1 Introduction

Computer science as a whole contains many fields. They can be very broadly divided into practical and theoretical areas. A member of the former area, for example, is the field of software engineering, of designing and implementing specific software and tools that are of direct use for people or companies. Another example of practical computer science is the field of computer graphics and multimedia, where one goal might be the modeling of three dimensional objects in computers or the automatic recognition of people's faces by computer programs. In the area of theoretical computer science most research is not directly useful for everyday life; however, it forms a basis for most other fields of computer science. Developing computer software would be less advanced if it were not for the research in the theory of programming languages. Similarly the complexity theory is a broad field, that allows people to know in advance, for example, whether it makes sense to tackle a certain problem or whether the problem simply cannot ever be solved by any computers, no matter how advanced they may become. That is the field, where this thesis is located.

The general idea behind complexity theory is to analyze different problems or tasks a computer might encounter, and compare them to one another. For this, many so called complexity classes have been introduced, which incorporate problems that are similar to each other in the way computers deal with them; that is, they have a resembling running time or they utilize similar amounts of memory space. Two very famous complexity classes are **P** and **NP**. The former contains, colloquially spoken, all problems that are *easy to solve*, whereas the latter contains all problems where it is *easy to verify*, whether a given solution is correct. A formal definition of these and other complexity classes will be given later in this thesis. Trivially, all problems in **P** are also in **NP**. The question, whether also all problems in **NP** are in **P** (thus making the two classes equal), is probably the best examined yet still unsolved problem in computer science. Though most people believe the two classes to be different, nobody has yet been able to prove or disprove their equality.

Of course, complexity theory is a very broad area in itself, since there is a vast number of different problems to consider. To name only a few, there are, for example, cryptographic problems and their protocols, optimization problems, interactive proof systems, probabilistic algorithms, and so forth. It is way beyond the scope of this thesis to discuss all of them. Though, another major issue in complexity theory, which we will take a closer look at, are Boolean formulae. A Boolean formula is a mapping from a list of input variables to one output variable, with the restriction that only values from $\{0, 1\}$ are allowed. Instead of 0 one often uses *false* and 1 is often identified with *true*. So, a very obvious question is to decide, given a certain input to the variables, whether a

Boolean formula evaluates to true or false. Another, related question is, whether there exists any input to the variables such that a given formula evaluates to true. The latter one is well-known under the name *satisfiability problem* for Boolean formulae. These formulae are the most natural kind of problems in complexity theory when dealing with the polynomial hierarchy, which is a hierarchy of complexity classes based upon the classes **NP** and **P**, because they provide – in different variations – complete problems for most classes of the polynomial hierarchy. For instance, Stephan A. Cook showed already in 1971, that the above mentioned satisfiability problem (known as SAT) is, in its general version, one of the most difficult problems in the class **NP** [Coo71]. Those most difficult problems will be called **NP**-*complete*. Naturally, the complexity of such problems depends to a large extend on the type of Boolean formula that is used. Even though SAT without any restrictions is **NP**-complete, the problem becomes trivial, when we only allow the formulae to be a conjunction of positive literals. Then, any such formula is always satisfiable. The questions that arise are: For what type of formulae is the problem still difficult? What changes it to become easy? And where exactly is that borderline? Another interesting question that comes to mind is, whether this borderline is sharp, or what kind of complexity classes between the "difficult" (**NP**-complete) and "easy" (in **P**) ones can be assumed by Boolean formulae. This question is especially reasonable, because Richard E. Ladner proved that unless $\mathbf{P} = \mathbf{NP}$, there are infinitely many complexity classes between **P** and **NP** [Lad75]. However, an intricacy is, that there are, of course, infinitely many different kinds of Boolean formulae; and it is not clear, how to look at all of these at the same time.

A good way to get to work with this infinite number of Boolean formulae is with the help of *constraints*. Basically, a constraint is simply a relation; and a constraint formula is the conjunction of such constraints. A set of constraints will be called *constraint language*. The disadvantage of using constraints is, that it is not possible to express every Boolean formula with the help of constraints. However, this drawback is compensated on the one hand by the fact that constraints still cover an infinite number of Boolean formulae, and most of the "important" problems, that have already been analyzed through other means, can be modeled by a constraint formula (e. g., 3-SAT, 2-SAT, Horn-formulae, etc.). On the other hand a great advantage of constraint formulae in comparison to "usual" Boolean formulae (of course, every constraint formula is also a Boolean formula, just a restricted version) is, that it is possible to analyze the complexity of constraint formulae in a very succinct way. As we will see, there are several properties of constraints (or constraint languages) and the complexity of all problems, that we will examine, depends solely on these properties. This allows us to state complexity results for an infinite number of Boolean formulae. This method has first been used by Thomas Schaefer in 1978. He classified the *constraint satisfaction problem* (CSP), which is the constraint version of SAT. Surprisingly, he found out, that the problem is in **P**, if the constraint language has some certain well-defined properties, and in all other cases, the CSP is already **NP**-complete [Sch78]. This theorem of his will be used as a starting point for this thesis, which is organized as follows:

After we have introduced all necessary notations and definitions in Chapter 2, we will

refine Schaefer's Theorem in Chapter 3 in such a way that we will take a closer look at the Boolean constraint satisfaction problem. Schaefer obtained a dichotomatic complexity result by separating the problem into **P** and **NP**-complete cases. We will examine the tractable cases in more detail and show that these can be divided further into four cases (or five, if we count the trivial ones as a case of their own).

In Chapter 4 we generalize the constraint satisfaction problem by introducing quantifiers for the variables. Then the question is not anymore, whether there is a satisfying assignment for a given formula, but whether the variables are quantified in such a way, that the formula is true. In addition to the decision problem, we also consider its corresponding counting version: There not all variables need to be quantified and we are interested in the number of solutions for the not quantified variables.

Finally, we consider graph related problems in Chapter 5. We start with the constraint analogon to graph isomorphism – the equivalence problem – and its one-sided version – implication. Then, we move on to the isomorphic implication problem, which is the constraint counterpart to subgraph isomorphism. Incidentally, the isomorphic implication problem discloses an interesting and new approach, which could lead to a proof that graph isomorphism is in **P**, which it is not known to be.

The results of Chapter 3 are based on a joint work with Eric Allender, Neil Immerman, Henning Schnoor, and Heribert Vollmer [ABI$^+$05]. Chapter 4 relies on the previous publications [BCC$^+$04] and [BBC$^+$05], which are joint works with Elmar Böhler, Philippe Chapdelaine, Nadia Creignou, Steffen Reith, Henning Schnoor, and Heribert Vollmer. Parts of Chapter 5 have been obtained together with Edith Hemaspaandra and have been published in [BH05].

2 Preliminaries

This chapter contains definitions and theorems, which will be used extensively throughout the paper. We will start with some basic mathematical definitions including Boolean formulae. Then follows an introduction to complexity classes and reductions. Thereafter we will define constraints and finally we will introduce some closure properties, which will enable us to give short and elegant proofs as we will see in Chapters 3 and 4.

2.1 Basics

A basic mathematical object is the *set*, which is a collection of elements. Each such element in a set is called a *member* of that particular set. There are two different ways to describe a set. One is to list all of its elements (e. g., $A = \{a_1, a_2, a_3, a_4\}$) and the other is to describe, what properties all elements in the set possess (e. g., $A = \{a \mid a \text{ has property } E\}$). Instead of listing each and every element, which is not possible for infinite sets and infeasible for very large sets, one can also use ellipses $(\dots)$. If an element a is a member of a set A, we also write $a \in A$, and accordingly $a \notin A$, if a is not a member of A. Further, we will write $A \subseteq B$ if every member of A is also a member of B, and say A is a *subset* of B; we write $A \subset B$ if $A \subseteq B$, but not $B \subseteq A$, and say A is a *proper subset* of B. If on the other hand $A \subseteq B$ and $B \subseteq A$, we say that A *equals* B, denoted by $A = B$. The set with no elements is called the *empty set* and will be denoted by $\emptyset$. The number of the elements in a set is called its *cardinality* and is denoted by $|A|$ for a set A.

Example 2.1.1 *The sets A, B, and C will be defined by listing all elements, whereas D and E use the description technique: $A = \{2, 3, 5\}$, $B = \{1, 2, \dots\}$, $C = \{1, \dots, 131072\}$, $D = \{a \mid a \text{ is a natural number}\}$, $E = \{a \mid a \text{ is prime}\}$. Obviously, the following holds: $1 \notin A$, $2 \in A$, $A \subset E \subseteq B = D$, $|A| = 3$, and $|\emptyset| = 0$. The set D of natural numbers will be used quite often and is usually denoted by $\mathbb{N}$.*

There are three basic operations on sets: the *union* ($\cup$), *intersection* ($\cap$), and the *difference* ($\setminus$). The union of A and B is defined to be the set of elements, which are in A or in B or in both. An element is in the intersection of A and B, if and only if it is an element of A and an element of B. And finally, $a \in A \setminus B$ if and only if $a \in A$ and $a \notin B$.

An object similar to the set is the *tuple*. It is also a collection of elements; however, contrary to a set, a tuple has to be finite and all elements are ordered. Tuples are written in parentheses (e. g., $(1, 3, 1)$). If n is the number of elements in a tuple u, we also call it an n-tuple or an n-ary tuple. For $n = 2$ and $n = 3$ we use the short forms of *pair* and *triple*. Similarly to sets, the number of elements in a tuple u is also denoted by $|u|$.

Instead of tuple, we sometimes also use the more general expression vector, which, for the scope of this thesis, should always denote a tuple.

There exists a direct connection between sets and tuples, namely the *cross product* (also called *Cartesian product*) of sets, which yields a set of tuples. For $A_1, \ldots, A_n$ sets, define $A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_1 \in A_1, \ldots, a_n \in A_n\}$. As a special case, A^n for A a set and $n \in \mathbb{N}$ is, analogously to the power of numbers, defined as $A \times \cdots \times A$. Hereby $A^0 = \{()\}$ and A^1 formally is $\{(a) \mid a \in A\}$; however, to simplify matters we will define $A^1 = A$.

An n-ary Boolean *relation* R is a subset of $\{0,1\}^n$. An n-ary Boolean *function* (sometimes also called *mapping*) f is a subset of $\{0,1\}^{n+1}$ with the additional property that if $(a_1, \ldots, a_n, b) \in f$ and $(a_1, \ldots, a_n, b') \in f$, then $b = b'$, where $a_1, \ldots, a_n, b, b' \in \{0,1\}$. Instead of $(a_1, \ldots, a_n, b) \in f$ we also write $f(a_1, \ldots, a_n) = b$ and say that f maps $(a_1, \ldots, a_n)$ to b. Accordingly, instead of $f \subseteq \{0,1\}^{n+1}$ we mainly write $f\colon \{0,1\}^n \to \{0,1\}$, where $\{0,1\}^n$ is the *input* and $\{0,1\}$ is the *output*. As a generalization of the Boolean function we allow the input and output to be arbitrary sets A and B instead of $\{0,1\}^n$; thus, yielding a function $f\colon A \to B$. A function f is called *injective*, if it has the property that $f(a) = f(b)$ implies $a = b$ for any $a, b \in A$. An injective function $f\colon A \to B$ is called *bijective*, if for all $b \in B$ there is an $a \in A$ such that $f(a) = b$, that is, every element from B is mapped to. A *permutation* π is a special kind of function, that maps from one set A to itself, such that $A = \{\pi(a) \mid a \in A\}$.

In the following we will use the standard correspondence between Boolean relations and Boolean formulae; that is, an n-ary Boolean relation R corresponds to an n-ary Boolean formula f in such a way, that a vector v is in R if and only if $f(v) = 1$.

A finite non-empty set Σ is called *alphabet*. A sequence $w = a_1 \ldots a_n$ of n elements of Σ for $n \geq 0$ is called a *word* (or sometimes also *string*) over the alphabet Σ and $|w| = n$ is called its *length*. In the special case of $n = 0$ we write $w = \varepsilon$ and call ε the *empty word*. The set of all words over Σ is denoted by Σ^*. Let $A \subseteq \Sigma^*$; then, A is called a *language* over Σ. If $A \subseteq \Sigma^*$ is a language, we define the *complement* of A as $\overline{A} = \Sigma^* \setminus A$.

Example 2.1.2 *Let $\Sigma = \{0,1\}$ be an alphabet. Then the set of all words over Σ is the set $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Possible languages over Σ would be $A = \emptyset$, $B = \Sigma^*$, $C = \{1, 10\}$, and $D = \{\varepsilon, 0, 00, 000, \ldots\}$. Obviously $\overline{A} = B$ and $\overline{B} = A$.*

Finally, we will define some notions of the propositional logic, in particular formulae and their assignments. Let $X = \{x_1, \ldots, x_n\}$ be a set of variables. Then any variable $x_i \in X$ is a *propositional formula*. If φ and ψ are propositional formulae, then also $\neg\varphi$, $\varphi \wedge \psi$, and $\varphi \vee \psi$ are propositional formulae. Instead of $\neg\varphi$ we sometimes also write $\overline{\varphi}$. If φ is a propositional formula and $x_1, \ldots, x_n$ are all variables occurring in φ, we say that φ is a formula *over* $X = \{x_1, \ldots, x_n\}$. For the following, let φ be a propositional formula over variables $X = \{x_1, \ldots, x_n\}$. A function I from X to $\{0,1\}$ is called a *truth assignment* (or *assignment*) of φ. We now define inductively what it means that a truth assignment I *satisfies* a given formula φ. If $\varphi = x_i$ for a variable x_i, then I satisfies φ if and only if $I(x_i) = 1$. If $\varphi = \psi_1 \wedge \psi_2$ ($\varphi = \psi_1 \vee \psi_2$), then I satisfies φ if and only if I satisfies ψ_1 and ψ_2 (ψ_1 or ψ_2, resp.). This way, a propositional formula can also be seen

formula	name	description	abbreviation for
φ	*id*	identity	n/a
$\neg\varphi$, $\overline{\varphi}$	*not*	negation	n/a
$\varphi \wedge \psi$	*and*	conjunction	n/a
$\varphi \vee \psi$	*or*	disjunction	n/a
$\varphi \rightarrow \psi$	*impl*	implication	$\neg\varphi \vee \psi$
$\varphi \leftrightarrow \psi$	*equiv*	equivalence	$(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
$\varphi \oplus \psi$	*xor*	exclusive or	$\neg(\varphi \leftrightarrow \psi)$

Table 2.1: List of important Boolean functions

as a Boolean function and for a formula φ over variables $X = \{x_1, \ldots, x_n\}$ we thus write $\varphi(x_1, \ldots, x_n)$. Depending on the assignment, it calculates either 0 or 1. Table 2.1 gives an overview of the most important Boolean functions. When a formula is considered as a function, the truth assignments are usually written in form of a tuple instead of a function. Thus, $(\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$ is said to satisfy a formula φ, if $\varphi(\alpha_1, \ldots, \alpha_n) = 1$. We denote with $\mathrm{sat}(\varphi)$ ($\mathrm{unsat}(\varphi)$) the set of satisfying (unsatisfying, resp.) assignments of φ; that is, $\mathrm{sat}(\varphi)$ ($\mathrm{unsat}(\varphi)$) is the set $A \subseteq \{0,1\}^n$ such that for every $\alpha \in A$ we have that $\varphi(\alpha) = 1$ ($\varphi(\alpha) = 0$, resp.). Further we write $\#\mathrm{sat}(\varphi)$ for $|\mathrm{sat}(\varphi)|$ and $\#\mathrm{unsat}(\varphi)$ for $|\mathrm{unsat}(\varphi)|$ to denote the number of satisfying respectively unsatisfying assignments of φ.

There are two main normal forms of propositional formulae. One is the *conjunctive normal form* (CNF) and the other is the *disjunctive normal form* (DNF). A formula φ is said to be in CNF if it is of the form $\varphi = c_1 \wedge \cdots \wedge c_n$, where $c_i = l_{j_1} \vee \cdots \vee l_{j_{m_i}}$ for $1 \leq i \leq n$ and all l_k are variables or negated variables. The c_i's are called *clauses* and the l_k's are *literals*. A DNF is similarly a disjunction of conjunctions; that is, $\varphi = d_1 \vee \cdots \vee d_n$, where $d_i = l_{j_1} \wedge \cdots \wedge l_{j_{m_i}}$ for $1 \leq i \leq n$ and all l_k are variables or negated variables. The d_i's are then called *disjuncts* and the l_k's are again literals. It can be shown that any propositional formula can be transformed into a CNF and a DNF. If we only allow up to three literals per clause (disjunct), the resulting formula is said to be a 3-CNF (3-DNF, resp.).

An extension to this is a formula with *quantifiers*, where a quantifier is a symbol $\forall$ or $\exists$. For this we first have to introduce the notion of *free variables*. Let φ be a formula over variables X. Then the free variables of φ are all occurring variables. If φ is a (quantified) formula with free variables $X = \{x_1, \ldots, x_n\}$, then $\exists x_i \varphi$ and $\forall x_i \varphi$ with $x_i \in X$ are *quantified formulae* with free variables $X \setminus \{x_i\}$. An assignment I of the free variables satisfies an existentially quantified formula $\exists x \varphi$ if and only if there is a value $\alpha \in \{0,1\}$ such that $I \cup \{x = \alpha\}$ satisfies φ. An assignment I of the free variables satisfies a universally quantified formula $\forall x \varphi$ if and only if for all values $\alpha \in \{0,1\}$, we have that $I \cup \{x = \alpha\}$ satisfies φ. If a formula φ has no free variables, it is said to be *closed*, and it is either true or false.

2.2 Complexity

In complexity theory a main objective is to classify problems by their complexity, that is, by their difficulty. In order to do this, we first have to clarify two things: What kind of difficulty are we looking for? And how exactly do we measure it? For example, if the problem is to get from one place to another, we could define the difficulty as the time that is needed to do the journey. Another difficulty might be, how much money it takes to do the journey. Now measuring the time seems not to be a problem (you just take a stop watch and see how long the journey takes), whereas the measurement of money is not so clear anymore. If you take a short trip of a few hundred metres, you can certainly do it for free if you just walk. However, on a trip from Paris to Rome, you could also just walk, but it would take you days and you would have to sleep and eat meanwhile, which would probably cost you some money. So, the question is, what kind of money is included and what is not. Also the measurement of time is not always obvious. The fastest way from Paris to Rome is certainly to fly. However, do you include the time you need to buy a plane ticket, get to the airport, wait for the next plane, and so on? Or do you just take the flying time from one airport to the other?

2.2.1 A Basic Model

Our problems are of course somewhat more theoretical, though, the general principle remains the same. You might, for example, be given a list of numbers and want to sort them. One important complexity is, of course, the time that is needed to sort those numbers. Similar to the practical example above, this certainly depends on the help you have. If you do it by hand, it will take much longer than if you do it with the help of a computer. But, even with a computer, there are large differences of speed mainly depending on the age of the computer, but also depending on how the data is fed to the computer. We therefore would like to have a fixed computational model, which is supplied by the *Turing machine*, or short TM. Basically, a TM is an automaton with a finite number of states. It has an infinite tape divided into cells and a head, which can look at one cell at a time. Each cell contains exactly one out of finitely many symbols. At the start of each run of a TM the input data is assumed to be on the tape starting at the head's position. Depending on the symbol at the head's current location and the state of the machine, it may enter a new state, may change the symbol, and afterwards move the head one cell to the left, one cell to the right, or leave it at the current position. Although this model is very simple, it is as powerful as any programming language. A detailed definition of the Turing machine is omitted here, but a very good introduction can be found in, for example, [Sip97] and [HU79].

Although a TM could produce any kind of output, we will almost only deal with TMs for *decision problems* (with a little exception in Chapter 4). These are TMs designed for a specific language and they decide, whether a word is in that language or not; that is, they either accept or reject their input by entering a unique accepting or rejecting state. For our purposes we still need to define four variants of such a TM.

- The first one is a generalization. Instead of having only one tape, we allow the TM to have a constant number k of tapes. Accordingly, there are also k heads, one for each tape. Such a TM is then called *k-tape Turing machine* or in case k is not known or not important, we call it a *multi-tape Turing machine*. The action of the k-tape TM depends on its state and the content of the tapes at each head's position. All heads can be moved independently from each other.

- Another variant is that the TM has an additional input tape. This input tape is a read-only tape. The purpose will be seen later.

- The third variant is a *nondeterministic* TM (NTM). In contrast to a (deterministic) TM, an NTM is allowed to have more than one possible action for any given pair of state and read symbol. Nondeterminism can then be explained in two ways. The first being that whenever there is more than one possible action the NTM could do, it branches and follows all possibilities in parallel. Each sequence of possible branchings is called a *computation path* or simply path of the TM. The NTM accepts an input if and only if at least one of those paths terminates in an accepting state. We also call such a path an accepting path. The other explanation is that by some "divine intuition" the TM always chooses the right action if such exists, whenever several possibilities occur. Thus, it "guesses" a computation path and if there is at least one accepting path, the NTM will always choose an accepting path.

- Finally, we define the so-called *oracle Turing machine*. This is a general TM with an additional oracle A. Such a TM possesses a separate write-only oracle tape and three distinguished states $q_?$, q_0, and q_1. Whenever the state $q_?$ is reached, the TM either enters one of the distinguished states q_0 or q_1 depending on whether the word on the oracle tape is in A or not. Thus, it has access to the language A with almost no cost. Intuitively such an oracle can be considered as a subroutine of a programming language. Except from this peculiarity the oracle TM works as usual.

A formal definition of nondeterministic and oracle Turing machines can again be found in many text books about theoretical computer science, see, for example, [Sip97] and [HU79] for nondeterminism and [Pap94] for oracles.

2.2.2 Complexity Classes

We finally have a palpable and well-defined model for computations, which allows us to define complexity classes. The *time* that is needed to solve a problem, is defined as the steps it takes a TM to decide it. A second important complexity is the *space* a TM uses during its computation. This is defined as the number of cells on the work tape that are being visited by the TM during its computation. For space complexity we always assume a TM with additional input tape, but the visited cells of that tape are not considered. However, in complexity theory we are not interested in the exact amount of steps taken or cells visited, but just in their order of magnitude. Therefore, the *O-notation* is very