text{(A.2)}\\
\boldsymbol{P}[\,A \vee B\,] &= \boldsymbol{P}[\,A\,] + \boldsymbol{P}[\,B\,] \qquad \textbf{iff } A \text{ and } B \text{ are disjoint .} && \text{(A.3)}\\
\boldsymbol{P}[\,A \vee B\,] &= \boldsymbol{P}[\,A\,] + \boldsymbol{P}[\,B\,] - \boldsymbol{P}[\,A \cap B\,] \;. && \text{(A.4)}\\
\boldsymbol{P}[\,A \vee B\,] &\leq \boldsymbol{P}[\,A\,] + \boldsymbol{P}[\,B\,] \;. && \text{(A.5)}
\end{aligned}$$

Note that the identities in (A.2) and (A.3) do not hold in general. The inequality (A.5) is called *union bound* and holds in general.

Let X be a random variable which takes values in a finite (or countable) set $S = \{s_1, \ldots, s_n\}$. Then the *expectation* $\boldsymbol{E}[\,X\,]$ of X is the "theoretical average" of the outcomes:

$$\boldsymbol{E}[\,X\,] = \sum_{i=1}^{n} \boldsymbol{P}[\,X = s_i\,] \cdot s_i = \sum_{i=1}^{n} p_i s_i \;. \tag{A.6}$$

Example: Let X be the fair die. Then the expectation is:

$$\boldsymbol{E}[\,X\,] = \sum_{i=1}^{6} p_i s_i = \sum_{i=1}^{6} \frac{1}{6}\, i = \frac{1}{6} \sum_{i=1}^{6} i = \frac{1}{6}\, 21 = 3{,}5 \;.$$

Note that the expectation can be a value that is not in the set S of possible values. The following fact, which is sometimes called the *probabilistic method*, will be often used:

Fact A.1 *Let X be a random variable which takes values in a finite set $S = \{s_1, \ldots, s_n\}$ and probability distribution $P = \{p_1, p_2, \ldots, p_n\}$. If $\boldsymbol{E}[\,X\,] = z$ then there is an $i \in \{1, 2, \ldots, n\}$ such that $p_i > 0$ and $s_i \geq z$, i.e, there is a value greater than or equal to z which has positive probability. An analogous statement holds if S is countably infinite.*

We shall often make use of the fact that the expectation is *linear*. This means that for random variables X and Y and $a, b \in \mathbb{R}$ the following equation holds:

$$\boldsymbol{E}[\,aX + bY\,] = a\boldsymbol{E}[\,X\,] + b\boldsymbol{E}[\,Y\,] \;. \tag{A.7}$$

For **independent** random variables X and Y, the expectation is distributive over multiplication:

$$\boldsymbol{E}[\,X \cdot Y\,] = \boldsymbol{E}[\,X\,] \cdot \boldsymbol{E}[\,Y\,] \;. \tag{A.8}$$

For a convex respectively concave function f, the following holds (*Jensen's inequality*):

$$\boldsymbol{E}[\, f(X)\,] \geq f\,(\boldsymbol{E}[\, X\,]) \quad \text{respectivley} \quad \boldsymbol{E}[\, f(X)\,] \leq f\,(\boldsymbol{E}[\, X\,])\ . \tag{A.9}$$

If X is a random variable that cannot take negative values, then deviations from its expected value can be bounded through *Markov's inequality*:

$$\forall c \geq 0\colon\ \boldsymbol{P}[\, X \geq c \cdot \boldsymbol{E}[\, X\,]\,] \leq \frac{1}{c}\ . \tag{A.10}$$

Let X be a random variable. Then its *variance* $\mathbf{Var}[\, X\,]$ is defined by

$$\mathbf{Var}[\, X\,] := \boldsymbol{E}\big[\, (X - \boldsymbol{E}[\, X\,])^2\,\big]\ , \tag{A.11}$$

which simplifies to

$$\mathbf{Var}[\, X\,] := \boldsymbol{E}\big[\, X^2\,\big] - (\boldsymbol{E}[\, X\,])^2\ . \tag{A.12}$$

The following identity is true for $a, b \in \mathbb{R}$

$$\mathbf{Var}[\, aX + b\,] = a^2\mathbf{Var}[\, X\,]\ . \tag{A.13}$$

The variance bounds deviations of a random variable from the expected value through *Chebyshev's inequality*:

$$\forall c \geq 0\colon\ \boldsymbol{P}[\,|X - \boldsymbol{E}[\, X\,]| \geq c] \leq \frac{\mathbf{Var}[\, X\,]}{c^2}\ . \tag{A.14}$$

Finally, very strong deviation bounds can be obtained in the special case that a random variable is given as the sum of n simple random variables. More precisely, we state *Chernoff's inequality*: Let $X_1, \ldots, X_n$ be independent $\{0,1\}$-random variables with $p_i := \mathbf{P}[X_i = 1]$, $1 \leq i \leq n$. For $X := \sum_{i=1}^n X_i$ and $\mu := \boldsymbol{E}[\, X\,] = \sum_{i=1}^n p_i$ it holds

$$\mathbf{P}[X < (1-\delta)\mu] < e^{-\mu\delta^2/2} \text{ for } 0 < \delta < 1, \tag{A.15}$$

$$\mathbf{P}[X > (1+\delta)\mu] < e^{-\mu\delta^2/3} \text{ for } 0 < \delta < 1. \tag{A.16}$$

Example: Assume a fair coin is tossed n times. Then X_i, $1 \leq i \leq n$, is defined to be 1 iff the outcome is heads in the i-th trial. We can apply Chernoff bounds on $X = X_1 + \cdots + X_n$ using $\mu = n/2$ and $\delta = 1/5$ and obtain that the number of heads among 1000 toin cosses is in $[400, 600]$ with probability ≥ 0.997. If the coin is tossed 10000 times, then the number of heads is in $[4000, 6000]$ with probability $\geq 0.99999999999999999999999999992$. That is, Chernoff bounds tend to become more powerful with larger expected values.

Sometimes, one considers the probability of an event A assuming that another event B happens. For example, if we roll a die and assume it shows an even number, then the probability of obtaining an outcome from $\{1, 3, 5\}$ is 0, and the probability of each of

the outcomes $2, 4, 6$ is $1/3$. Formally, we write $A \mid B$ to denote the event A under the condition that B happens. We have

$$\boldsymbol{P}[\, A \mid B \,] = \frac{\boldsymbol{P}[\, A \wedge B \,]}{\boldsymbol{P}[\, B \,]}. \tag{A.17}$$

Similarly, let $X \mid B$ denote the random variable X conditioned on B. For example, if X denotes the number shown by a die, and B denotes that event that the die shows an even number, then we obtain

$$\boldsymbol{E}[\, X \mid B \,] = \sum_{i=1}^{6} \boldsymbol{P}[\, X = i \mid B \,] \cdot i = \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 4 + \frac{1}{3} \cdot 6 = 4.$$

Finally, if $B_1, \ldots, B_n$ partitions the space of all events (i. e., exactly one of the B_i happens for sure and no two of them happen at the same time), then the *law of total probability* applies:

$$\boldsymbol{P}[\, A \,] = \sum_{i=1}^{n} \boldsymbol{P}[\, A \mid B_i \,] \cdot \boldsymbol{P}[\, B_i \,]. \tag{A.18}$$

A similar identity (again called law of total probability) holds for expected values:

$$\boldsymbol{E}[\, X \,] = \sum_{i=1}^{n} \boldsymbol{E}[\, X \mid B_i \,] \cdot \boldsymbol{P}[\, B_i \,].$$

As an example, let B_1 denote the event that the die shows an even and B_2 the event that it shows an odd number. Then B_1 and B_2 partition the space of all events and $\boldsymbol{P}[\, B_1 \,] = \boldsymbol{P}[\, B_2 \,] = 1/2$. With respect to the number X shown by the die, we have already seen that $\boldsymbol{E}[\, X \mid B_1 \,] = 4$. Similarly, one can show $\boldsymbol{E}[\, X \mid B_1 \,] = 3$. The law of total probability yields

$$\boldsymbol{E}[\, X \,] = 3 \cdot \frac{1}{2} + 4 \cdot \frac{1}{2} = 3.5,$$

which matches the unconditional expectation we have computed above.

Sometimes stochastic processes are considered that at each point of time have a certain chance of doing something "good". We are interested in the first point of time until the good event happens.

Lemma A.2 (Waiting time argument) *Suppose that a certain event occurs in each trial of a stochastic process independently with probability p. Then the expected number of trials until it occurs equals $1/p$.*

For example, the expected number of times we have to roll a die until it shows 1 for the first time equals 6.

A.1 Augmenting Success Probabilities

An experiment has a success probability p, $0 \leq p \leq 1$. How many times do I have to run it to see one success with probability $1/2$?

We bound the failure probabilities. The failure probability is $(1-p)$ for a single trial, and $(1-p)^k$ for k failures in k trials. The solution for the inequality

$$(1-p)^k \leq \frac{1}{2}$$

is

$$k \geq \frac{-\ln(2)}{\ln(1-p)} \ .$$

The right hand side can be upper bounded as follows (by first order Taylor approximation of the logarithm):

$$\frac{-\ln(2)}{\ln(1-p)} \leq \frac{\ln(2)}{p} \ .$$

Hence choosing

$$k \geq \frac{\ln(2)}{p}$$

implies

$$k \geq \frac{-\ln(2)}{\ln(1-p)} \ .$$

A.2 Size of a Random Selection

We are given a sequence $S = (a_1, a_2, \ldots, a_n)$ of n objects and a sequence $P = (p_1, p_2, \ldots, p_n)$ of real numbers $0 \leq p_i \leq 1$. The p_i do **not** have to be a probability distribution. A set $T \subseteq S$ is constructed by selecting a_i into T with probability p_i (and not selecting it with probability $(1-p_i)$). The set T is a *random selection without repetitions*. We want to know the expected size $\boldsymbol{E}[\,|T|\,]$ of T. Consider an object a_i and compute its expected contribution to the size of T. With probability p_i it contributes 1 (is added) to the size of T. With probability $1-p_i$ it contributes 0 (is not added). Using (A.6), the expected contribution is $p_i \cdot 1 + (1-p_i) \cdot 0 = p_i$. Then, by (A.7) the expected size of T is the sum of individual contributions:

$$\boldsymbol{E}[\,|T|\,] = \sum_{i=1}^{n} p_i \ . \tag{A.19}$$

B Other Results

B.1 Useful (In)equalities

Lemma B.1 *For non-negative real numbers $a_1, a_2, \ldots, a_n$ we have*

$$a_1 \cdot a_2 \cdot \cdots \cdot a_n \leq \left(\frac{a_1 + a_2 + \cdots + a_n}{n}\right)^n .$$

Lemma B.2 *For $x \in [0,1]$ the following holds:*

$$1 - \left(1 - \frac{x}{k}\right)^k \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot x \ .$$

Lemma B.3 *For all $n \in \mathbb{N}$:*

$$\left(1 + \frac{1}{n}\right)^n < e \ . \tag{B.1}$$

$$\left(1 - \frac{1}{n}\right)^n < \frac{1}{e} \ . \tag{B.2}$$

$$\left(1 - \frac{1}{n}\right)^{n-1} > \frac{1}{e} \ . \tag{B.3}$$

By $e = 2.71\ldots$ the base of the natural logarithm is meant.

Lemma B.4 *Let $n \in \mathbb{N}$ and let $k \in \{0, \ldots, n\}$. Then the* binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ *denotes the number of ways of selecting k out of distinct n objects (without*

replacement, i. e., selecting each object at most once). The following estimations apply:

$$\begin{aligned}
\binom{n}{k} &\leq \frac{n^k}{k!}. \\
\binom{n}{k} &\leq \frac{e^k n^k}{k^k}. \\
\binom{n}{k} &\geq \frac{n^k}{k^k}. \\
\binom{n}{k} &\geq \frac{1}{n+1} \cdot \left(\left(\frac{k}{n}\right)^{k/n} \cdot \left(1 - \frac{k}{n}\right)^{1-k/n} \right)^{-n}.
\end{aligned}$$

Lemma B.5 *Suppose that we carry out n trials, each of which is a success with probability p and a failure otherwise (independently of the other trials). Let the random variable X denote the total number of successes. Then X is called* binomially distributed *with parameters n and p, and it holds*

$$\mathbf{P}(X = k) \;=\; \binom{n}{k} p^k (1-p)^{n-k}$$

for $0 \leq k \leq n$.

Lemma B.6 *The n-th* Harmonic number*, where $n \in \mathbb{N}$, is defined by*

$$\sum_{i=1}^{n} \frac{1}{i}.$$

It holds

$$(\ln n) \;\leq\; \sum_{i=1}^{n} \frac{1}{i} \;\leq\; (\ln n) + 1.$$

Lemma B.7 *Consider the following recursive relation for $a, c > 0$ and $b > 1$. We assume that n is a power of b to avoid rounding.*

$$\begin{aligned}
T(1) &= 0\ , \\
T(n) &= a\,T\left(\frac{n}{b}\right) + c\,n\ .
\end{aligned}$$

Then

$$\begin{aligned}
T(n) &= c\,\frac{b}{b-a}\,n = \Theta(n) \qquad \text{if } a < b\ , \\
T(n) &= c\,n\,(\log_b(n) + 1) = \Theta(n \log(n)) \qquad \text{if } a = b\ , \\
T(n) &\leq c\,\frac{a}{a-b} n^{\log_b(a)} = \Theta(n^{\log_b(a)}) \qquad \text{if } a > b\ .
\end{aligned}$$

B.2 Estimation of Success Probabilities

Note the following:

$$\frac{1}{\left(\frac{n}{k}\right)^2} = \left(\frac{k}{n}\right)^2 \text{ and } \ln(a) = -\ln(1/a) \text{ and } \ln(a) < 0 \text{ for } a \in (0,1)$$

and for $0 < a < b$ we have $\ln(a) < \ln(b)$.

We want to solve

$$\left(1 - \left(\frac{k}{n}\right)^2\right)^t < 1/2 \tag{B.4}$$

for t.

We start by simplifying the left hand side. The simplifications make it larger.

$$\begin{aligned}
\left(1-\left(\tfrac{k}{n}\right)^2\right)^t &= \left(\left(1-\left(\tfrac{k}{n}\right)^2\right)^{\left(\frac{n}{k}\right)^2\left(\frac{k}{n}\right)^2}\right)^t && \text{(exponent 1)}\\
&= \left(\left[\left(1-\left(\tfrac{k}{n}\right)^2\right)^{\left(\frac{n}{k}\right)^2}\right]^{\left(\frac{k}{n}\right)^2}\right)^t && \text{(setting parentheses)}\\
&< \left(\left[\tfrac{1}{e}\right]^{\left(\frac{k}{n}\right)^2}\right)^t && \text{(Formula (B.2) with } x = (n/k)^2)\\
&= \left(\tfrac{1}{e}\right)^{t\left(\frac{k}{n}\right)^2} && \text{(rearranging)}
\end{aligned}$$

Now if

$$\left(\frac{1}{e}\right)^{t\left(\frac{k}{n}\right)^2} < \frac{1}{2} \tag{B.5}$$

then Inequality (B.4) also holds. Let us solve (B.5) for t, by the following equivalences:

$$\begin{aligned}
\left(\frac{1}{e}\right)^{t\left(\frac{k}{n}\right)^2} &< \frac{1}{2} \\
\ln\left(\left(\frac{1}{e}\right)^{t\left(\frac{k}{n}\right)^2}\right) &< \ln\left(\frac{1}{2}\right) \\
t\left(\frac{k}{n}\right)^2 \ln\left(\frac{1}{e}\right) &< \ln\left(\frac{1}{2}\right) \\
t\left(\frac{k}{n}\right)^2 \ln(e) &> \ln(2) \\
t &> \ln(2)\left(\frac{n}{k}\right)^2 .
\end{aligned}$$

Hence, for constant k, Inequality (B.4) is satisfied for $t > \ln(2)\left(\frac{n}{k}\right)^2 = O(n^2)$.

B.3 Infinite Sums

When computing expected running times, often infinite sums appear.

Theorem B.8 *Let $0 \leq q < 1$. Then*

1. $\sum_{k=0}^{\infty} q^k = \frac{1}{1-q}$.
2. $\sum_{k=1}^{\infty} q^k = \frac{q}{1-q}$.
3. $\sum_{k=1}^{\infty} k \cdot q^k = \frac{q}{(1-q)^2}$.

C Pseudo-Random Number Generators

The following generator was proposed by G. J. Mitchell and D. P. Moore in 1958. This generator is very well suited for pratical purposes. It is sometimes called the *ring-55* generator because it uses a circular list with 55 entries $x_0, x_1, \ldots, x_{54}$. These entries are initialized with arbitrary integers, not all of which are even. There is also a (large) number m. Then the pseudo-random sequence $x_{55}, x_{56}, x_{57}, \ldots$ is generated according to the following rule:

$$x_{n+1} = (x_{n-24} + x_{n-55}) \bmod m \ .$$

Only the last 55 numbers have to memorized thus the aforementioned list suffices.

In practice, a simpler generator is used to initialize the first 55 values. Then the generator is run for a couple of hundred rounds to deminish the effects of the initialization.

The generator also contains methods to generate random permutations of the numbers $0, 1, \ldots, (n-1)$. An array `A` of is initialized by `A[i] = i`. The random permutation is constructed from left to right. Then a position i, $0 \leq i \leq n-1$, is selected at random under uniform distribution. The value `A[i]` is exchanged with `A[0]`. The value at position 0 is the first number of the random permutation. Then a position i, $1 \leq i \leq n-1$, is selected at random, and $A[i]$ and $A[1]$ are interchanged, etc. Once the process is complete, the array is duplicated and returned. The array does not have to be re-initialized before the next permutation is computed.

Below you find the listing of a JAVA code for this generator.

```
package randomgen;

public class RandRing55
{
  private static final int modulus = 1000000000;
  private static final double modulusDouble = (double)modulus;
```

```
  private static final int seed    = 161803398;
  private int[] ring;
  private int index1,index2,help;

  public RandRing55(int newSeed)
  {
      ring = new int[55];
      int zj,zk;
      int index;

      zj = Math.abs(seed-Math.abs(newSeed));
      zj %= modulus;

     // initialize the ring
      ring[54] = zj;
      zk=1;

      for (int i = 1; i < 55; i++)
      {
        index = ((21*i) % 55)-1;
        ring[index] = zk;
        zk = zj-zk;
        if(zk < 0)
        {
          zk += modulus;
        }
        zj = ring[index];
      }

 // warm-up (randomize a little)
    for (int k = 0; k < 4; k++)
      {
      for (int i = 0; i < 55; i++)
      {
        ring[i] -= ring[(i+30)% 55];
        if(ring[i] < 0)
        {
          ring[i] += modulus;
        }
      }
    }

    index1 = 0;
    index2 = 31;
```

```
  }

 /** Returns an integer between  0 and modulus - 1 (including both)*/

  private int getRawRandom(){

    if(++index1 == 55) index1 = 0;
    if(++index2 == 55) index2 = 0;
    help = ring[index1] - ring[index2];
    if(help < 0)
    {
     help += modulus;
    }
    ring[index1] = help;
    return(help);
  }

/** Returns a real from [0,1) */

  public double getRandom(){
    return((double)getRawRandom()/modulusDouble);
  }

  /** Returns an integer between  low and high (including both) */

  public int getRandomIntIn(int low,int high){
    int result = 0;
    if(low > high)
    {
     System.out.println("Error in RandRing.getRandomIntIn: low > high");
    }
    if(low == high)
    {
      result = low;
    }
    else
    {
     int span = high-low+1;
     result = low+(int)(((double)getRawRandom()/modulusDouble)*span);
     if(result > high)
     {
       result = high;
     }
    }
    return(result);
```

```
  }

  public void initPermutaion(int n){
    if (n > 0){
     N = n;
     A = new int[N];
     for (int i = 0; i < N; i++)
     {
       A[i] = i;
     }
    }
  }

  public int[] randomPermuation(){
    int[] B = null;
    if(A != null){
     B = new int[N];
     int j,h;
     for (int i = 0; i < N-1; i++){
       j = getRandomIntIn(i,N-1);
       h    = A[i];
       A[i] = A[j];
       A[j] = h;
       B[i] = A[i];
     }
    }
    return(B);
  }
}
```

Sometimes a *random permutation* is needed. This is the case for the randomized algorithm for convex hulls. Let $a_0, \ldots, a_{n-1}$ be the numbers we want to permute randomly. We assume that they are stored in an array A, i.e., $A[i] = a_i$. We use a random number generator $rand(0, n-1)$. For each i we call $rand(i, n-1)$ and swap the content of array cell i and rand$(i, n-1)$.

RandomPermutation
for $i = 0, \ldots, (n-1)$ **do**
  swap $(A[i] \leftrightarrow A[\text{rand}(i, n-1)])$
end for

Index

Bibliography

[DJW02] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276:51–81, 2002.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman, 1979.

[HMU01] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2001.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MU05] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[NW10] F. Neumann and C. Witt. *Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity*. Springer, 2010. online version available on `www.bioinspiredcomputation.com`.

[Weg05] Ingo Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005.