Chapter 1

Introduction: Logic, Search, and $\mathcal{NP}$-hard Problems

The aim of this introductory chapter is to give the background and motivation for the work presented in this thesis. After a brief general introduction to Artificial Intelligence and the types of problems arising in this field of computer science, we discuss and compare the concepts of systematic and local search. We then present the model of generic problem solving which provides the motivation for the major part of the topics investigated in this thesis. Next, we give a short introduction to $\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems, after which we introduce two fundamental $\mathcal{NP}$-hard problems, the satisfiability problem in propositional logic (SAT) and the constraint satisfaction problem (CSP), which form the basis for a major part of this thesis. Finally, we give some pointers to the literature related to the topics covered here and conclude with a short summary.

1.1 AI Problems, Logic, and Search

Within computer science, a relatively young and still rapidly growing academic discipline with a vast impact on a significant part of our modern societies, some of the most intriguing problems can be found in a field called Artificial Intelligence (AI), or Intellectics[1]. The general goal of AI is to realise intelligent behaviour in machines, an idea which by far predates the construction of the first universal computers. Since this basic motivation is tightly correlated with general questions regarding human intelligence, cognition, and the workings of the mind, AI problems are often, if not always, closely related to disciplines other than computer science, such as Psychology, Philosophy, and Neurobiology.

Some of the most prominent problems in AI are in the areas of language understanding, the-

[1] This name has been coined by Wolfgang Bibel, it refers to the discipline uniting traditional AI and certain areas of cognitive science [Bib80].

orem proving, game playing, pattern recognition, and musical composition. Unfortunately, most of these tasks involve computationally very hard problems and despite quite impressive successes in many areas of AI, today still most general AI problems lack completely satisfactory and feasible solutions. Like all problems in computer science, AI problems can be categorised in two major types: representational problems and computational problems. While the former problem type involves finding a suitable representation or formalisation of a problem from some domain, the latter type is about finding algorithms to solve these problems. Of course, both types of problems are closely related, since usually solving a real-world problem involves both finding a suitable representation and devising an algorithm to solve the formalised problem. Moreover, the construction and implementation of problem solving algorithms often critically depends on the problem being represented in a suitable way, just as certain representations almost naturally lead to specific algorithms.

In AI, problems are often represented in the framework of formal logics. A formal logic $\mathcal{L}$ is given by a formal language which specifies the syntax of logical expressions, and a mapping of these expressions to a system of truth values specifying the semantics of $\mathcal{L}$. Examples for formal logics which are used and studied in AI are propositional logic, first-order logic, second and higher order logics, modal logics, temporal logics, and action logics. Sentences which are true in a given logical system are called *theorems*. Theorems can be proven semantically, using formal reasoning outside the logical system. Alternatively, syntactical prove mechanisms called *calculi* can be used for establishing truth or falsehood of logical sentences. A logical calculus is a formal system consisting of a set of *axioms*, which are elementary true sentences, and a set of *rules*, which define truth preserving transformations of sentences. Calculi are the basis for the automatisation of logic. However, not for every logical system semantical truth can be characterised by a calculus, *i.e.*, for certain logical systems, there are semantically true sentences which cannot be algorithmically proven. In these logics, theoremhood is undecidable. First-order logic is of this type, while for propositional logic the validity of sentences can be decided algorithmically.

Many AI problems, particularly the ones arising in the context of formal logics, are decision problems, where solutions are characterised by a set of conditions which have to be satisfied in order to solve the problem. Classical theorem proving is usually formulated as a decision problem. Here, the solutions are the proofs of the given formula within a given proof system (or calculus). However, problems from application domains are often rather optimisation than decision problems. Optimisation problems can be seen as generalisations of decision problems, where the solutions are additionally valued by an objective function. There are two types of optimisation problems, a maximisation and a minimisation variant. For these problems, the goal is to find optimal solutions, *i.e.*, solutions with maximal resp. minimal value of the objective function. The theorem proving example from above can be formulated as an optimisation problem, if we take the proof length as the objective function and modify the problem goal to finding minimal length proofs. As we will see in the course of this work, many algorithms for decision problems can be extended to optimisation problems in a rather natural way. However, this is not always the case, since for many optimisation problems finding valid solutions is not very hard, whereas finding optimal or near-optimal solutions is computationally very expensive.

In many cases, decision and optimisation problems in AI can be characterised as search problems, *i.e.*, problems which involve some sort of searching through a number of solution candidates or partial solutions while trying to find an actual, valid (and maybe optimal) solution for the given problem instance. Many problems from game playing, theorem proving, and other areas of AI can be formulated as search problems in this sense. Note, however, that algorithms for solving search problems need not always be classical search algorithms. Actually, the search aspect does not not even have to be explicitly represented in such algorithms. Nevertheless, the abstract notion of search is fundamental for many AI problems and algorithms. As mentioned before, many AI problems are computationally very hard. Using the search metaphor, this is usually reflected by the fact that for a given problem instance, the set of solution candidates or partial solutions which defines the space in which the search takes place, is very large and no algorithmic methods for effectively finding actual solutions in these huge search spaces are known. A very typical situation is given in the case of problems where the time complexity of finding solutions grows exponentially with the problem size, while to decide whether a solution candidate is an actual solution can be done in polynomial time. It is in this situation, where algorithms based on the search metaphor are most frequently applied.

1.2 Systematic versus Local Search

Search methods can be roughly classified into two categories: systematic and local search.[2] Systematic search algorithms traverse the search space of a problem instance in a systematic manner which guarantees that eventually either a solution is found, or, if no solution exists, this fact is determined with certainty. This typical property of algorithms based on systematic search is called completeness. On the other hand, local search initialises the search process in some point of the search space and then procedes by iteratively moving from the present location to a neighbouring location, where the decision on each step is based on local knowledge only. Even if the local decisions are made in a rather systematic way, typically local search algorithms are incomplete, *i.e.*, even if the given problem instance has a solution, they are not guaranteed to find it eventually. Also, local search methods can visit the same location within the search space more than once. Actually, they are frequently prone to getting stuck in some part of the search space which they cannot escape from without special mechanisms like a complete restart of the search process or some other sort of diversification steps.

It might appear that, due to their incompleteness, local search algorithms are generally inferior to systematic methods. But as will be shown later, this is not quite true. Firstly, many problems are constructive by nature, it is known that they are solvable and what is required is actually the generation of a solution rather than just deciding whether a solutions

[2]Systematic search often exploits structural features of the problem representation and may thus also be referred to as "structural search"

does exist. This holds in particular for optimisation problems, like the Traveling Salesperson Problem (TSP) where the actual problem is to find a solution of sufficient optimality, but also for underconstrained decision problems which are quite common.

Secondly, in a typical application scenario often the time to find a solution is limited. Examples for such real-time problems can be found in virtually all application domains. Actually one might argue that almost every real-world problem involving interaction with the physical world, including humans, has real-time constraints. (Common examples are robot motion planning and decision making, most game playing situations, and speech recognition for natural language interfaces.) In these situations, systematic algorithms often have to be aborted after the given time has been exhausted, which — of course — renders them incomplete. This is particularly problematic for systematic optimisation algorithms which search through spaces of partial solutions; if those are aborted usually no solution candidate is available, while in the same situation local search algorithms typically offer the best solution found so far.[3] Ideally, algorithms for real-time problems should be able to deliver reasonably good solutions at any point during their execution. For optimisation problems this typically means that run-time and solution quality should be positively correlated, for decision problems one could imagine to guess a solution when a time-out occurs, where the accuracy of the guess should increase with the run-time of the algorithm. This so-called *any-time property* of algorithms is usually difficult to achieve, but in many situations the local search paradigm seems to be better suited for devising any-time algorithms than systematic search models.

As a matter of fact, systematic and local search algorithms are somewhat complementary in their applications. A nice example for this can be found in [KS96], where a fast local search algorithm is used for finding actual solutions for planning problems the optimality of which is proven by means of a systematic algorithm. As we will discuss later in more detail, different views of the same problem may in certain cases, particularly if reasonably good solutions are required within a short time using parallel computation and the knowledge about the problem domain is very limited, call for local search algorithms. In other cases, usually if exact or optimal solutions are required, time constraints are less important and some knowledge about the problem domain can be exploited, systematic search will be the better choice. Finally, there is some evidence that for certain problem classes, local and systematic search methods are most effective on different subclasses of instances. Unfortunately, to date the general question on when to prefer systematic and when local search methods remains mainly open.

Often, local search algorithms make heavily use of stochastic mechanisms, such as probabilistic tie-breaking heuristics. These algorithms are called *stochastic local search (*or *SLS) algorithms*, and have been used for many years now in the context of combinatorial optimisation problems. Among the most prominent algorithms of this kind we find the Lin-Kernighan algorithm [LK73] for the Traveling Salesperson Problem, as well as general methods like Tabu Search [Glo89], and Simulated Annealing [KJV83]. Lately it has be-

[3] However, it should be noted that, though quite uncommon so far, local search algorithms can be also designed in such a way that they are searching in a space of partial solutions.

come evident that stochastic local search algorithms can also be very successfully applied to the solution of $\mathcal{NP}$-complete decision problems such as the Graph Colouring Problem (GCP) [HD87, MJPL90, MJPL92] or the Satisfiability Problem in propositional logic (SAT) [SLM92, GW93a, Gu94]. The latter is particularly interesting for AI research because it is not only one of the basic problems arising in a (rather simple) context of logic-based approaches, but it can also be used in the context of a generic problem solving approach where basic problems from other domains, such as Blocks World Planning or Graph Colouring, are solved by encoding them into SAT and subsequently solving the encoded instances using stochastic local search techniques. Recent research results indicate that this approach is viable for at least some problem domains [KS96, JKS95] and consequently today there is considerable interest, not only in the AI community, in the questions connected with it and the research in this areas.

1.3 Generic Problem Solving Using Stochastic Local Search

Much of the interest in efficient algorithms for rather abstract but syntactically simple problems, like the satisfiability problem in propositional logic (SAT), is based on the notion of generic problem solving. There are two variants of this indirect approach, the weaker of which relies on the assumption that general techniques can be identified which are effective for dealing with a broad range of problem classes. If this is correct, developing and studying algorithms on syntactically simple problems is a very reasonable approach. The stronger variant assumes that it is possible to solve problems by transforming them into some domain for which an effective universal solver is available. The solutions that are thus obtained are then back-transformed into the original problem domain which yields the desired solutions for the problem at hand.

This approach of generic problem solving relies on a number of assumptions. First of all, the domain the problems are transformed into (the *intermediate domain*) has to provide sufficient expressive power, *i.e.*, a significant number of interesting problem classes have to be representable in this domain. Next, the complexity of the transformations between the domains has to be sufficiently low compared to the complexity of finding a solution, otherwise the overhead involved in this indirect approach might easily outweigh its advantages. Finally, the most crucial assumption is the existence of a sufficiently general solver in the intermediate domain.

One could argue, that these assumptions are critical in the sense that at least for some of them there is some doubt on whether they can be met in practice. There is a number of domains, which are sufficiently expressive to represent a wide variety of interesting problem classes. For some of these, like SAT and CSP, efficient transformations have been found for many problem classes [SW93] such that there is some indication that the extra costs of encoding/decoding are almost negligible when compared to the costs of finding solutions. It is, however, not clear whether there are sufficiently universal solvers for these intermediate domains and how these are affected by the choice of problem encoding. Very probably there

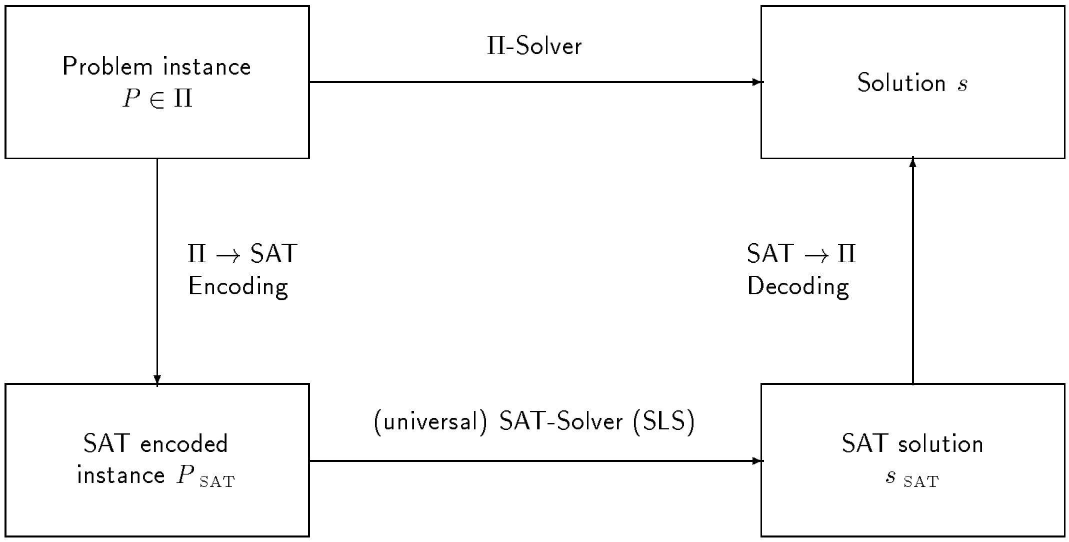

Figure 1.1: Generic problem solving by transformation.

will be a tight link between the problem encoding and a given algorithm's performance. But even assuming that one could find an intermediate-domain algorithm which achieves reasonable performance on *some* encoding for a sufficiently broad range of problems, it is still not clear whether finding suitable encodings is actually less difficult than devising specialised algorithms for the original application domains. Consequently, the crucial problem with the generic problem solving approach is to find an intermediate-domain algorithm and a number of sufficiently efficient and simple encoding strategies which together show a reasonable performance on a broad range of interesting problem domains.

On the other hand, the advantages of the generic solving approach are obvious. First of all, no specialised solver is needed for the original problem domain. Also, many domains benefit from improvements of intermediate-domain solvers. Thus, it pays off to invest significant effort in improving intermediate-domain solvers, and even special hardware support could be an effective way of speeding up these solvers [HM97]. The weak variant of the generic problem solving approach is widely accepted in AI and to date, many problem solving techniques are known which can be successfully applied to a variety of problem domains. The strong variant, however, is less commonly accepted. Nevertheless, recent results for solving SAT encoded problems using stochastic local search techniques give some indication that even the strong variant of generic problem solving might be feasible in practice.

There are several reasons for regarding SAT as a promising intermediate domain. First of all, SAT is $\mathcal{NP}$-complete which means that every $\mathcal{NP}$-complete problem can be polynomially reduced to SAT. Thus, reasonable efficient transformations between SAT and other $\mathcal{NP}$-complete decision problems exist. Since the class of $\mathcal{NP}$-complete decision problems covers a wide range of interesting application problems, SAT is expressive enough for acting as an intermediate domain. The second advantage of SAT is its conceptual simplicity. Because the syntax as well as the semantics of SAT is very simple, especially when restricted to CNF formulae, SAT offers substantial advantages for devising and evaluating algorithms.

So if SAT appears to be a reasonable choice for the intermediate domain, why should one consider SLS algorithms as SAT solvers? First, as mentioned before, SLS algorithms have been found to be competitive with or even superior to systematic search for both native SAT problems as well a broad range of SAT encoded problems from other domains. Furthermore, most popular SLS algorithms for SAT are relatively easy to implement because they are conceptually quite simple. Finally, most of these SLS algorithms can be parallelised in a rather straightforward way. On the other hand, there are some disadvantages of SLS algorithms to be mentioned. First, typical SLS algorithms are incomplete, *i.e.*, they are not guaranteed to find an existing solution. Then, these algorithms are highly non-deterministic which sometimes is seen as a problem by itself (for example, when it comes to the reliability and reproducibility of results). But a more important drawback of this property is the fact that these algorithms have been found extremely difficult to analyse and consequently to date, there is almost no theoretical understanding of their behaviour. Nevertheless, the advantages, especially the superior performance that could be achieved for different types of SAT problems, seem to outweigh these drawbacks such that SLS algorithms are considered to be among the most promising methods for solving large and hard satisfiability problems.

1.4 $\mathcal{NP}$-Complete and $\mathcal{NP}$-Hard Problems

Today, the theory of computational complexity is a well-established field with considerable impact on other areas of computer science. In the context of this work, complexity theory plays a certain role, because the primary field of application of stochastic local search algorithms is a class of computationally very hard problems, for which no efficient, *i.e.*, polynomial time, algorithms are known. Moreover, todate a majority of the experts in complexity theory believe that for principal reasons the existence of efficient algorithms for these problems is impossible.

The complexity of an algorithm is defined on the basis of formal machine models. Usually, these are idealised, yet universal models, designed in a way which facilitates formal reasoning about their behaviour. One of the first, and still maybe the most prominent of these models is the Turing machine. For Turing machines and other formal machine or programming models, computational complexity is defined in terms of the space and time requirements of computions.

Complexity theory usually deals with whole problem classes (generally countable sets of problem instances) instead of single instances. For a given algorithm or machine model, the complexity of a computation is characterised by the functional dependency between the size of an instance and the time and space required to solve this instance. For reasons of analytical tractability, many problems are formulated as decision problems, and time and space complexity are analyzed in terms of the worst asymptotic behaviour. Given a suitable definition of the computational complexity of an algorithm for a specific problem class, the complexity of the problem class itself can be defined as the complexity of the best algorithm on this problem class. Because generally time complexity is the more restrictive factor, problem classes are often categorised into complexity classes with respect to their asymptotic worst case time complexity.

Two particularly interesting complexity classes are $\mathcal{P}$, the class of problems that can be solved by a deterministic machine in polynomial time, and $\mathcal{NP}$, the class of problems which can be solved by a nondeterministic machine in prolynomial time. Of course, every problem in $\mathcal{P}$ is also contained in $\mathcal{NP}$, basically because deterministic calculations can be emulated on a nondeterministic machine. However, the question whether also $\mathcal{NP} \subseteq \mathcal{P}$, and consequently $\mathcal{P} = \mathcal{NP}$, is one of the most prominent open problems in computer science. Since many extremely application relevant fundamental problems are in $\mathcal{NP}$, but possibly not in $\mathcal{P}$ (*i.e.*, no polynomial time deterministic algorithm is known), this so-called $\mathcal{P} = \mathcal{NP}$-Problem is not only of theoretical interest. For these computationally hard problems, the best algorithms known so far have exponential time complexity. Therefore, for growing problem size, the problem instances become quickly intractable, and even the tremendous advances in hardware design have little effect on the size of the problem instances solvable with state-of-the-art technology in reasonable time. Well-known examples of $\mathcal{NP}$-hard problems include the Propositional Satisfiability Problem (see also below), the Graph Colouring Problem, the Knapsack Problem, the Traveling Salesperson Problem, scheduling problems, and time table problems, to name just a few [GJ79].

Many of these hard problems from $\mathcal{NP}$ are closely related to each other and can be translated into each other in polynomial deterministic time (polynomial reduction). A problem, which is at least as hard as any other problem in $\mathcal{NP}$ (in the sense that each problem in $\mathcal{NP}$ can be polynomially reduced to it) is called *$\mathcal{NP}$-hard*. Thus, $\mathcal{NP}$-hard problems in a certain sense can be regarded as at least as hard as every problem in $\mathcal{NP}$. But they do not necessarily have to belong to the class $\mathcal{NP}$ themselves, as their complexity might be actually much higher. $\mathcal{NP}$-hard problems which are contained in $\mathcal{NP}$ are called *$\mathcal{NP}$-complete*; in a certain sense, these problems are the hardest problems in $\mathcal{NP}$.

One fundamental result of complexity theory states that it suffices to find a polynomial time deterministic algorithm for one single $\mathcal{NP}$-complete problem to prove that $\mathcal{NP} = \mathcal{P}$. This is a consequence of the fact, that all $\mathcal{NP}$-complete problems can be encoded into each other in polynomial time. Today, most computer scientists believe that $\mathcal{P} \neq \mathcal{NP}$; however, so far all efforts of finding a proof for this inequality have been unsuccessful and there is some indication that today's mathematical methods might be too weak to solve this fundamental problem.

Many practically revelant problems from application domains, such as scheduling and planning problems, are $\mathcal{NP}$-complete and therefore generally not efficiently solvable today (and maybe, if $\mathcal{NP} \neq \mathcal{P}$, not efficiently solvable at all). However, being $\mathcal{NP}$-complete or $\mathcal{NP}$-hard does not mean for a problem that is impossible to practically solve it efficiently. Practically, there are at least three ways of dealing with these problems:

- Try to find an application relevant subclass of the problem which can be efficiently solved.
- Use efficient approximation algorithms.
- Use stochastic approaches.

Regarding the first strategy, we have to keep in mind that $\mathcal{NP}$-hardness is a property of a whole problem class P, whereas in practice, often only instances from a certain subclass $P' \subseteq P$ occur. And of course, P' does not have to be $\mathcal{NP}$-hard in general, *i.e.*, while for P an efficient algorithm might not exist, it might still be possible to find an efficient algorithm for the subclass P'.

If, however, the problem at hand is an optimisation problem which cannot be narrowed down to an efficiently solvable subclass, another option is to accept approximations of the solution instead of trying to compute optimal solutions. This way, in many cases the computational complexity of the problem can be sufficiently reduced to make the problem practically solvable. In some cases, allowing a comparatively small margin from the optimal solution makes the problem deterministically solvable in polynomial time. In other cases, the approximation problem remains $\mathcal{NP}$-hard, but the reduction in computational effort is sufficient to find acceptable solutions of practically occurring problem instances.

Sometimes, however, even reasonably efficient approximation schemes cannot be devised or the problem is a decision problem, to which the notion of approximation cannot be applied at all. In these cases, one further option is to focus on probabilistic rather than deterministic algorithms. At the first glance, this idea has quite an appeal: After all, according to the definition of the complexity class $\mathcal{NP}$, at least $\mathcal{NP}$-complete problems can be efficiently solved by nondeterministic machines. But this, of course, is of little practical use, since for an actual probabilistic algorithm this means only, that there is a chance, however small, that it can solve the given problem in polynomial time. In practice, the success probability of such an algorithm can be arbitrarily small. Nevertheless, in numerous occasions, probabilistic algorithms have been found to be considerably more efficient on $\mathcal{NP}$-complete or -hard problems than the best deterministic methods available. In other cases, probabilistic methods and deterministic methods complement each other in the sense, that for certain types of problem instances one or the other have been found to be superior. Among the most fundamental and best known problems from this category are the satisfiability problem in propositional logic (SAT) and the constraint satisfaction problem (CSP), which are formally introduced in the next sections.

1.5 Propositional Logic and Satisfiability (SAT)

Propositional logic is based on a formal language over an alphabet comprising propositional variables, truth values and logical operators. Using logical operators, propositional variables and truth values are combined into propositional formulae which represent propositional statements. Formally, the syntax of propositional logic can be defined by:

Definition 1.1 (Syntax of Propositional Logic)

$S = V \cup C \cup O \cup \{(,)\}$ is the *alphabet of propositional logic*, with $V = \{x_i \mid i \in \mathbb{N}\}$ denoting the countable infinite set of *propositional variables*, $C = \{\top, \bot\}$ the set of *truth values* and $O = \{\neg, \wedge, \vee\}$ the set of *propositional operators*.

The set of *propositional formulae* is characterised by the following inductive definition:

- the truth values $\top$ and $\bot$ are propositional formulae;
- each propositional variable $x_i \in V$ is a propositional formula;
- if F is a propositional formula, then $\neg F$ is also a propositional formula;
- if F_1 and F_2 are propositional formulae, then also $(F_1 \wedge F_2)$ and $(F_1 \vee F_2)$ are propositional formulae. □

Assignments are mappings from propositional variables to truth values. Using the standard interpretations of the logical operators on truth values, assigments can be used to evaluate propositional formulae. This way, we can define the semantics of propositional logic:

Definition 1.2 (Semantics of Propositional Logic)

The *variable set* $Var(F)$ *of formula* F is defined as the set of all the variables appearing in F.

A *variable assignment of formula* F is a mapping $B : Var(F) \mapsto C$ of the variable set of F to the truth values. The *complete set of assignments of* F is denoted by $Assign(F)$.

The *value* $Val_B(F)$ *of formula* F *under assignment* B is defined by induction on the syntactic structure of F:

- $F \equiv \top \Longrightarrow Val_B(F) = \top$
- $F \equiv \bot \Longrightarrow Val_B(F) = \bot$
- $F \equiv x_i \in V \Longrightarrow Val_B(F) = B(x_i)$
- $F \equiv \neg F_1 \Longrightarrow Val_B(F) = \neg\, Val_B(F_1)$
- $F \equiv F_1 \wedge F_2 \Longrightarrow Val_B(F) = Val_B(F_1) \wedge Val_B(F_2)$
- $F \equiv F_1 \vee F_2 \Longrightarrow Val_B(F) = Val_B(F_1) \vee Val_B(F_2)$

The truth values $\top$ and $\bot$ are also known as *verum (true)* and *falsum (false)*, resp.; the operators $\neg$ (*negation*), $\wedge$ (*conjunction*), and $\vee$ (*disjunction*) are defined by the following truth tables:

$\neg$	
$\top$	$\bot$
$\bot$	$\top$

$\wedge$	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

$\vee$	$\top$	$\bot$
$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$

$\square$

Because the variable set of a propositional formula is always finite, the complete set of assignments for a given formula is also finite. More precisely, for a formula containing n variables there are exactly 2^n variable assignments.

Considering the values of a formula under all possible assignments, the fundamental notions of validity and satisfiability can be defined in the following way:

Definition 1.3 (Satisfiability & Validity)

A variable assignment B is a *model of formula* F, exactly if $Val_B(F) = \top$; in this case we say that B *satisfies* F.

A formula F is *valid*, if it is satisfied by all its variable assignments. Valid formulae are also called *tautologies*.

If for a formula F there exists at least one model, F is *satisfiable*. $\square$

Now, we can formally define the Satisfiability Problem (SAT) and the Validity Problem (VAL) for propositional logic.

Definition 1.4 (The SAT and VAL problems)

For the *Satisfiability Problem (SAT) in propositional logic*, the goal is to decide for a given formula F, whether or not it is satisfiable.

For the *Validity problem (VAL) in propositional logic*, the goal is to decide for a given formula F, whether or not it is valid. $\square$

For SAT, there are two variants of the problem, the decision variant and the model finding variant. In the decision variant, only a yes/no decision for the satisfiability of the given formula is required, while for the model finding variant, in case the formula is satisfiable, a model has to be found. Note that there is a simple relationship between SAT and VAL: For each formula F, F is valid if and only if $\neg F$ is not satisfiable, *i.e.*, a formula is valid exactly if its negation has no model. For that reason, decision procedures for SAT can be used to decide VAL and vice versa. This, however, does not hold for incomplete procedures, such as the SLS algorithms for SAT discussed in this work. Since these cannot determine whether a given formula is *not* satisfiable, they cannot be used for establishing validity.

Often, problems like SAT and VAL are studied for syntactically restricted classes of formulae. Imposing syntactical restrictions usually facilitates theoretical studies and can also be very useful for simplifying algorithm design. Normal forms are syntactically restricted formulae such that for an arbitrary formula F there is always at least one normal form formula F'. Thus, each normal form induces a subclass of propositional formulae which is as expressively powerful as full propositional logic. Because for certain syntactically restricted classes of formulae it is not proven whether or not they are normal forms in that sense, we will use the term "normal form" in a more general sense here, covering reasonably expressively powerful subclasses of propositional logic which are characterised by syntactical restrictions. Some of the most commonly used normal forms are subsumed by the following definition.

Definition 1.5 (Normal forms)

A *literal* is a propositional variable (called a positive literal) or its negation (called a negative literal). Formulae of the syntactic form $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ are called *conjunctions*, while formulae of the form $d_1 \vee d_2 \vee \ldots \vee d_n$ are called disjunctions.

A propositional formula F is in *conjunctive normal form (CNF)*, if it is a conjunction over disjunctions of literals. The disjunctions are called *clauses*. A CNF formula F is in k-CNF, if all clauses of F contain exactly k literals.

A propositional formula F is in *disjunctive normal form (DNF)*, if it is a disjunction over conjunctions of literals. In this case, the conjunctions are called *clauses*. A DNF formula F is in k-DNF, if all clauses of F contain exactly k literals.

A CNF formula F is *Horn*, if every clause in F contains at most one unnegated variable. □

SAT is the prototypical $\mathcal{NP}$-complete problem. Historically, it was the first problem for which $\mathcal{NP}$-completeness was established [Coo71]. $\mathcal{NP}$-completeness of SAT can directly be proven by encoding the calculations of a Turing machine M for an $\mathcal{NP}$ problem into a propositional formula the models of which correspond to the accepting computations of M. Furthermore, it is quite easy to show that SAT is $\mathcal{NP}$-complete when restricted to CNF or even 3-CNF formulae [Rei90]. At the other hand, SAT is decidable in linear time for DNF, for Horn formulae [DG84], and for 2-CNF [Coo71].

As outlined above, VAL is equivalent to the complement of SAT. Therefore, VAL is co-$\mathcal{NP}$-complete. While the same holds for VAL on DNF formulae, VAL is polynomially decidable on CNF formulae. Note that to date, it is unknown, whether co-$\mathcal{NP}$ is different from $\mathcal{NP}$. In an intuitive sense, the VAL problem could actually be harder than SAT because while a solution for SAT can be verified in polynomial (actually linear) time, it is not clear how this should be generally done for a solution of VAL.

1.6 Constraint Satisfaction Problems (CSP)

A *constraint satisfaction problem (CSP)* is a problem defined by a set of variables each of which can take a number of values from some domain and a set of constraining conditions involving one or more variables. Depending whether the variable domains are discrete or continous, finite or infinite, different types of CSPs can be defined. In the context of this work, we restrict our attention mainly to *finite discrete CSPs.* This problem class can formally be defined as follows:

Definition 1.6 (Finite discrete CSP)

A constraint satisfaction problem can be defined as a triple $P = (X, \mathcal{D}, \mathcal{C})$, where $X = \{x_1, \dots, x_n\}$ is a finite set of n variables, $\mathcal{D} = \{D_1, \dots, D_n\}$ a set of domains and $\mathcal{C} = \{C_1, \dots, C_k\}$ a finite set of constraints. Each constraint C_j is a relation $C_j \subseteq X_{j_1} \times \dots \times X_{j_{\#j}}$ on $\#j$ variables.

P is a *finite discrete CSP*, if all the D_i are discrete and finite.

A *variable assignment for P* is a vector $A = (d_1, \dots, d_n)$ containing exactly one value $d_i \in D_i$ for each variable x_i.

If $\mathcal{A}$ denotes the set of all possible assignments for P, then $S \in \mathcal{A}$ is a *solution of P* exactly if it simultaneously satisfies all constraints in $\mathcal{C}$, *i.e.*, if for $S = (s_1, \dots, s_n)$ and all $C_j \in \mathcal{C}$, $C_j \subseteq X_{j_1} \times \dots \times X_{j_{\#j}}$ we have $(s_{j_1}, \dots, s_{j_{\#j}}) \in C_j$. $\square$

Of course, for this type of CSP the set of all possible assignments is finite. As SAT, the class of finite discrete CSPs is $\mathcal{NP}$-complete. This can be proven quite easily as there is a close relation between propositional SAT and finite discrete CSPs: SAT on CNF formulae corresponds directly to a finite discrete CSP where all the domains contain only the boolean values $\top, \bot$ and each constraint contains exactly all the satisfying assignments of one particular clause. Vice versa, each finite discrete CSP can be directly transformed into a SAT instance, where each propositional variable represents a particular assignment of one CSP variable and each constraint is represented by a number of clauses. Note, however, that this type of direct encoding does generally not yield a CNF formula, but a formula in CDNF (conjunctive disjunctive normal form, *i.e.*, a conjunction over disjunctions over conjunctions of literals) which can then be transformed into CNF.

Although CSP appears to be more adequate for the representation of hard combinatorial problems than SAT, SAT is a syntactically considerably simpler domain, and therefore designing and analysing algorithms tends to be easier for SAT than for CSP. At the same time, most SLS algorithms for SAT can be generalised to CSP in a straightforward way, and techniques which are working well for SAT tend to show good performance on CSP as well. Consequently, concepts and algorithms for SAT solving are in many respects more advanced than the corresponding CSP development, as can be seen in the recent literature. Mainly for this reason, we focus on SAT instead of CSP in the context of this thesis. Nevertheless, CSP is an important and interesting problem class; and, as we will see later, it is particularly useful as an intermediate step when transforming problems from other domains into SAT.

1.7 Related Work

Due to the introductory nature of this chapter, there is a huge body of literature related to the concepts presented here. Introductions to AI problems and search methods can be found in any modern or classis textbook on AI (such as [PMG98, RN95, Gin93, Nil80], *etc.*); for details on heuristic search, see also [Pea84]. For reference purposes, the Encyclopedia of Artificial Intelligence [Sha92] provides information on most AI concepts and approaches. An introduction to automated deduction and theorem proving can be found, *e.g.*, in [Bib92]. The generic problem solving approach is folklore in AI as well as in computer science in general; it is, however, tightly related to general issues of knowledge representation (see see [Bib93] for further references). For a detailed discussion of complexity theory, $\mathcal{NP}$-completeness, and $\mathcal{NP}$-hard problems, see [GJ79] or [Rei90]; Constrained Satisfaction problems and solving techniques are described in [Mac92]. Regarding stochastic local search methods for SAT and CSP, seminal studies are [SLM92], [Gu92], and [MJPL90]. Most of the literature on SLS algorithms for AI problems can be found in journals or the proceedings of the major AI conferences, as mentioned in the Prologue; the appropriate references are given in throughout this work, especially in the related work sections of the succeeding chapters.

1.8 Conclusions

In this chapter, we gave a brief introduction to AI problems. We distinguished various types of problems, representational versus computational, and decision versus optimisation problems. As we have argued, many important problems in AI are formalised in the context of logics, and can be interpreted as search problems. We introduced and briefly discussed two algorithmic approaches for solving these problems, systematic and local search algorithms, and reported some of the successes which could recently be achieved using stochastic local search techniques.

We then outlined a generic approach for solving hard problems by transforming them into an intermediate domain and solving them using stochastic local search methods. This technique has recently been successfully applied to some problem domains, yet, as of this writing, it is not clear whether these results can be achieved for a larger number of interesting problems. However, the overall approach of solving problems by mapping them onto alternate representations is widely used in computer science in general, and AI in particular. The propositional satisfiability problem (SAT) seems to be a good candidate for an intermediate domain, since it has sufficient expressive power for covering a large class of interesting problems. At the same time, even when compared to tightly related problems like CSP, SAT has the advantage of being syntactically and conceptionally very simple — which facilitates the development and analysis of algorithms.

Like many AI problems, SAT is $\mathcal{NP}$-hard, thus there is little hope for finding algorithms with better than exponential worst-case behaviour. However, there is still the chance that interesting or application relevant subclasses of these problems are algorithmically tractable. But as we argued, even if for a particular problem this is not the case, transforming problems into SAT and solving them using general SAT algorithms could still be advantageous, *e.g.*, because improving these algorithms pays off for a potentially large number of domains.

Certainly, the concept of generic problem solving by transformation into and solving within an intermediate domain is very elegant. The practical viability of this approach is suggested by the recent impressive successes which are mainly based on the dramatic improvements in SLS techniques for solving hard SAT instances. Thus, within the AI community there is a strong and continuously growing interest in topics like the design and analysis of modern SAT algorithms, SAT encodings, and their applications. These issues are investigated in the remainder of this thesis; while the question, whether generic problem solving using SAT encodings and SLS algorithms for SAT is a viable solution strategy for a large number of hard application problems, remains open, we are convinced that the results presented in the subsequent chapters can be regarded as important contributions to the analysis, improvement, and understanding of the overall approach.