INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig-
Maximilians-
Universität
München
LMU

Machine Learning in the Phase Transition Framework

Ulrich Rückert

Diplomarbeit

Beginn der Arbeit:	28.1.2002
Abgabe der Arbeit:	30.4.2002
Betreuer:	Prof. Dr. François Bry (Ludwig-Maximilians-Universität München)
	Prof. Dr. Luc de Raedt (Albert-Ludwigs-Universität Freiburg)
	Dr. Stefan Kramer (Albert-Ludwigs-Universität Freiburg)

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 30.4.2002 Ulrich Rückert

Zusammenfassung

Wir untersuchen k-term DNF Learning, d.h. das Problem, eine aussagenlogische Formel mit höchstens k Klauseln aus positiven und negativen Beispielen herzuleiten. Obwohl dieses Problem NP-vollständig ist, liegt es im Kern den meisten aussagenlogischen Lern-Algorithmen zugrunde. Das Ziel dieser Diplomarbeit ist es, die Anwendung von stochastischen lokalen Suchalgorithmen zur Lösung von k-term DNF Learning zu untersuchen. Als Referenzlösung entwerfen und bewerten wir zunächst eine Reihe von systematischen Suchalgorithmen. Mit Hilfe dieser Algorithmen können wir lösbare von unlösbaren Probleminstanzen unterscheiden und dann mit Hilfe des Phasenübergangs-Rahmens die durchschnittliche Such-Komplexität dieses Problems einschätzen. Wir untersuchen den Bereich der schwierigen Probleminstanzen, den sogenannten Phasenübergang und zeigen, dass die Position und die Ausmaße dieses Bereichs mit Hilfe einer Gleichung bestimmt werden können, die aus der statistischen Mechanik hergeleitet ist. Schließlich untersuchen wir die Anwendung von bekannten stochastischen lokalen Suchalgorithmen auf schwierige Probleminstanzen aus dem Phasenübergangs-Bereich. Unsere Experimente ergeben, dass WalkSAT in der Lage ist, den größten Anteil an schwierigen Probleminstanzen zu lösen. Zuletzt entwerfen und bewerten wir eine Reihe von stochastischen lokalen Suchalgorithmen zur Suche in den beiden möglichen Suchräumen für k-term DNF Learning. Eine geschwindigkeitsoptimierte Version einer WalkSAT nachempfundenen Suche im Formelraum ist in der Lage, Lösungen für den Leistungstest des Schach-Endspiels König-Turm-König zu finden, die kompakter sind als die Lösungen, die von einer aussagenlogischen Version des Regellernalgorithmus FOIL gefunden werden.

Abstract

We investigate k-term DNF learning, the task of inducing a propositional DNF formula with at most k terms from a set of positive and negative examples. Though k-term DNF learning is NP-complete, it is at the core of most propositional learning algorithms. The main aim of this thesis is to evaluate stochastic local search algorithms to solve hard k-term DNF learning tasks. As a reference we first design and evaluate a range of systematic search algorithms for the problem. Using those algorithms we can separate soluble from insoluble problem instances and then analyze the problem's average search complexity using the Phase Transition framework. We examine the region of hard problem instances, the so called phase transition, and show that location and extent of this region can be predicted by an equation derived from statistical mechanics. We then evaluate existing stochastic local search algorithms on the hard problem instances from the phase transition region. Our experiments indicate that WalkSAT is able to solve the largest fraction of hard problem instances. Finally, we design and evaluate a range of stochastic local search algorithms searching through the two possible search spaces for k-term DNF learning. A speed-optimized version of a WalkSAT-like search in formula space is able to find solutions to the king-rook-king chess endgame benchmark, which are more compact than the solutions found by a propositional version of the rule learning algorithm FOIL.

Contents

1 Introduction **1**

1.1 Concept Learning 1

1.1.1 Motivation 1

1.1.2 Definition 1

1.1.3 The Inductive Bias 2

1.1.4 Occam's Razor 3

1.1.5 PAC Learnability 3

1.2 Learning DNF Formulae: Background and Motivation 5

1.2.1 DNF Formulae 5

1.2.2 Learning as Search 5

1.3 Outline of the Thesis 6

2 K-Term DNF Learning **9**

2.1 The k-term DNF Learning Problem 9

2.1.1 Preliminaries 9

2.1.2 Definition 10

2.2 Problem Characteristics 11

2.2.1 Upper Bounds for k 11

2.2.2 NP-Completeness 12

2.2.3 Learnability 12

2.3 Problem Relevance 13

2.4 Least General Specialization and Least General Generalization 14

2.5 Reducing k-term DNF to SAT 15

3 Complete Algorithms **17**

3.1 Searching the Solution Space 17

3.2 Searching the Formula Space 18

3.2.1 A Trivial Algorithm . . . 18
3.2.2 A Refined Algorithm . . . 18
3.2.3 Results . . . 19
3.3 Searching the Partitioning Space . . . 20
3.3.1 Main Idea . . . 20
3.3.2 Recursion over k . . . 21
3.3.3 Recursion over $|Pos|$ . . . 22
3.3.4 Algorithm Performance . . . 24
3.4 Randomized Complete Algorithms . . . 25
3.4.1 Randomized Algorithms . . . 26
3.4.2 Heavy-tailed Distributions . . . 27
3.4.3 Algorithm Performance . . . 28

4 The Phase Transition **29**
4.1 The Phase Transition Framework . . . 29
4.1.1 History and Background . . . 29
4.1.2 The Phase Transition in Combinatorial Search . . . 30
4.2 The Phase Transition in k-term DNF Learning . . . 31
4.2.1 The Phase Transition . . . 31
4.2.2 The Location of the Phase Transition . . . 32
4.2.3 Finite Size Scaling . . . 35
4.3 Conclusions . . . 36

5 Stochastic Local Search **37**
5.1 Stochastic Local Search . . . 37
5.1.1 Background and History . . . 37
5.1.2 The GSAT and WalkSAT Architectures . . . 38
5.2 Using SLS on SAT-encoded k-term DNF Learning . . . 40
5.2.1 The Kamath et al. Test Set . . . 40
5.2.2 Results . . . 40
5.3 Using SLS on k-term DNF Learning . . . 41
5.3.1 Motivation . . . 42
5.3.2 GSAT-style Algorithms . . . 42
5.3.3 Weighting and Averaging . . . 45
5.3.4 WalkSAT-style Algorithms . . . 46

5.3.5 Optimization 49
5.4 The King-Rook-King Endgame Benchmark 51
5.5 Conclusions 52

6 Summary and Outlook **55**
6.1 Summary 55
6.2 Outlook 56

Bibliography **57**

Chapter 1

Introduction

The aim of this introduction is to give the background and motivation for the work presented in this thesis. After defining the task of concept learning, we introduce the inductive bias and the PAC-learnability framework, two basic concepts from computational learning theory. We then discuss learning as a search problem and give a short outline of the thesis.

1.1 Concept Learning

Concept learning is one of the central learning settings in machine learning. There is a broad range of theoretical and empirical research on concept learning tasks, and many concept learning algorithms have been proposed and evaluated.

1.1.1 Motivation

Imagine a car dealer having a large database with data about cars at his disposal. The database might include information such as the car's extents, color, technical data such as the power of the car's engine, and configuration details such as the presence or absence of a window regulator. Let's assume, however, that the dealer is mainly interested in knowing whether or not a certain car in this database is a sports car. This kind of information is not explicitely stated in the database. Of course, the dealer could label each database record by hand or query the database for records with typical attribute values. A less troublesome approach might be to label a small subset of the database records and let a computer program inductively learn the definition of the concept "x is a sports car". The learned concept definition can then be used to automatically label the remaining records in the database. This latter approach is an example of a concept learning task.

1.1.2 Definition

Of course, there is a broad range of different learning settings and ways to formalize concept learning. The following terminology is used in this thesis:

- At the core of each concept learning task is a pool of information about the entities,

which are classified by the concept. Following the terminology in [Mit97] we denote the entities as *instances*. In the previous example, a record about a car is an instance.

- The information about the instances is specified as a value assignment to a set of *attributes*. For example, the car database might specify that the "Color" attribute of a particular car has the value "blue", while the "Max. rotations per minute" attribute of another car might be "7200".
- The learner is presented a set of *training examples*. Examples that match the definition of the concept are called *positive examples*, the examples, which do not match, are called *negative examples*. The records labelled as "is a sports car" in the subset are positive, the ones labeled as "is not a sports car" are negative in our preceding example.

We are only interested in Boolean concepts, i.e. each instance is either of a given concept or it is not. Furthermore we assume that there is an actual *target concept*, which classifies all possible instances correctly and which we would like to learn. With that we can formally state a definition of the concept learning task:

Given

- an *instance space* X,
- a (possibly infinite) set H of hypotheses (the *hypothesis space*),
- a *target concept* $c : X \to \{0, 1\}$,
- a set $Pos \subset X$ of positive training examples, and
- a set $Neg \subset X$ of negative training examples with $Pos \cap Neg = \emptyset$.

Find a hypothesis $h \in H$ such that $h(x) = c(x)$ for all $x \in X$.

A hypothesis $h \in H$ is said to be *consistent* with the examples in a set S, if $h(x) = c(x)$ for all $x \in S$. Note that there is no guarantee that we can identify the correct h from Pos and Neg or even that a hypothesis equivalent to c is contained in H. However, there is a (well-founded) hope that any hypothesis found to approximate the training set well will also approximate the target concept well over the unobserved instances.

1.1.3 The Inductive Bias

When designing a concept learning algorithm one has to decide how to choose H and how to select h from H given Pos and Neg. An obvious choice is to represent a hypothesis by a subset $h \subset X$. The hypothesis space then is the set of all possible concepts, that is, the power set of X. Now, whenever we examine a positive example p in Pos, we can rule out all hypotheses $h \in H$, which do not contain p. Accordingly we can rule out every $h \in H$ that does contain one or more negative examples from Neg. Unfortunately, there is still a huge amount of hypotheses left, that are consistent with Pos and Neg. More specifically, since the power set of a finite hypothesis space is of size $2^{|H|}$, there are still $2^{|H \setminus (Pos \cup Neg)|}$ consistent hypotheses left. Even worse, we have no information on which of the many consistent hypotheses the algorithm should output, because all of them are equally likely to be equivalent to the target concept. Mitchell formulates this fundamental dilemma of inductive inference as follows: "a

learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances" [Mit97].

There are two possible remedies to this dilemma:

1. We can choose a H, which is smaller than the set of all possible hypotheses, so that the set of consistent hypotheses converges quickly to one hypothesis, which is the target concept.
2. We choose the set of all possible hypotheses as the hypothesis space, but we favour certain hypotheses within H over others.

In both cases we must make some (ideally well educated) guess on which hypotheses we should prefer. This decision, the so called *inductive bias* is an important characteristic of every concept learning algorithm. An algorithm which uses the first option, a smaller hypothesis space, is said to have a *restriction bias* or *language bias*, an algorithm which prefers some hypotheses over others is said to have a *preference bias*. Often, a preference bias is more desirable, because it ensures that the target concept is contained in H.

1.1.4 Occam's Razor

A very popular inductive bias is based on *Occam's razor* [Mit97]:

> **Occam's Razor:** Prefer the simplest hypothesis that fits the data.

Of course, one can use various aspects such as syntactic, semantic, epistemological, or pragmatic properties to determine the simplicity of a given hypothesis. In practice, most concept learning algorithms prefer the hypothesis with the shortest representation. For example, if we represent a concept as a set of rules, then a learning algorithm utilizing Occam's razor would choose the concept with the fewest rules.

Sometimes, people express the expectation that the application of Occam's razor increases the predictive accuracy (i.e. the fraction of correctly classified instances outside the training set) of a learning algorithm. This is usually justified by the claim that simpler hypotheses are supposed to better capture the instance space's inherent structure or that a simple hypothesis is less susceptible to overfitting, i.e. finding hypotheses which perform very well on the training set, but which are too specific to excel on other test sets. The validity of these claims is controversial. According to [Dom99], there are no theoretical results supporting those claims. Even worse, [Dom99, Web96] describe examples of real-world problems, where the application of Occam's razor actually decreases the predictive accuracy. It is, however, uncontroversial that simplicity is desirable in itself. Thus, if an algorithm finds two hypotheses with more or less the same generalization performance, it is reasonable to prefer the shorter one, not because it promises to further improve predictive accuracy, but because people prefer simpler explanations.

1.1.5 PAC Learnability

Within the field of Computational Learning Theory there has been a lot of research on whether or not a concept can be efficiently learned under various settings. The most prominent

framework for this is Valiant's *Probably Approximately Correct Learning* (PAC) model [Val84]. In this model the training examples are randomly chosen according to an unknown distribution D. Since the learner has no information about D, we can not expect to identify the true target concept in every case. Instead we allow the algorithm to output only in a "reasonably large amount of cases" a "reasonably well approximated" hypothesis. The more confidence we want to have on the correctness of the hypothesis, the more time we need to allow the algorithm for examining labelled examples. We measure the error of a hypothesis with respect to the distribution D:

Definition 1.1.1 *The* true error *of hypothesis h with respect to target concept t and distribution D is the probability that h will misclassify an instance drawn at random according to D:*

$$error_D(h) =_{def} \Pr_{x \in D}[t(x) \neq h(x)]. \tag{1.1}$$

There are various versions of the PAC model. First, we give the definition of the *Proper PAC model*:

Definition 1.1.2 *Let X be an instance space with instances of length n and H be a hypothesis space over X. If we give an algorithm L access to labeled examples chosen at random from X according to D, then H is said to be* efficiently PAC learnable *in the proper model by L, if for any target concept $t \in H$, any distribution D over X, any $0 < \epsilon < 1/2$, and any $0 < \delta < 1/2$, L outputs with probability at least $1 - \delta$ a hypothesis $h \in H$ with $error_D(h) \leq \epsilon$ in time polynomial in n, the encoding length of t, $1/\epsilon$, and $1/\delta$.*

There are usually two limiting factors, which might prevent a learning setting from efficiently being PAC learnable. The *sample complexity* describes the amount of examples required to identify a "reasonably well approximated" hypothesis, i.e. a hypothesis with a true error smaller than a given threshold. If a concept class requires an exponential amount of examples to identify such an h, then every algorithm will need exponential time just to read the examples. Such a concept class is certainly not efficiently PAC learnable. However, even if a learning setting is known to have polynomial sample complexity, an algorithm might need an exponential amount of time to compute and output the hypothesis h. Such an algorithm is said to have exponential *computational complexity*. The computational complexity of a learning setting depends very much on how the hypotheses are represented. There are sets of concepts, which are not efficiently learnable in the proper PAC model when represented by a certain hypothesis space, but that are efficiently PAC learnable when represented by a different hypothesis space. Thus, the *PAC prediction model* distinguishes between the actual concept class and the representation of the hypotheses. The concept class determines, which concepts a learning algorithm has to be able to learn (i.e. the restriction bias). The hypothesis space determines the used representation.

Definition 1.1.3 *Let X be an instance space with instances of length n, C be a set of concepts, and H be a hypothesis space over X. If we give an algorithm L access to labeled examples chosen at random from X according to D, then C is said to be* efficiently PAC learnable *using H by an algorithm L, if for any target concept $t \in C$, any distribution D over X, any $0 < \epsilon < 1/2$, and any $0 < \delta < 1/2$, L outputs with probability at least $1 - \delta$ a hypothesis $h \in H$ with $error_D(h) \leq \epsilon$ in time polynomial in n, the encoding length of t in H, $1/\epsilon$, and $1/\delta$.*

1.2 Learning DNF Formulae: Background and Motivation

Concepts can be expressed in a broad range of different ways. In this thesis we will use propositional logic to represent concepts, thus benefitting from a rich and well researched field. The aim of this thesis is to solve a certain class of learning problems using search algorithms.

1.2.1 DNF Formulae

A natural way to represent a concept is to construct a propositional formula over Boolean variables representing the attributes of the instances. The formula evaluates to *true*, if a given instance is of a given concept and $false$ otherwise. Though any Boolean function can be represented by many different, but equivalent formulae, choosing formulae in disjunctive normal form (DNF) has some advantages:

- Any Boolean concept can be expressed by a DNF formula. It is easy to build the (finite) set of all possible concepts, represented as DNF formulae. This is much harder, if we allow formulae to be of any arbitrary form. It is considerably easier to check for the equivalence of two formulae in DNF than for two unconstrained formulae.
- It is easy to find a model of a DNF formula. Thus, given a concept represented by a DNF formula, one can easily determine some positive instances for this concept. In contrast, finding a model of a formula in CNF is an NP-complete problem.
- Humans tend to favour DNF when describing concepts. For example, a typical dictionary entry like "A key is either a notched and grooved metal implement to open locks, or a vital crucial element, or a button that is depressed to operate a machine" resembles the form of a DNF formula: $(IsNotched \wedge IsGrooved \wedge OpensLock) \vee (IsVital \wedge IsCrucial) \vee (IsButton \wedge OperatesMachine)$.

Because of these properties DNF formulae are the most popular choice for representing Boolean concepts in propositional concept learning algorithms.

1.2.2 Learning as Search

If we restrict the representation of concepts to DNF formulae over a finite set of variables, the corresponding set of all possible concepts is finite and recursively enumerable. In this case a useful approach to learning is to search the space of all possible concepts for formulae consistent with the positive and negative examples. Because of the enormous size of such a search space, the practicability of using a search algorithm for concept learning depends very much on techniques to decrease the computational complexity of the search.

A *complete algorithm* searches the whole search space except for areas that are known in advance to not contain any solutions. Therefore a complete algorithm is guaranteed to find a solution if there exists one, and to correctly output "no solution present", if there is no solution. For many problems (especially NP-hard problems), complete algorithms require too much processing power to be used in practice. Even for easier problems one is often willing

to sacrifice the guarantee of finding a solution in order to find a solution quickly. This is especially true for most optimization problems, where one is usually not interested in the best possible, but a "reasonably good" solution. In those cases, *incomplete algorithms* like *stochastic local search* (SLS) can be applied successfully. While being faster than complete algorithms, incomplete algorithms do not search the whole search space and might therefore with a certain probability output "no solution found", even though there is one. The main aim of this thesis is to evaluate the application of stochastic local search to the problem of learning DNF formulae with at most k terms. In chapter 2 we will discuss this problem in detail.

1.3 Outline of the Thesis

The main aim of this thesis is to investigate and evaluate the application of a certain class of incomplete algorithms, the stochastic local search (SLS) algorithms, on the k-term DNF learning problem.

In chapter 2 we formally introduce the k-term DNF learning problem. We investigate some relevant characteristics of the problem, such as its NP-completeness and PAC-learnability results. We then present some problem-specific methods and concepts, which will be used in the latter parts of the thesis.

When evaluating incomplete algorithms, one needs to avoid some pitfalls. First of all, comparing only the performance of incomplete algorithms can be very misleading, because it gives no indication on how often the algorithm did not find a solution, though there is one. In order to estimate this error, a complete algorithm is needed to distinguish soluble from insoluble problem instances. Furthermore, a complete algorithm is a very good reference for performance comparisons: if an incomplete algorithm is only a little faster than the complete reference algorithm, then it is probably better to use the complete algorithm, because it ensures that a solution is found, if one is present. Thus, we present and evaluate complete algorithms for the k-term DNF learning problem in chapter 3.

When evaluating the performance of an incomplete algorithm, a suitable test set is needed to obtain meaningful results. Finding a good test set can be a hard problem as well. If we generate the problems in the test set randomly, we have to be very careful: not all problem instances are equally difficult. For many problem classes, there is no easy way to predict the difficulty of a problem instances from the problem parameters. For example, there are SAT problems with thousands of clauses, that can be solved in a few seconds by a complete algorithm, while other SAT problem instances with less than one hundred clauses require months to be solved. The solubility of NP-complete problems has been studied in the so-called phase transition framework. A phase transition is the region between obviously soluble and obviously insoluble randomly generated problem settings. This framework provides a way to estimate the average search costs for solving the problems. We examine the phase transition of k-term DNF learning in chapter 4

Finally, we use the results from this chapter to generate a hard test set, which we then use to evaluate SLS algorithms in chapter 5. We start with some of the published SLS algorithms for the SAT problem. Those algorithms do not work on k-term DNF learning directly, but on a SAT-encoded version of the problems. We then design and evaluate some new algorithms

that solve the problem instances using a "native" representation. It can be dangerous to test search algorithms only on randomly generated problem instances, because many real world problems might have some structural properties that are not (or only with very a low probability) present in random problems. We therefore present the performance of our SLS algorithms on a real-world problem, the problem of predicting winnable positions in a chess endgame. We conclude the thesis with a summary of our results and an outlook on some possible directions for further research in chapter 6.

Chapter 2

K-Term DNF Learning

This chapter introduces the k-term DNF learning problem. We give a formal definition and present some theoretical results about certain properties of the problem. We then introduce the least general generalization (lgg) and least general specialization (lgs). Finally we show that k-term DNF Learning can be reduced to the satisfiability problem.

2.1 The k-term DNF Learning Problem

We formulate the k-term DNF problem using the terminology of Propositional Logic. Note that the literature on inductive inference sometimes uses equivalent or similar terms from Boolean Algebra or Circuit Design Theory.

2.1.1 Preliminaries

Before we can formally state the k-term DNF learning problem, we need to introduce some basic concepts:

Definition 2.1.1 *The logical constant* true *is represented by the number 1 and the logical constant* false *is represented by the number 0.*

Definition 2.1.2 *Let* $V =_{def} \{v_1, v_2, v_3, \ldots, v_n\}$ *be a set of Boolean variables. A* literal *is either the variable* v_i *or its negation* $\neg v_i$*. A* clause *over variable set* V *is the disjunction of literals built from variables in* V*:* $\bigvee_{i \in V'} l_i$*,* $V' \subset V$*. A* term *over variable set* V *is the conjunction of literals built from variables in* V*:* $\bigwedge_{i \in V'} l_i$*,* $V' \subset V$*.*

Definition 2.1.3 *Let* $V =_{def} \{v_1, v_2, v_3, \ldots, v_n\}$ *be a set of Boolean variables. A* formula *over variable set* V *is a (possibly nested) combination of conjunctions, disjunctions, and negations of the variables in* V*. A formula in* Disjunctive Normal Form *(DNF) is a disjunction of terms. A formula in* Conjunctive Normal Form *(CNF) is a conjunction of clauses. A* k-term DNF formula *is a formula in Disjunctive Normal Form with exactly k terms.*

For example, $\neg v_2 \vee (\neg v_1 \wedge v_2 \wedge v_3)$ is a 2-term DNF formula over $V = \{v_1, v_2, v_3\}$. $(\neg v_1 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee \neg v_3)$ is a CNF formula over V. If we assign truth values to the variables in V, we can calculate the value of the Boolean function defined by the formula:

Definition 2.1.4 *Let F be a formula over a variable set $V =_{def} \{v_1, v_2, v_3, \ldots, v_n\}$. A* truth (value) assignment *over V is an assignment $x : V \to \{0,1\}$. We say that F* covers *a truth assignment a (alternatively: a* satisfies *F), if F evaluates to 1 when all variables v_i in F are replaced by $a(v_i)$, the truth value according to a. F is* uncovered *by a (alternatively: a* violates *F), if F does not cover a.*

In the following we will denote a truth value assignment a over $\{v_1, v_2, v_3, \ldots, v_n\}$ by a string $s \in \{0,1\}^*$ where the ith character in s is assigned to v_i. If we are given a truth assignment a we can easily construct a term $term(a)$ so that $term(a)$ covers a and only a:

Definition 2.1.5 *Let a be a truth value assignment over a variable set $V =_{def} \{v_1, v_2, v_3, \ldots, v_n\}$. Then:*

$$term(a) =_{def} \bigwedge_{i=1}^{|V|} t(a, v_i)$$

with:

$$t(a, v_i) =_{def} \begin{cases} v_i & \text{if } a(v_i) = 1 \\ \neg v_i & \text{if } a(v_i) = 0 \end{cases}$$

For each variable, which is assigned to 1 in a, v_i occurs unnegated in $term(a)$ and for each variable, which is assigned to 0 in a, $term(a)$ contains $\neg v_i$. Obviously, every a uniquely identifies exactly one $term(a)$ and every term t with $|Var|$ literals identifies exactly one truth assignment a. In the following we will not discriminate between a and $term(a)$ if it is obvious from the context whether we are talking about truth assignments or terms.

2.1.2 Definition

We can now formally introduce the k-term DNF learning problem [KV94]:

Given

- a set of Boolean variables Var,
- a set Pos of truth value assignments $p_i : Var \to \{0,1\}$,
- a set Neg of truth value assignments $n_i : Var \to \{0,1\}$ with $Pos \cap Neg = \emptyset$, and
- a natural number k,

Find a k-term DNF formula that is *consistent* with Pos and Neg, i.e. that is covered by all truth assignments in Pos and that is not covered by any truth assignment in Neg.

In the following we call the truth assignments in Pos *positive examples* and the truth assignments in Neg *negative examples*. The number of variables in Var is usually denoted by n.

Here is a short example: consider the learning problem with three variables $Var = \{v_1, v_2, v_3\}$, three positive examples $Pos = \{001, 011, 100\}$, two negative examples $Neg = \{101, 111\}$ and $k = 2$. Now let F_1 be $\neg v_2 \vee (\neg v_1 \wedge v_2 \wedge v_3)$. F_1 is not a solution to this k-term DNF learning problem, because the first negative example 101 is covered by the first term of F_1. However, $F_2 =_{def} (\neg v_1 \wedge v_3) \vee (v_1 \wedge \neg v_2 \wedge \neg v_3)$ is consistent with Pos and Neg: the first two positive examples are covered by the first term of F_2 and the third positive example is covered by the second term of F_2. None of the two negative examples is covered by any term in F. Therefore F_2 is a solution to the k-term DNF learning problem.

2.2 Problem Characteristics

The general k-term DNF learning problem is NP-complete. In the following we discuss bounds for k, the difference between the decision and the optimization version of the problem, and the implications on the PAC-learnability of DNF formulae with at most k terms.

2.2.1 Upper Bounds for k

The difficulty of finding a solution to a given k-term DNF learning problems depends very much on the value of k. In practice we are interested in finding solutions with a k that is as small as possible. To get some information on which values of k can be considered "small", we present some upper bounds of k. For all problems instances, whose k is larger than the upper bound, there is an easy way to identify a solution. Such a problem can be easily solved and there is no need to use one of the algorithms presented in chapter 3 and 5.

For example, if $k = |Pos|$, there is a trivial solution $F_t =_{def} \bigvee_{p \in Pos} term(p)$. Since F_t covers all positive examples and no other truth assignments, it is a solution to the k-term DNF learning problem. Thus we are only interested in learning problems with $k < |Pos|$. On the other hand, assume we know a solution F to a given k-term DNF learning problem. Then there is an easy way to construct solutions for all k'-term DNF learning problems with $k' > k$: select a random example p from Pos and add $\bigvee_{i=1}^{k'-k} term(p)$ to F. The resulting formula has k' terms and is still consistent with the examples, because $term(p)$ does not cover any other examples than p. Because of this, we can safely weaken the condition "formula needs to have exactly k terms" to "formula needs to have at most k terms".

The size of k is also limited by the number of variables n in the problem setting. Obviously, we can only generate 2^n discriminable examples, so $k \leq |Pos| \leq 2^n$. In the following we will show that we can find a solution with at most 2^{n-1} terms for all problem instances with n variables. The proof is designed as an induction over n. We start with the base case $n = 1$. The two possible problem settings with one variable v can be solved by v and $\neg v$ respectively, so $k \leq 1 = 2^{1-1}$ holds. Our inductive hypothesis is: assume that all possible problem instances with n variables can be solved by a formula with at most 2^{n-1} terms. Then every possible problem setting with $(n+1)$ variables can be solved by a formula with $2^{(n+1)-1} = 2^n$ terms.

To prove the hypothesis, assume we have an arbitrary problem instance (Var, Pos, Neg) with $(n+1)$ variables. Now let P_0 be the set of the problem's positive examples whose first variable is set to 0 and P_1 be the set of examples whose first variable is set to 1. Accordingly N_0 and

N_1 contain the negative examples, whose first variable is set to 0 and 1. Obviously, $P_0 \cap P_1 = \emptyset$ and $P_0 \cup P_1 = Pos$. Removing the first variable v_1 of the examples in P_0 and N_0 yields two example sets P_0' and N_0'. The resulting problem setting $(Var \setminus \{v_1\}, P_0', N_0')$ has n variables. Because of the inductive hypothesis, there is a solution F_0 with at most 2^{n-1} terms to this problem instance. Now, if we add the literal $\neg v_1$ to all terms in F_0, the resulting formula F_0' is a solution to the problem (Var, P_0, Neg): since all examples in P_0 satisfy $\neg v_1$ and the other literals of the terms are consistent with P_0', P_0 is covered by F_0'. N_0 is not covered by F_0', because N_0' is not covered by F_0. N_1 is not covered by F_0', because all examples in N_1 violate $\neg v_1$, which is a literal in all terms of F_0'.

Similar considerations can be made about P_1, N_1 and the solution F_1 of the induced problem instance $(Var \setminus \{v_1\}, P_1', N_1')$. Finally, we can argue that $F =_{def} F_0' \vee F_1'$ is a solution of the problem (Var, Pos, Neg), because all positive examples are covered by either F_0' or F_1'. F has at most $2^{n-1} + 2^{n-1} = 2^n$ terms, so the inductive hypothesis holds.

Note that 2^{n-1} is actually a sharp bound: for an arbitrarily chosen n, let the set Odd_n contain all possible examples, that assign an odd number of variables to 1, and $Even_n$, contains all examples, that assign an even number of variables to 1. Since for any pair of examples (p_1, p_2) with $p_1, p_2 \in Odd_n$ p_1 and p_2 differ in at least two literals, there is no term t covering only p_1 and p_2, but no example from $Even_n$. Thus, the problem given by $(Var, Odd_n, Even_n)$ has exactly one solution $F =_{def} \bigvee_{p \in Odd_n} term(p)$. F has exactly $|Odd_n| = \frac{2^n}{2} = 2^{n-1}$ terms.

2.2.2 NP-Completeness

In [KV94] Kearns et al. prove that k-term DNF learning is NP-complete. The proof is essentially a reduction of the k-Colorability problem to k-term DNF learning. As with most other NP-complete problems we can distinguish between a decision version and an optimization version of the problem. The *decision problem* is the version outlined in section 2.1: given a set of Boolean variables, a set of positive examples, a set of negative examples and a constant k we would like to figure out whether or not there is a solution, i.e. a consistent formula. The *optimization problem* is the problem of finding a formula with minimal number of terms: we are given a set of Boolean variables and the positive and negative examples. Now we would like to identify the minimal k so that there is a solution to the stated problem with exactly k terms. This problem is also known as *DNF minimization* problem.

As argued in [GW96a] there is a close relation between the problems: an algorithm for the decision problem can easily be generalized to solving the optimization problem (usually by adding branch and bound techniques). In the other direction, we can solve any decision problem by computing the minimal k (i.e. solving the optimization version). If the minimal k is smaller or equal than the desired k, the decision problem has a solution (as outlined in section 2.2.1), otherwise it has not.

2.2.3 Learnability

Because of its NP-completeness, k-term DNF learning is not PAC-learnable in the proper PAC model, unless NP=RP. However, since $\vee$ distributes over $\wedge$ (that is $((u \wedge v) \vee (w \wedge x) = (u \vee w) \wedge (u \vee x) \wedge (v \vee w) \wedge (v \vee x))$, every k-term DNF formula can be represented by an equivalent CNF formula whose clauses contain at most k literals. Therefore, the class

of concepts, which can be represented by a k-term DNF formula is a subset of the class of concepts that can be represented by a k-CNF formula, i.e. a formula with at most k literals per clause. It is a remarkable fact that the class of k-CNF formulae has been proven to be PAC learnable in the proper PAC model [KV94].

Indeed, the complexity of learning k-term DNF formulae depends on the representation: if we represent the concepts as k-term DNF formulae, the task is NP-complete. If we represent the same concepts as k-CNF formulae, the task is computable in polynomial time. In both cases we only need a polynomial amount of examples to uniquely (or with a given probability ϵ) identify the desired concept. Thus, k-term DNF learning has polynomial *sample complexity*, but exponential *computational complexity*.

One could avoid the exponential time complexity of learning the class of k-term DNF formulae by using k-CNF formulae as representation. However, when representing the concepts as k-CNF formulae, we lose some of the preferable properties of the DNF representation, as outlined in section 1.2.1. Even worse, the number of clauses in a k-CNF formula is not restricted. A DNF formula with k terms requires up to n^k clauses, if represented by a k-CNF formula. A consistent k-CNF formula with minimal k might have considerably more clauses than a consistent k-CNF formula with non-minimal k. Thus, minimizing k can be seen as an application of Occam's razor for k-term DNF formulae, but not for k-CNF formulae. If one would like to apply Occam's razor, k-CNF formulae are therefore not very well suited.

2.3 Problem Relevance

In Machine Learning, we are mainly interested in the optimization problem. In fact, the DNF minimization problem seems to be at the core of most propositional concept learning settings, in the sense that:

1. Concept learning problems with discrete attribute sets can be easily reformulated into a form that uses only two-valued (i.e. Boolean) attributes. This form corresponds exactly to our problem description. For problems with continuous-valued attribute sets, one can use one of the many published methods to generate a range of discrete attributes which approximate the continuous attributes.

2. Most propositional concept learners use representations that are subsets of or equivalent to DNF formulae. Learning disjunctive sets of rules is basically the same as learning DNF formulae. Decision trees can be represented by DNF formulae of a certain form. Representations in CNF are usually hard to understand for humans and therefore seldom used [Moo95].

3. Most concept learning algorithms include a bias towards a short representation of the hypotheses. As outlined in section 1.1.4, this might not necessarily increase the predictive accuracy. Nevertheless, short representations are an important goal per se. The goal of finding short DNF formulae is exactly the DNF minimization problem as described in section 2.2.2.

If framed as a concept learning task, the k-term DNF learning problem has a restriction bias: it considers only DNF formulae of a certain length k. The DNF minimization problem

corresponds to a concept learning task with preference bias: since every concept can be represented by a DNF formula, we can learn all possible concepts. However, we favour formulae with a minimal number of terms. In the following we will focus on the decision version, i.e. k-term DNF learning. All of our results can be easily generalized to the DNF minimization problem.

2.4 Least General Specialization and Least General Generalization

The number of literals in a term t determines how many examples the term covers. It is therefore reasonable to order terms according to their "generality". In fact, one can use a partial order $\preceq$ with:

$$t \preceq s \iff \text{each literal in } s \text{ is also contained in } t$$

to order the terms. The set of all terms over Var together with $\preceq$ is then a *lattice*. Though we will not give any detailed analysis of the algebraic structure of terms and formulae, there are two concepts that are relevant for our discussion about k-term DNF learning: the *least general specialization* and the *least general generalization*. The main motivation for those two constructs is to limit the (stochastic or systematic) search on *least general solutions* to a k-term DNF learning problem, i.e. the solutions that cover as few assignments as possible. In that way we are able to search a smaller space and boost the performance of the search algorithm.

As an example, assume we have discovered a solution F for a given problem instance. Upon closer inspection we might find out that some literals are redundant in F, i.e. removing or adding these literals from or to F would still yield a solution. To examine which and how many literals might be added to F, we can compute the *least general specialization* of F. The *least general specialization (lgs) of* F is a formula, that covers the same positive *examples* as F, but as few other *instances* as possible. To construct the *lgs*, we determine, which positive examples are covered by the individual terms in the solution:

$$Cov_i(F) =_{def} \{p \in Pos | \text{ the } i\text{-th term of } F \text{ is satisfied by } p\}$$

We can then compute the *lgs* using the *least general generalization (lgg)* of those positive examples. The *least general generalization* of a set of examples e (over the variables Var_i, $1 \leq i \leq n$) can be efficiently computed by merging the literals:

$$merge(i, e) =_{def} \begin{cases} Var_i & \text{if all examples in } e \text{ set } Var_i \text{ to 1} \\ \neg Var_i & \text{if all examples in } e \text{ set } Var_i \text{ to 0} \\ 1 & \text{otherwise} \end{cases}$$

$$lgg(e) =_{def} \bigwedge_{i=1}^{n} merge(i, e)$$

The *least general specialization* then is:

$$lgs(F) =_{def} \bigvee_{i=1}^{k} lgg(Cov_i(F))$$

In the following we will show that $lgs(F)$ is a solution if F is a solution. Therefore, if a problem instance P is soluble, it also has a solution which is a *least general specialization* of a solution of P. If a problem instance has only one solution F, then $F = lgs(F)$. In section 3.3 we will make use of this fact to search in the (smaller) space of least general solutions than in the (large) space of all possible solutions.

We will now prove that if F is a solution, $lgs(F)$ is a solution as well. Since F is a solution, every positive example is covered by a term of F and thus $\bigcup_{i=1}^{k} Cov_i(F) = Pos$. Note then that for any set of examples e the term $lgg(e)$ covers all examples in e, because $lgg(e)$ contains only literals that evaluate to 1 for the examples in e. Therefore, $lgs(F)$ covers all positive examples. Let t_j be an arbitrary term in $lgs(F)$ and let t'_j be the corresponding term in F. Now consider an arbitrary literal l' in t'_j: since the examples in $Cov_j(F)$ are covered by t'_j, all of them assign the variable in l' to the same value. From the definition of lgg follows that t_j also contains l'. Thus, the set of literals in t'_j is a subset of the set of literals in t_j. If t_j covers a negative example n, then t'_j covers n as well. A t_j which covers a negative example would therefore be a contradiction to the premise that F is a solution. Thus, no t_j can cover any negative examples and $lgs(F)$ is consistent with Pos and Neg.

2.5 Reducing k-term DNF to SAT

NP-complete problems can be translated into other NP-complete problems by using some "reduction" algorithm of polynomial time complexity [Kar72]. In the following, we show that the k-term DNF learning problem can be reduced to a satisfiability (SAT) problem. In chapter 5 we will make use of this reduction to solve SAT-encoded hard k-term DNF learning problems using some of the many available stochastic local search algorithms for SAT. There are various ways to reduce k-term DNF learning to SAT, but we chose the reduction published in [KKRR92], because it resembles the k-term DNF learning problem's inherent asymmetry quite well (see section 4.2.2 for a discussion of the asymmetry). We will only sketch the main idea of the reduction; a more detailed explanation is given in [KKRR92].

The SAT problem is defined as following:

Given

- a set of Boolean variables Var,
- a formula F in CNF

Find a truth assignment a that satisfies F.

Now let us examine a given k-term DNF learning problem (Var_{DNF}, Pos, Neg, k). To reduce this problem to a SAT problem (Var_{SAT}, F_{SAT}), we need to encode the solution S_{DNF} using the Boolean variables in Var_{SAT}:

$$s_{ji} =_{def} \begin{cases} 0 & \text{if } x_i \text{ is in the } j\text{-th disjunctive term of } S_{DNF} \\ 1 & \text{if } x_i \text{ is not in the } j\text{-th disjunctive term of } S_{DNF} \end{cases}$$

$$s'_{ji} =_{def} \begin{cases} 0 & \text{if } \neg x_i \text{ is in the } j\text{-th disjunctive term of } S_{DNF} \\ 1 & \text{if } \neg x_i \text{ is not in the } j\text{-th disjunctive term of } S_{DNF} \end{cases}$$

Each positive example in Pos needs to be covered by at least one term in S_{DNF}. We use another $k \cdot |Pos|$ variables to denote which term covers which positive example:

$$z_j^a =_{def} \begin{cases} 0 & \text{if the } a\text{-th positive example violates the } j\text{-th term of } S_{DNF} \\ 1 & \text{if the } a\text{-th positive example satisfies the } j\text{-th term of } S_{DNF} \end{cases}$$

With that, $Var_{SAT} =_{def} \{s_{ji}, s'_{ji}, z_j^a \mid 1 \leq i \leq |Var_{DNF}|, 1 \leq j \leq k, 1 \leq a \leq |Pos|\}$ has $(2 \cdot |Var_{DNF}| + |Pos|) \cdot k$ variables.

The desired CNF formula F_{SAT} consists of four kinds of clauses:

1. We must ensure that v_i and $\neg v_i$ are not part of the same term in S_{DNF}. Therefore F_{SAT} contains $|Var_{DNF}| \cdot k$ clauses of the form:

$$s_{ji} \vee s'_{ji} \text{ for } 1 \leq i \leq |Var_{DNF}|, 1 \leq j \leq k$$

2. No term may cover any negative examples. For every negative example n_r in Neg let P_r be the set of indices i so that n_r assigns the i-th variable v_i to 1. Then the following $|Neg| \cdot k$ clauses ensure that no negative example satisfies S_{DNF}:

$$(\bigvee_{i \in P_r} \neg s'_{ji}) \vee (\bigvee_{i \notin P_r} \neg s_{ji}) \text{ for } 1 \leq i \leq |Var_{DNF}|, 1 \leq j \leq k, 1 \leq r \leq |Neg|$$

3. Each positive example must be covered by at least one term of S_{DNF}. This is guaranteed by the following $|Pos| \cdot k$ clauses:

$$\bigvee_{1}^{k} z_{j=1}^a \text{ for } 1 \leq a \leq |Pos|$$

4. The clauses in 3 determine that every positive example must be covered by at least one term. However, we did not yet state how the literals in S_{DNF} need to be adjusted so that a given positive example satisfies a given term t in S_{DNF}. The following $|Var_{DNF}| \cdot k \cdot |Pos|$ clauses express the constraints put on the literals of t, if t covers a given positive example p. If p assigns a variable v_i to 1, then v_i must occur unnegated in t, if p assigns v_i to 0, it must occur negated in t.

$$\sigma_{ji}^a \vee \neg z_j^a \text{ for } 1 \leq i \leq |Var_{DNF}|, 1 \leq j \leq k, 1 \leq a \leq |Pos|$$

with:

$$\sigma_{ji}^a =_{def} \begin{cases} s'_{ji} & \text{if the } a\text{-th positive example assigns } v_i \text{ to } 1 \\ s_{ji} & \text{if the } a\text{-th positive example assigns } v_i \text{ to } 0 \end{cases}$$

Altogether, F_{SAT} contains $k \cdot (|Var_{DNF}| \cdot (|Pos| + 1) + |Neg|) + |Pos|$ clauses. It is obvious from the definition of F_{SAT} that any satisfying variable assignment specifies a formula S_{DNF} which is a solution to the original k-term DNF learning problem.

Chapter 3

Complete Algorithms

In this chapter, we present various complete algorithms for the k-term DNF learning problem. First, we frame the k-term DNF learning problem as a search problem. We show, that there are two different search spaces that can be utilized for solving k-term DNF learning. We then present three complete algorithms searching through those search spaces. Finally, we examine the distribution of solutions in the partitioning search space and evaluate randomized restarts to boost the search.

3.1 Searching the Solution Space

Assume we are given the parameters for a particular NP-complete problem. The problem specification together with the parameters determine exactly which constraints the output of an algorithm has to fulfil in order to be a solution. For the hard problem instances, only a tiny fraction of the set of all candidates will be a solution. We can therefore formulate the NP-complete problem as a search problem: given the parameters, we would like to search through the space of all possible outputs until we find one that fulfils all constraints. During this search we can often take advantage of the combinatorial structure of the search space: in some cases we can use heuristics to search first through the "interesting" parts of the space or we can use pruning techniques to avoid subspaces, which are known to contain no solution. Consequently, many well known complete algorithms are designed as a recursive search (e.g. the Davis Putnam procedure [DP60] for the satisfiability problem or Little's algorithm [LMSK63] for the Travelling Salesman problem).

For k-term DNF learning, we can choose between two possible search spaces: we can simply search the space of all k-term formulae, the *formula space*. Alternatively, we can make use of the results in section 2.4 and search only for least general solutions. If a problem is soluble at all, it also has a least general solution. In section 3.3 we will show that any least general solution can be represented by a partitioning of Pos into k pairwise disjoint subsets. An algorithm can therefore simply search the *partitioning space* to generate all potential least general solutions.

3.2 Searching the Formula Space

We start with an easy algorithm for searching the formula space and then refine the algorithm to improve the runtime behaviour.

3.2.1 A Trivial Algorithm

Our first (trivial) approach to k-term DNF learning is to recursively add terms to an initially empty formula. If all positive and no negative examples are covered during the final recursion step, we have a solution. If during the recursion at least one negative example is covered by term $i, 1 \leq i \leq k$, we can prune the search and continue without generating the terms $i, i+1, \ldots, k$. If the formula has reached size k and there are still positive examples uncovered, the algorithm backtracks. Algorithm 1 states the pseudo code.

Algorithm 1 The trivial complete algorithm for searching the formula space.

```
procedure SearchFormulaSpace(k)
  Search(empty formula, 0, k)
end procedure

procedure Search(formula, nrTerms, k)
  if nrTerms = k then
    if formula covers all positive examples then
      formula is a solution
    end if
  else
    for all possible terms t do
      Search(formula ∨ t, nrTerms + 1, k)
    end for
  end if
end procedure
```

3.2.2 A Refined Algorithm

Since any variable can either occur negated, unnegated, or not at all in any term, the size of the space searched by the trivial algorithm is $3^{n \cdot k}$ (n is the number of variables). Obviously, we can decrease the size of this space by exploiting some structural properties of DNF. First, the order of the terms does not matter: $(A \wedge B \wedge C) \vee (\neg A \wedge \neg B \wedge \neg C)$ is the same as $(\neg A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$. We can avoid processing equivalent formulae twice by ordering the terms according to an (arbitrary) order $\prec$. The recursion step then adds only terms which are larger according to $\prec$ than the largest term in the existing formula. The size of this search space is $\frac{3^{n \cdot k}}{k!}$, because there are $k!$ different ways to order the terms in the formula.

Second, some terms subsume others. For instance, $(A \wedge B) \vee (A \wedge B \wedge \neg C)$ is equivalent to $(A \wedge B)$, because $(A \wedge B)$ subsumes $(A \wedge B \wedge \neg C)$. Again, we can decrease the search space size by choosing $\prec$ so that t_1 is more general than t_2, if $t_1 \prec t_2$ and testing for subsumption before adding a term.

Third, sometimes two terms can be merged to a more general term. For example, $(A \wedge B \wedge C) \vee (A \wedge B \wedge \neg C)$ is equivalent to $lgg(A \wedge B \wedge C, A \wedge B \wedge \neg C) = (A \wedge B)$. If for two terms t_1, t_2 a literal l occurs negated in t_1 and unnegated in t_2 and the other literals are equal in t_1 and t_2, then $lgg(t_1, t_2)$ is more general than t_1 and t_2 and it covers exactly the same instances as $t_1 \wedge t_2$. Assume t_1 is already contained in the formula and we want to add t_2. Since we add the terms to the formula according to $\prec$, i.e. with decreasing generality, we know that $lgg(t_1, t_2)$ has already been added to the formula in a previous recursion step. Therefore, we can skip adding t_2, because the resulting formula is equivalent to a previously tested formula.

However, the search space of this refined algorithm still grows exponentially in $n \cdot k$. Algorithm 2 sketches this refined version.

Algorithm 2 A refined complete algorithm for searching the formula space.

```
procedure SearchFormulaSpace(k)
  Search(empty formula, 0, k)
end procedure

procedure Search(formula, nrTerms, k)
  if nrTerms = k then
    if formula covers all positive examples then
      formula is a solution
    end if
  else
    if nrTerms = 0 then
      term ← smallest term according to ≺
    else
      term ← largest term in formula according to ≺
    end if
    repeat
      term ← next term according to ≺
      if term is not subsumed and can not be merged with any other term in formula
      then
        if term does not cover any negative example then
          Search(formula ∨ term, nrTerms + 1, k)
        end if
      end if
    until term = largest possible term according to ≺
  end if
end procedure
```

3.2.3 Results

Even this refined search is still redundant. For instance, since the formula $F_1 : (C \wedge D \wedge \neg E) \vee (C \wedge \neg D \wedge E) \vee (B \wedge C \wedge D)$ is equivalent to $F_2 : (C \wedge D \wedge \neg E) \vee (C \wedge \neg D \wedge E) \vee (B \wedge C \wedge E)$, we could evaluate F_1 and skip F_2 (or vice versa). There are a couple of algorithms for determining the equivalence of Boolean formulae, but they are computationally too demanding to be used

in our case.

We tested the refined algorithm on three easy problem settings ($|Pos| = 10, |Neg| = 10, n = 10, k = 3$). Though the algorithms presented in the next section were able to find a solution to these problems within a few seconds, the formula space algorithm still ran after five minutes. We therefore cancelled our experiments. Clearly, this algorithm is only suitable for problems with very small n and k.

3.3 Searching the Partitioning Space

In the following we discuss searching the partitioning space, i.e. the space of least general solutions.

3.3.1 Main Idea

As shown in section 2.4, if a problem instance has a solution F, it also has a solution that is a *least general specialization*. We can leverage this fact to construct a complete algorithm for solving the k-term DNF learning problem. Instead of searching through the space of all possible formulae, we only search for least general solutions.

To achieve this, we will show that a least general solution for a given learning problem is uniquely identified by a partitioning P of Pos into k pairwise disjoint subsets. Formally:

Definition 3.3.1 *Let Pos be a set and k be a natural number. Then a* (complete) k-partitioning P *of Pos is a set with*

1. $|P| = k$
2. *For all $Pos' \in P$: $Pos' \subset Pos$ and $Pos' \neq \emptyset$.*
3. *For all $Pos_1, Pos_2 \in P$ with $Pos_1 \neq Pos_2$: $Pos_1 \cap Pos_2 = \emptyset$.*

An incomplete partitioning *of Pos then is a partitioning of Pos with one or more elements of Pos removed from the subsets. If we remove all elements, we get the* empty partitioning.

If a given learning problem has a solution F, each of the k terms of F covers a subset Pos_i of Pos. The $lgs(F) = \bigvee_{i=1}^{k} lgg(Pos_i)$ is then a solution as well. If the problem setting has only one solution F, then $F = lgs(F)$. Note that some positive examples might be covered by two or more terms in F. Let p be such a positive example: $p \in Pos_i$ and $p \in Pos_j$ for some $i \neq j$. We can easily remove p from all but one Pos_i. If we do so for all p, that are covered twice, the resulting $P =_{def} \{Pos_i \mid 1 \leq i \leq k\}$ is a partitioning of Pos. $\bigvee_{Pos_i \in P} lgg(Pos_i)$ then still is a solution of the problem setting, because all positive examples are still covered and no new instances (i.e. potential negative examples) got covered in the process.

This means that if and only if a problem setting has a solution, then there is a k-partitioning P of Pos so that $\bigvee_{Pos_i \in P} lgg(Pos_i)$ is a solution as well. In this way we can extend the definition of the least general solution in section 2.4 to partitionings:

Definition 3.3.2 *Let P be a partitioning over a set of truth assignments. Then:*

$$lgs(P) =_{def} \bigvee_{R_i \in P} lgg(R_i)$$

We can use this modified *lgs* to construct a complete algorithm that calculates all possible partitionings P of Pos into k pairwise disjoint subsets and then calculates the least general specializations $lgs(P)$. If the problem setting has a solution, we will find a P so that $lgs(P)$ is a solution as well. More precisely:

1. Recursively enumerate all possible partitionings of Pos into k pairwise disjoint subsets P_i.

2. In every recursion step, build the formula $F =_{def} \bigvee_{1 \leq i \leq k} lgg(P_i)$, which corresponds to the current (incomplete) partitioning.

3. Whenever F is satisfied by a negative example, backtrack, otherwise continue the recursion.

4. When the partitioning is complete and the resulting F does not cover any negative examples, F is a solution.

There are a couple of ways to recursively enumerate all possible partitionings. However, it is not obvious which kind of recursion is well suited for k-term DNF learning. We designed and implemented two ways of enumerating the partitionings and tested the performance of the resulting algorithms. The size of the partitioning space for a k-term DNF learning problem with n positive examples is the number of possible partitionings of Pos into k pairwise disjoint nonempty subsets. This is the Stirling number of the second kind $S(n,k) = \frac{1}{k!}\sum_{i=0}^{k-1}(-1)^i\binom{k}{i}(k-i)^n$ [Wei02]. For large n, $S(n,k)$ grows approximately exponentially to the base k. For most practical settings this is considerably lower than $3^{|Var|\cdot k}$, the size of the space of all k-term formulae. Additionally, one can prune the search whenever a negative example is covered during formula construction. Note, however, that searching the partitioning space is redundant: two or more partitionings of Pos might lead to the same formula. For that reason it might be more efficient to search the formula space in some (rare) settings.

3.3.2 Recursion over k

The first algorithm enumerates all possible partitionings by recursively adding new subsets to an initially empty partitioning. During each recursion step we have to add a subset that has not been used in one of the preceding recursions. The following order $\preceq$ on sets of examples can be used to ensure that no subset is used twice:

$$A \preceq B \iff |A| < |B| \text{ or } (|A| = |B| \text{ and } max(A \setminus B) \leq max(B \setminus A))$$

Algorithm 3 describes this.

Algorithm 3 A complete algorithm using recursion over the terms.

```
procedure SearchPartitioningSpace(k)
  pool ← set of all positive examples
  Search(empty formula, pool, 0, k)
end procedure

procedure Search(formula, pool, nrTerms, k)
  if nrTerms = k-1 then
    if lgg(pool) does not cover any negative examples then
      formula ∨ lgg(pool) is a solution
    end if
  else
    subset ← smallest subset of pool according to ⪯
    while subset ⪯ pool do
      subset ← next subset of pool according to ⪯
      if lgg(subset) does not cover any negative examples then
        Search(formula ∨ lgg(subset), pool \ subset, nrTerms+1, k)
      end if
    end while
  end if
end procedure
```

3.3.3 Recursion over $|Pos|$

The second approach enumerates all possible partitionings by recursively adding positive examples to an initially empty partitioning. Here is a short example of how this algorithm works: consider the learning problem with three variables $Var = \{v_1, v_2, v_3\}$, three positive examples $Pos = \{001, 011, 100\}$, two negative examples $Neg = \{101, 111\}$ and $k = 2$. The algorithm will start with the empty partitioning $\{\emptyset, \emptyset\}$ and recursively add the positive examples, thereby calculating the formula $F = lgg(P_1) \vee lgg(P_2)$. As soon as the algorithm reaches the partitioning $\{\{001, 100\}, \{011\}\}$, F will be $\neg v_2 \vee (\neg v_1 \wedge v_2 \wedge v_3)$ and will cover the first negative example. Thus, the algorithm backtracks. However, when generating the partitioning $\{\{001, 011\}, \{100\}\}$, F is $(\neg v_1 \wedge v_3) \vee (v_1 \wedge \neg v_2 \wedge \neg v_3)$, which is consistent with Neg. The algorithm outputs F as a solution. Note that the terms of F are always satisfied by the positive examples in the corresponding subsets of the partitioning. We sketch this in algorithm 4.

The algorithm stores the available positive examples that have not yet been assigned to a partition in the variable *pool*. Obviously, the algorithm terminates, because each recursive call decreases $|pool|$ or it decreases $|pool|$ and k. Thus, the recursion terminates whenever either $|pool| = k$ or $k = 1$.

The correctness of the algorithm is less obvious. Note that this algorithm does not explicitely represent the calculated partitionings. Instead, it just calculates formulae which are the *lgs* of some incomplete partitionings. In the following we will prove that this algorithm does indeed implicitly calculate all possible k-partitionings of Pos (and thus all least general formulae). Since each generated formula corresponds to exactly one incomplete partitioning, we will

Algorithm 4 A complete algorithm using recursion over the positive examples.

```
1: procedure SearchPartitioningSpace(k)
2:    formula ← ⋁_{i=1}^{k} 1
3:    pool ← set of all positive examples
4:    Search(formula, pool, k)
5: end procedure
6:
7: procedure Search(formula, pool, k)
8:    if k = 1 then
9:       replace the first term t in formula with lgg(t, pool)
10:      if formula does not cover any negative examples then
11:         formula is a solution
12:      end if
13:   else if k = |pool| then
14:      replace each term t_i in formula with lgg(t_i, i-th element of pool)
15:      if formula does not cover any negative examples then
16:         formula is a solution
17:      end if
18:   else
19:      pos ← first element from pool
20:      for the first k terms t in formula do
21:         newFormula ← formula with t replaced by lgg(t, pos)
22:         if newFormula does not cover any negative examples then
23:            Search(newFormula, pool \ {pos}, k)
24:         end if
25:      end for
26:      newFormula ← formula with the k-th term t replaced by lgg(t, pos)
27:      if newFormula does not cover any negative examples then
28:         Search(newFormula, pool \ {pos}, k − 1)
29:      end if
30:   end if
31: end procedure
```

not discriminate between formulae and incomplete partitionings. Replacing a term t_i of a generated formula F by $lgg(t_i, p)$ for a positive example p is the same as adding p to the i-th partition of the partitioning, whose lgs is F. Using this analogy one can say that the algorithm calculates all possible partitionings by recursively adding positive examples to an initially empty partitioning. We show that a call to Search($formula$, $pool$, k) for any k and any $pool \subseteq Pos$ computes all possible complete k-partitionings of $pool$. From that follows that Search(empty partitioning, Pos, k) calculates all possible k-partitionings of Pos. The proof is an induction over $|pool|$ and k.

We start with the two base cases:

1. If $k = 1$, the desired partitioning has only one partition. Thus there is only one possible partitioning $P = \{pool\}$ (line 9).
2. If $k = |pool|$, We have to assign each positive example to one partition in order to get a complete partitioning. Thus, the only possible partitioning is $P = \{\{p\}|p \in pool\}$ (line 13).

Our inductive hypothesis is: assume that Search($formula$, $pool$, k) calculates all possible complete k-partitionings of $pool$ and Search($formula$, $pool$, $k-1$) calculates all possible complete $(k-1)$-partitionings of $pool$. Then Search($formula$, $pool \cup \{p\}$, k) computes all possible complete k-partitionings of $pool \cup \{p\}$.

Let A be the set of all possible k-partitionings of $pool \cup \{p\}$. Now let B be the set that contains all partitionings where p is the only example in its partition. Consequently $C =_{def} A \setminus B$ is the set of partitionings so that p is not the only example in its partition. We will examine the partitionings in B and C separately:

- Let P_B be a partitioning from B. If we remove p from P_B, the resulting partitioning contains $\emptyset$ as an element. Thus, $P_B \setminus \emptyset$ is a complete $(k-1)$-partitioning of $pool$. Because of the inductive hypothesis we can easily compute all partitionings in B by calling Search($formula$, $pool$, $k-1$) and adding $\{p\}$ to the resulting partitionings. This is done in line 26 of the algorithm.
- Let P_C be a partitioning from C. Then let P'_C be P_C with p removed. Clearly, P'_C is a complete k-partitioning of $pool$. Because of the inductive hypothesis we can simply call Search($formula$, $pool$, k) to find all possible P'_C. However, given a particular P'_C we can add p to any of the k partitions. Thus, the algorithm calls Search() k times, once for each partition (line 21).

Since Search() correctly calculates the partitionings in $A = B \cup C$, the inductive hypothesis holds.

3.3.4 Algorithm Performance

We compared the performance of the two algorithms on three test sets. The test sets were generated by setting the variables in the examples with the same probability to 0 or 1. For the first test set we chose $|Pos| = 10$, $|Neg| = 10$, $n = 10$, for the second $|Pos| = 15$, $|Neg| = 15$, $n = 20$, and for the third one $|Pos| = 20$, $|Neg| = 20$, $n = 42$. All of the 100 problem

Test Set	Recursion Over k	Recursion Over $\|Pos\|$
1	1.15 s	0.37 s
2	3 min	0.75 s
3	> 6 h	15 s

Table 3.1: The performance of the two complete algorithms.

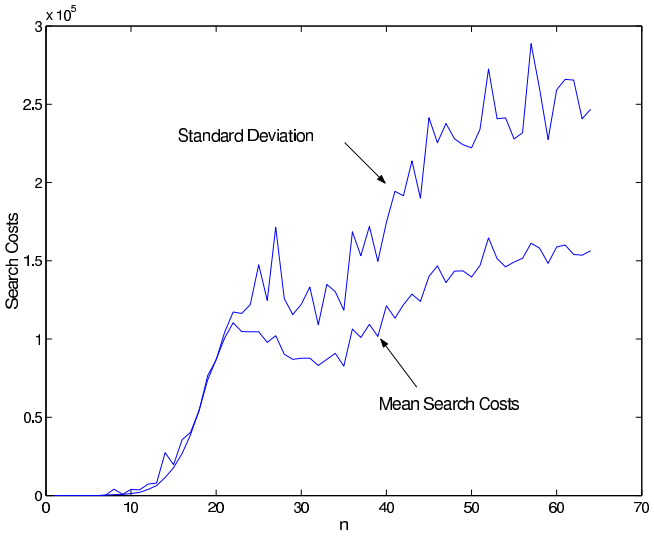

Figure 3.1: The mean and standard deviation of the search costs for problem settings with with $|Pos| = 16$, $|Neg| = 16$

instances in the test sets were soluble for $k = 3$. Note that the problems were taken from the phase transition region (see chapter 4 for details), so we examined only hard problem instances. The tests were performed on a 1GHz Pentium III class computer. Table 3.1 shows the results. Obviously, the second algorithm outperforms the first one by far. This might be due to the fact that the recursion over $|Pos|$ can prune the search earlier and more effectively. In the following we will use the second algorithm as a reference.

3.4 Randomized Complete Algorithms

The actual performance of a complete search algorithm on a given problem instance depends very much on the search order of the algorithm. Randomized algorithms can boost the performance on some problem instances by randomly modifying the search order. In the following we will present results that motivate the use of randomized complete algorithms. We will then sketch a randomized complete algorithm and evaluate its performance on selected problem instances.

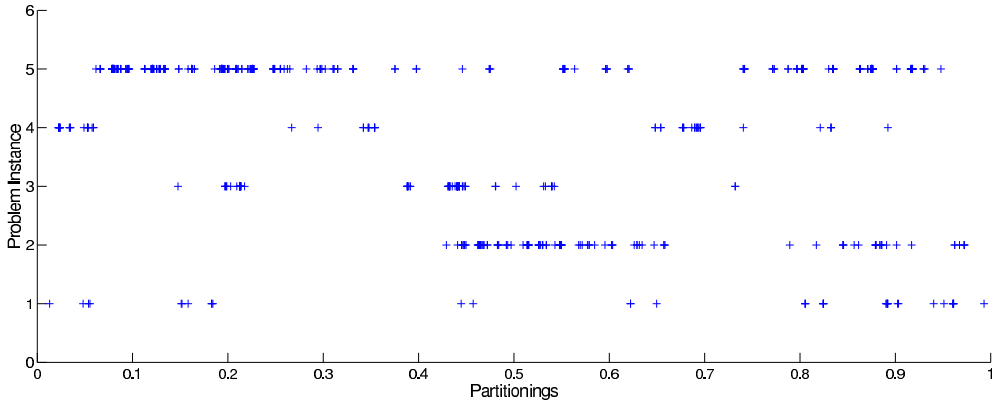

Figure 3.2: The location of solutions in the (normalized) partitioning space for five problem instances with $k = 3$, $|Pos| = 18$, $|Neg| = 40$ and $n = 55$

3.4.1 Randomized Algorithms

When calculating the mean search costs for problem instances one can make some interesting observations. In figure 3.1 we plot the mean and the standard deviation of the search costs for problem settings with increasing number of variables n. We counted the number of recursions of our second partitioning search algorithm as a measure for the search costs. Each data point was calculated by examining search costs for $k = 4$ of 1000 randomly generated problem instances with $|Pos| = 16$, $|Neg| = 16$. Obviously, the standard deviation of the search costs increases dramatically with increasing n.

As suggested in [GSCK00], this behaviour does not necessarily mean that some problem instances are "much harder" than others with the same parameters. Instead, it might just indicate that the search order of the given algorithm does not match very well with the distribution of solutions in the search space for those problem instances. According to this hypothesis, the search costs should be high, if the algorithm searches through solution-less parts of the search space first. If, on the other hand, a problem instance has solutions that are in an area first searched by the algorithm, then we should get low search costs. To examine this, we calculated the location of solutions in partitioning space according to our algorithm's search order for five randomly selected problem instances with $k = 3$, $|Pos| = 18$, $|Neg| = 40$ and $n = 55$. In figure 3.2 we plot the (normalized) search space along the x-axis and the five (arbitrarily ordered) examples along the y-axis. Each cross indicates a partitioning, whose lgs is a solution. For each problem instance the algorithm searches the search space from left to right. As can be seen, the solutions are not spread evenly across the whole space, but clustered in some small areas. For example, in the second problem instance, the algorithm has to search over 40% of the search space until it finds the first solution. If the algorithm would search in reverse order, it would find four solutions in the first 5% of the search space.

This means we might be able to boost the search by choosing a search order that is better suited to a given problem instance than the standard search order. One can easily modify the search order of our second algorithm by reordering the positive examples before starting the search. Unfortunately, we do not know in advance, which search order matches well with

a given problem instance. Gomes et al. proposes *random restarts* to "test" various search orders [GSK98]:

1. Start search with a random search order.

2. If the search does not find a solution after a threshold of t recursions, abort the search.

3. Increase t by some factor and repeat from 1.

Note that an algorithm with random restarts is still complete: as soon as the threshold t gets larger than the search space size, it will surely find a solution, if there is one. Whether or not random restarts decrease the overall search time depends on the change of search costs for different search orders. If the search costs do not vary too much from one search order to the other, random restarts are not expected to boost the search. If, however, there are a lot of search orders with low search cost and only a few search orders with exceptionally high search cost, then random restarts might – on average – decrease the overall search time.

3.4.2 Heavy-tailed Distributions

In [GSCK00] Gomes et al. report that for some NP-complete problems, the distribution of the search costs is *heavy-tailed*, i.e. the tails of those distributions have a power-law decay. That means that the probability of finding a problem instance with search costs higher than a threshold t does not decrease exponentially with increasing t (such as with the normal distribution), but only according to a power law: $P(X > t) \sim C \cdot t^{-\alpha}$ where C and α are unknown constants. Thus, there is a small but significant chance that we will find a problem instance with arbitrarily high search costs[1]. If α approaches 2, the variance of the search costs distribution diverges. Thus, for $\alpha \leq 2$, the distribution does not even have a finite variance. Because of the heavy-tailed distribution of search costs, random restarts can considerably reduce the average search time of the problems presented by Gomes.

We examined, how the search costs are distributed for various search orders in k-term DNF. Figure 3.1 indicates that the variance of the search costs grows with increasing n. Therefore we expect heavy-tailed distributions – if at all – for problems with a relatively high value of n. As we will see in chapter 4, those problem instances are in the "obviously soluble" region. To test this hypothesis we modified our second algorithm to use a different search order on each run. We then counted the search costs for 2000 different search orders on three problem instances with $k = 4$, $|Pos| = 24$, $|Neg| = 24$ and $n \in \{100, 200, 400\}$. Figure 3.3 shows $P(\text{Search costs} > t)$ plotted against t as linear plot and as a log-log plot. As can be seen, the graphs in the log-log plot can be approximated by linear functions. This indicates that $P(X > t)$ can in fact be approximated by a power law $C \cdot t^{-\alpha}$. The search costs are distributed according to a heavy-tailed distribution.

[1] As mentioned by Gomes et al., there is of course an upper bound determined by the size of the search space. This means, that in fact the heavy tails are "truncated" at some unknown high value. However, for many practical settings this upper bound is too large to be relevant for our discussion.

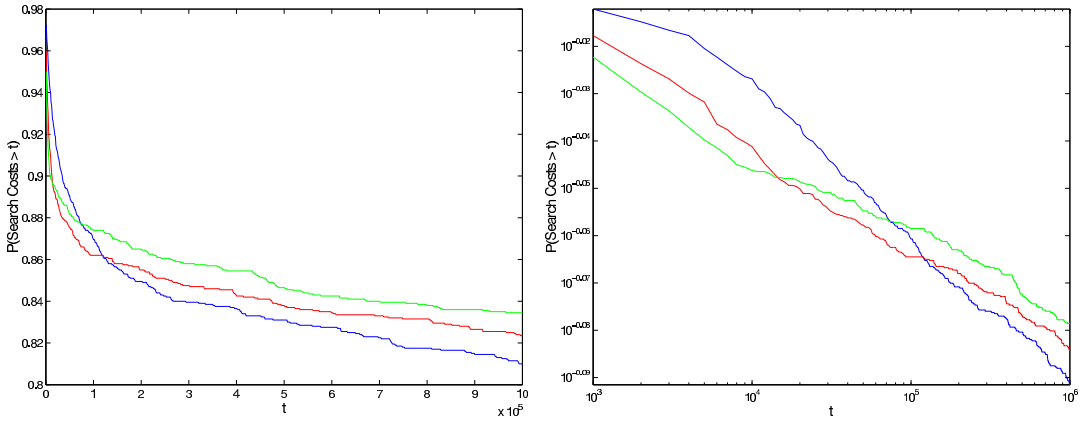

Figure 3.3: The distribution of search costs for three problem settings with $k = 4$, $|Pos| = 24$, $|Neg| = 24$ and $n \in \{100, 200, 400\}$ when varying the search order.

n	Algorithm	Mean Search Costs	Standard Deviation of Search Costs
100	Original	87.4 million	172.1 million
100	Randomized	4.6 million	15.5 million
200	Original	215 million	367 million
200	Randomized	6.6 million	18.6 million
400	Original	135 billion	268 billion
400	Randomized	2.8 million	9.4 million

Table 3.2: Search costs of the original and the randomized algorithm for search in partitioning space.

3.4.3 Algorithm Performance

The results in section 3.4.2 indicate that we might in fact be able to boost search time by using random restarts. For example, figure 3.3 shows that for the problem instance with $k = 4$, $|Pos| = 24$, $|Neg| = 24$ and $n = 400$, in 10% of all cases a solution was found in under 9000 recursions. However, in 83% of all tries the search costs were larger than 1 million recursions. In this situation it seems to be more efficient to test several search orders until we find a search order leading to a solution in less than 9000 recursions than to search according to a fixed search order, which might lead to a solution only after more than one million recursions.

To verify this hypothesis, we modified the algorithm in section 3.3.3 to use random restarts. We start with an initial cutoff threshold of 100 and double this value after each restart. It is clear that random restarts do not improve performance of searching insoluble problem instances. Therefore, we tested the randomized algorithm in the "obviously soluble" region ($k = 4$, $|Pos| = 24$, $|Neg| = 24$, $n \in \{100, 200, 400\}$). Table 3.2 shows the average search costs for the randomized and the original algorithm. The randomized algorithm found solutions in a fraction of the time used by the original algorithm. The performance gain is best for problems with $n = 400$.

Chapter 4

The Phase Transition

In this chapter, we introduce the Phase Transition Framework. We then show that k-term DNF learning has a phase transition and we locate the critical region. Also, we examine, if and how methods from statistical mechanics like finite-size scaling can be used to describe the k-term DNF phase transition.

4.1 The Phase Transition Framework

The Phase Transition framework has been successfully used in statistical mechanics to describe critical phenomena. We give a short overview and describe how the framework can be applied to combinatorial search.

4.1.1 History and Background

Phase transitions are very well known phenomena in physical systems. For example, when the temperature of water crosses the boiling point, its aggregate state changes rapidly from liquid to gaseous. Phase transitions occur in large systems, whose global behaviour is determined by the structural properties of the system's many parts of similar type. In the "boiling water" phase transition, the structural properties of the water molecules determine the state of the whole system. A special property of phase transitions in physical systems is that the change of a global *control parameter* (e.g. the temperature) at a *critical value* (e.g. 100 °C) can adequately describe the system's behaviour. In contrast, with chaotic systems like the convection in the lower atmosphere, some local changes can massively influence the behaviour of the global system. Thus, no global control parameter can describe the state of the whole system adequately. The field of statistical mechanics provides a rich set of methods to describe and evaluate critical phenomena and phase transitions.

Since the 1960s, researchers know about a behaviour in random graphs that resembles the phase transition in physical systems [PS96]. However, only in the early 1990s phase transitions were empirically discovered in combinatorial search problems. Especially the discovery of phase transitions in the NP-complete satisfiability (SAT) problem [KS94] has raised a lot of interest in the artificial intelligence community. Consequently, there has been a lot of research and up to now, phase transitions have been found in the Constraint Satisfaction

Problem (CSP) [SD96, CKT91], the Number Partitioning problem [GW96b], the Travelling Salesman problem (TSP) [GW96a], and many other NP-complete problems [CKT91]. In fact, up to now, there is no NP-complete problem, which has been proven to show no phase transition behaviour at all. From a theoretical point of view, phase transitions are not very well understood. There are some analytical results on the phase transition in SAT [PS96, Yug95], CSP [WH94] and some basic results on graph coloring [CKT92], but there is no comprehensive theory of phase transitions in combinatorial search problems.

4.1.2 The Phase Transition in Combinatorial Search

Many combinatorial search problems are NP-complete. NP-completeness is a worst-case complexity measure. In fact, most actual problem instances of NP-complete problems are easy to solve. For example, consider the decision version of the Travelling Salesman problem: we are given n cities, the distances d_{ij} between the cities and a maximum route length l. Now, we would like to know if there exists a route of length at most l through all n cities. If the value of l is very small (e.g. in the range of the smallest d_{ij}), then the problem is obviously insoluble. A combinatorial search algorithm will terminate quickly, because it can prune the search after a few tries. On the other hand, if l is very large, almost any route is a solution. Thus, the combinatorial search algorithm will quickly find a solution. Only if we choose l so that the problem is *critically constrained*, the algorithm will need huge computational resources to solve the problem. This transition between obviously soluble and obviously insoluble problem settings resembles the phase transition in physical systems: the change of a global *control parameter* (i.e. the maximum route length l) at a *critical value* (i.e. the value of l so that $P_{Sol} =_{def} P(\text{“problem is soluble”}) \approx 0.5$) adequately describes the system's behaviour (i.e. the average search costs).

It turned out that methods from statistical mechanics can be successfully applied to combinatorial search problems. For instance, simulated annealing [KGV83] is based on an analogy between annealing in solids and combinatorial optimization. The phase transitions in physical systems can be described by *finite size scaling*. According to this theory, the behaviour of a system around the critical point is equal for all configurations except for a change of scale. In the case of combinatorial search this can be expressed by the following equation:

$$P_{Sol} = f((\frac{\alpha - \alpha_c}{\alpha_c}) \cdot N^{\frac{1}{\nu}}) \tag{4.1}$$

In this equation α is the global control parameter, α_c is the value of α at the critical point and $N^{\frac{1}{\nu}}$ is the change of scale. The term $\frac{\alpha-\alpha_c}{\alpha_c}$ mimics the “reduced temperature” $\frac{T-T_c}{T_c}$ in physical systems. f is the behaviour of the system around the critical point. For most NP-complete problems f is the cumulative distribution function of a normal distribution (e.g. [GW96a, GW96b]).

Note that knowledge about the phase transition is especially important if we want to evaluate the performance of combinatorial search algorithms. It is easy to design algorithms that perform well on the easy problem instances outside the phase transition. In the past there have been publications presenting results only on easy problems (e.g. [Gu92]). Those results should be treated carefully when estimating the actual performance of the presented algorithms.

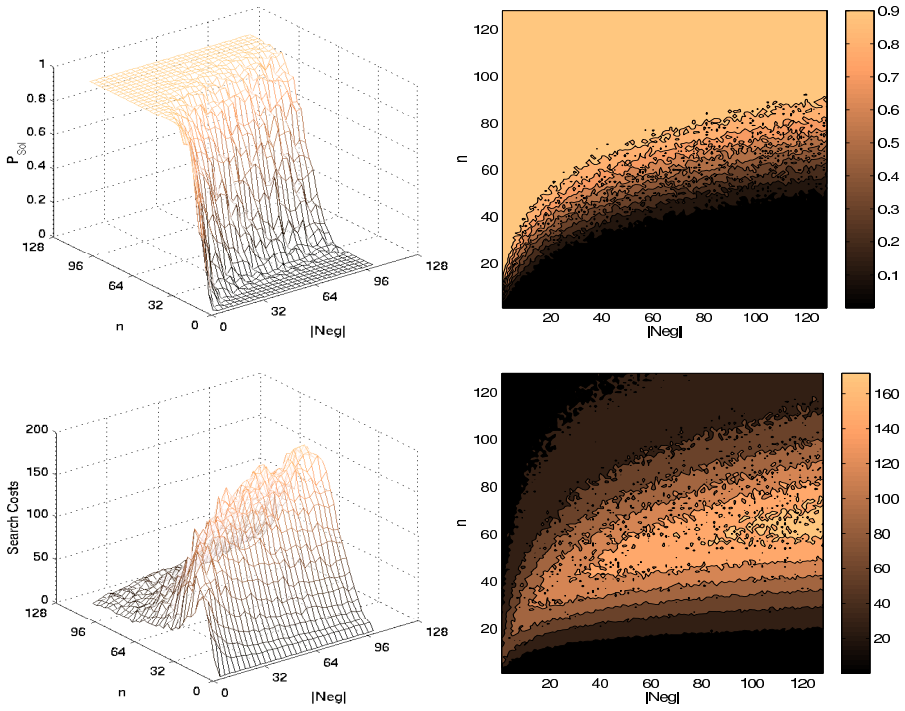

Figure 4.1: P_{Sol} (above) and search costs (below) plotted as 3D graph and contour plot for the problem settings with $k = 2$, $|Pos| = 10$, $1 \leq |Neg| \leq 128$, and $1 \leq n \leq 128$.

4.2 The Phase Transition in k-term DNF Learning

In order to estimate the performance of our complete and stochastic algorithms we need more information about the phase transition in k-term DNF learning. In the following we empirically show that there actually is a phase transition. We will then determine its location and use finite-size scaling to describe its shape.

4.2.1 The Phase Transition

To identify the location and size of the phase transition for k-term DNF learning, we examined the solubility and search costs for randomly generated problem instances. K-term DNF learning problem instances can be classified by four parameters: the number of Boolean variables n, the size of the set of positive examples $|Pos|$, the size of the set of negative examples $|Neg|$ and k, the maximal number of terms in the desired formula. We generated the positive and negative examples of the problem instances by choosing either $Var_i = 1$ or $Var_i = 0$ with the same probability for each variable $i, 1 \leq i \leq n$. The search costs were measured by counting the number of partitionings generated by the complete algorithm described in Section 3.3.3.

The search costs for finding a solution using the complete algorithm for such a randomly generated problem instance obviously depend on all four parameters. For instance, when keeping n, $|Pos|$, and k fixed and varying $|Neg|$, one would expect to have – on average – low

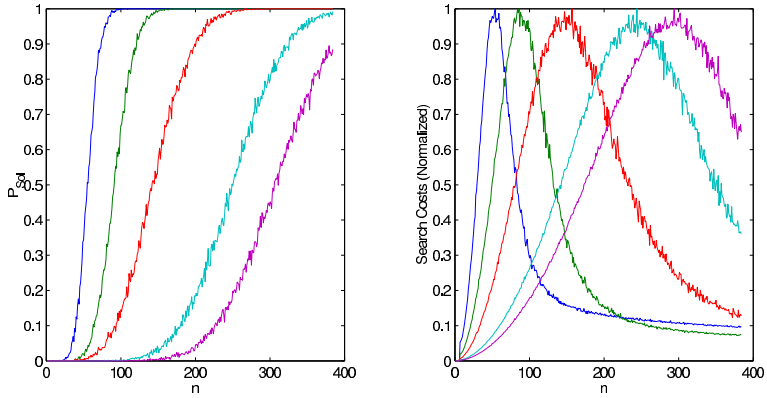

Figure 4.2: Some phase transitions for k=2, $|Pos| \in \{10, 12, 16\}, |Neg| \in \{20, 60, 100\}$

search costs for very low or very high $|Neg|$. With only a few negative examples, almost any formula covering Pos should be a solution, hence the search should terminate soon. For very large $|Neg|$, we can rarely generate formulae covering even a small subset of Pos without also covering one of the many negative examples. Consequently, we can prune the search early and search costs should be low, too. Only in the region between obviously soluble and obviously insoluble problem instances, the average search cost should be high. Similar considerations can be made about n, $|Pos|$, and k.

4.2.2 The Location of the Phase Transition

To examine this further, we calculated the probability P_{Sol} of a problem instance being soluble and the search costs for a broad range of problem settings. For instance, figure 4.1 shows the plots of these two quantities for fixed $|Pos| = 10$ and $k = 2$, and varying $|Neg|$ and n. Each data point represents the average over 100 problem instances. As expected, search costs are especially high in the region of $P_{Sol} \approx 0.5$. As described in section 4.1.2, we should be able to identify a "control parameter" α describing the location of the phase transition [GW96b, KS94], if the methods from statistical mechanics can be applied to k-term DNF learning. If finite-size scaling methods hold, we can express P_{Sol} using equation 4.1 around some critical point α_c.

We would now like to identify a suitable control parameter α for the k-term DNF learning problem. Figure 4.1 indicates that P_{Sol} changes rapidly as n increases. We examined this further: in figure 4.2 we plot P_{Sol} and the average search costs against n for some selected problem settings with $k = 2$, $|Pos| \in \{10, 12, 16\}, |Neg| \in \{20, 60, 100\}$. The figure shows that P_{Sol} in fact follows a curve that is similar to the cumulative distribution function of a normal distribution. The search costs are highest at the point $P_{Sol} \approx 0.5$. Thus, choosing n as the control parameter seems to be a reasonable idea. We (arbitrarily) choose the critical point n_c so that $P_{Sol}(n_c) = 0.5$. Unlike with some other NP-complete problems, in k-term DNF learning n_c is not simply a constant. Instead, its value depends on $|Pos|$, $|Neg|$, and k. We will now try to express n_c as a function of $|Pos|$, $|Neg|$, and k so that $P_{Sol}(n_c) = 0.5$.

First of all, note that there is an inherent asymmetry in the constraints imposed upon a term by the positive and negative examples: assume we have a term c containing l literals.

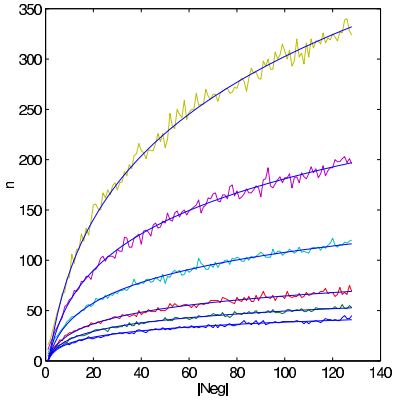

Figure 4.3: The location of n_c for $k = 2$ depending on $|Neg|$ for $|Pos| \in \{8, 9, 10, 12, 14, 16\}$.

Assume further that c is consistent with some positive examples $Pos_c \subseteq Pos$ and all negative examples in Neg. If we require the term to cover a newly added (random) positive example p, we have to replace c with $lgg(c, p)$. On average, we would expect the number of literals in $lgg(c, p)$ to be half the number of literals in c. Since a formula contains more than one term, we will expect that c needs to cover only "suitable" examples, so we expect the number of literals in c to decrease slightly slower than by factor 0.5. Still, the number of literals decreases exponentially with the number of covered positive examples. On the other hand, if we add a new negative example e, the term has to differ only in one literal in order to be consistent with the new negative example. If $l \geq 1$, c is already consistent with e with probability $1 - 0.5^l$. Only with probability 0.5^l we do have to add one new literal to c. Thus, the number of literals in c will increase considerably slower than the number of negative examples consistent with c.

This leads to two observations about n_c:

- **Observation 1:**
  n_c grows exponentially with the number of positive examples $|Pos|$. Assume, we found parameters n, $|Pos|$, $|Neg|$, and k so that $P_{Sol}(n, |Pos|, |Neg|, k) = 0.5$. If we add a new positive example e, a formula F has to additionally cover e in order to remain consistent with all positive examples. That means we have to replace at least one term c of F with $lgg(c, e)$, effectively reducing the number of literals in F by some unknown factor. Then, F more likely covers a negative example and this in turn decreases P_{Sol}. In order to keep P_{Sol} constant we have to increase n by a factor β, thus restoring the previous level of literals in c. Since formulae have more than one term, the size of the exponent is an (unknown) function γ, depending on $|Pos|$ and k. This yields:

$$n_c \approx \beta^{\gamma(|Pos|,k)} \tag{4.2}$$

- **Observation 2:**
  In fact, the value of β depends on the number of negative examples. Adding a new variable only increases P_{Sol}, if it increases the number of literals in F. The more

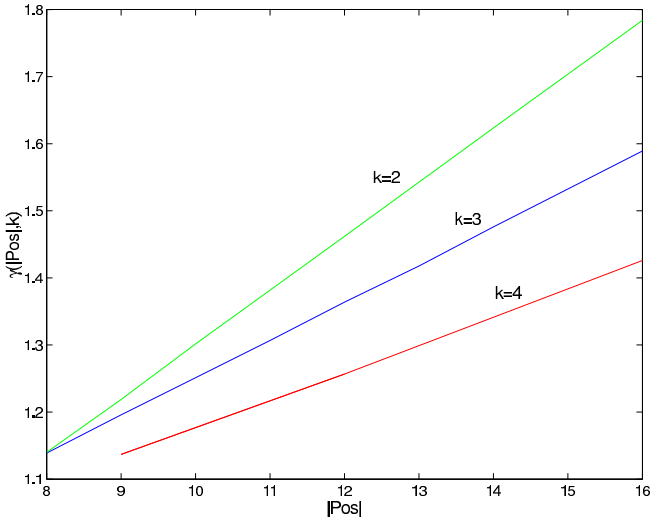

Figure 4.4: The estimated value of $\gamma(|Pos|, k)$ plotted against $|Pos|$ for $k \in \{2, 3, 4\}$

negative examples are present, the more variables we have to add on average until we can add a new literal to F without making it inconsistent. As indicated by Figure 4.1, n_c grows with $\log_2 |Neg|$. This seems to be reasonable given the fact that – on average – we need 2^l negative examples to increase the number of literals in term c by one. We would therefore expect that $n_c \propto a \cdot \log_2(|Neg|)$, with the factor a depending on k. Assuming $\beta = a \cdot \log_2 |Neg|$ as described above, we obtain:

$$n_c \approx (a \log_2 |Neg|)^{\gamma(|Pos|,k)} \tag{4.3}$$

a is the growth rate for fixed $|Pos|$ and variable $|Neg|$, while γ describes the growth rate of n_c with increasing $|Pos|$. To identify a and γ, we calculated P_{Sol} for a set of problem settings in the range of $2 \leq k \leq 5$, $1 \leq |Neg| \leq 120$, and $7 \leq |Pos| \leq 25$. From the resulting graphs, we could identify the location of n_c depending on $|Pos|$, $|Neg|$ and k. Figure 4.3 shows the measured values of n_c plotted against $|Neg|$ for $k = 3$ and $|Pos| \in \{9, 12, 15, 18\}$. n_c grows, as predicted, as a logarithmic function of $|Neg|$. We then examined, whether or not equation 4.3 does in fact describe the measured n_c. We used non-linear least square function regression (Nelder-Mead) to estimate the a and $\gamma(|Pos|, k)$, which fit to the measured curve best. The resulting curve fits are plotted in figure 4.3, too. As can be seen from this figure, equation 4.3 with the values of a and $\gamma(|Pos|, k)$ matches very well with the measured data.

We found the estimated a to be nearly constant for all problem settings with the same k. In figure 4.4 we plot the estimated values of $\gamma(|Pos|, k)$ for various k against $|Pos|$. Obviously, γ can be approximated very well by a linear function of $|Pos|$: $\gamma(x) =_{def} b \cdot x + c$. Table 4.1 shows the estimated values of a, b, and c for $2 \leq k \leq 5$.

Finally, these considerations lead us to our hypothesis about n_c.

$$n_c \approx (a \cdot \log_2 |Neg|)^{(b \cdot |Pos| + c)} \tag{4.4}$$

a seems to converge for larger values of k. Unfortunately, k-term DNF learning requires

Number of terms	a	b	c
2	3.6995	0.080602	0.49471
3	1.8072	0.056234	0.68868
4	1.4542	0.041363	0.76301
5	1.3927	0.026572	0.85334

Table 4.1: The values of a,b, and c for determining n_c depending on the number of terms k.

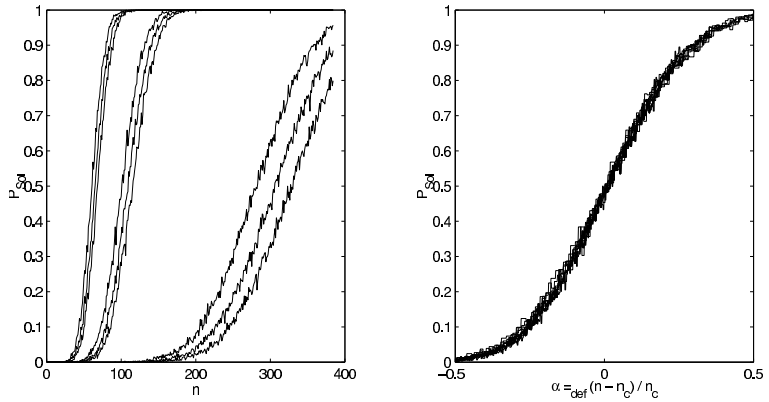

Figure 4.5: P_{Sol} for $k = 2$, $|Pos| \in \{10, 12, 16\}$, and $|Neg| \in \{80, 100, 120\}$, plotted against n and α.

huge computational resources for $k > 5$, so we could not examine this further. To verify the correctness of the approximation, we predicted n_c at $|Pos| = 30$, $|Neg| = 30$, and $k = 3$ to be about 178. We then computed 100 random problem instances and indeed found that $P_{Sol}(30, 30, 3, 178) = 0.51$, with an average search cost of 50 million recursions.

4.2.3 Finite Size Scaling

In order to put our hypothesized n_c to the test, we now check whether equation 4.1 adequately describes the phase transition. We computed P_{Sol} for $k = 3$, $|Pos| \in \{10, 12, 14, 16, 18\}$, and $|Neg| \in \{20, 40, 60, 80, 100, 120\}$. We varied n between 1 and 384 and solved 1000 randomly generated problem instances per parameter setting to determine P_{Sol}. Figure 4.5 shows P_{Sol} for some selected problem settings, plotted against n and $\alpha(n) =_{def} \frac{n-n_c}{n_c}$. The resulting graph mimics, as predicted, the cumulative distribution function of a normal distribution. As can be seen, the selected problem settings can be adequately described by α, even though we did not introduce a "change of scale" parameter $N^{1/\nu}$. It seems that the phase transition in k-term DNF learning does scale only with the size of n_c. Further investigations showed, that problem settings with the same $|Neg|$ are virtually indistinguishable when plotted against α. Only for small $|Neg|$, the slope of $P_{Sol}(\alpha)$ is slightly smaller than predicted. However, similar anomalies for small control parameter values are known for other NP-complete problems as well [GW96a, MSL92].

4.3 Conclusions

We showed that the k-term DNF learning problem has a phase transition. The phase transition's location can be estimated using equation 4.4 and its shape follows equation 4.4. One interesting property of this equation is that the actual size of the phase transition grows with n_c. Thus, the extent of the phase transition grows exponentially with $|Pos|$. For large values of $|Pos|$ relatively to k, huge parts of the problem space have high search costs.

The phase transition framework is a statistical tool. Therefore one has to be careful when evaluating real world problems using this framework. There are billions of different problem instances within each problem setting, many of which are very easy and many of which are very hard. The phase transition framework does only predict the average P_{Sol} and search costs of a problem setting. The P_{Sol} and search costs of an actual problem instance in this setting might differ greatly. In fact, we showed in section 3.4.2 that for some problem settings the variance of the search costs is remarkably high, so that the search costs differ strongly between the problem instances in a problem setting.

Chapter 5

Stochastic Local Search

In this chapter, we introduce Stochastic Local Search (SLS). We give a short overview over existing SLS algorithms for the SAT problem and present results on using these algorithms on SAT-encoded k-term DNF problems. We then design a range of SLS algorithms for the search in formula space and partitioning space and evaluate them using a test set of hard problems. Finally, we test our algorithms on a hard real-world problem.

5.1 Stochastic Local Search

Stochastic local search algorithms have been used since the early nineties to solve hard combinatorial search problems. We present some background information and introduce the GSAT and WalkSAT algorithms.

5.1.1 Background and History

In 1992 Selman and Gu independently from each other published a new method for solving hard satisfiability problems [SLM92, Gu92]. Their approach was remarkable in that a relatively small and simple algorithm could solve problems in a fraction of the time required by the most sophisticated complete algorithms. Though stochastic hill-climbing algorithms have been used before, Selman's results caused an explosion in research on the topic. Since then, various modifications and refinements of the original algorithm have been proposed and evaluated (e.g. [MSK97, SK93, SKC94, Seb94, Fra96, KMS97]). Those derivatives were dubbed "Local Random Search" algorithms, or, more precisely, "Stochastic Local Search" (SLS) algorithms. The main focus of the research was on SLS algorithms for the satisfiability (SAT) problem, though there are published SLS algorithms for the Constraint Satisfaction problem (CSP) problem [GBS99], and combinatorial auctions [HB00].

Technically, SLS algorithms are *Las Vegas algorithms*, i.e. nondeterministic algorithms that output a correct solution, if they terminate [Bab79]. Because of the non-determinism, the runtime of a Las Vegas algorithm is a random variable. Note, however, that we can modify any Las Vegas algorithm to stop after a certain runtime. This yields a *Monte Carlo algorithm*, i.e. a non-deterministic algorithm which might with a certain probability output a wrong result. SLS algorithms are characterized by the following properties:

- They are search algorithms, i.e. given a problem P they search through an instance space S_P of instances $i_P \in S_P$ for instances, that are solutions.
- They perform a local search. That means during their search they only consider instances, which are direct neighbours of the current instance according to a neighbourhood relation $R \subseteq S_P \times S_P$.
- The use of a global scoring function $score_P : S_P \times S_P \mapsto \mathbb{R}$. The decision on which instance should be examined next depends – at least partially – on the scoring function.

In the following we will present the most important SLS architectures and techniques. All presented algorithms work on the SAT problem.

5.1.2 The GSAT and WalkSAT Architectures

GSAT is the "classic" SLS algorithm for the SAT problem. It was introduced by Selman et al. in [SLM92]. Algorithm 5 sketches the algorithm idea. Basically, GSAT is a greedy hillclimbing procedure. It uses the number of unsatisfied clauses as global scoring function. Because of its greediness, GSAT gets easily stuck in local optima. Its only way to escape from local optima is to stop the local search (after *maxFlips* steps) and restart at a new random truth assignment. Because of this quite inefficient strategy GSAT is outperformed by almost all other algorithms that utilize more sophisticated ways to escape local optima.

Algorithm 5 The GSAT algorithm.

```
procedure GSAT(clauses, maxFlips, maxTries)
  for i ← 1 to maxTries do
    t ← a random generated truth assignment
    for j ← 1 to maxFlips do
      if t satisfies clauses then
        return t
      end if
      possFlips ← the set of propositional variables whose flip to the reverse value minimizes the number of unsatisfied clauses most
      p ← random variable from possFlips
      t ← t with the truth assignment of p reversed
    end for
  end for
  return "no satisfying assignment found"
end procedure
```

GSAT's inability to efficiently escape local optima can easily be avoided by adding some randomness or *noise* to the algorithm. This "GSAT with Random Walk" version was proposed first in [SK93]. The main idea is to take the greedy step (i.e. doing a flip which minimizes the number of unsatisfied clauses most) only with probability $1 - p$. With probability p pick a random variable from an unsatisfied clause and flip its value. The probability p is called *noise level*. Obviously, the optimal noise level depends on the structure of the problem instance. Research has shown that the performance of GSAT (and other random walk SLS algorithms)

depends very much on the right choice of the noise parameter. There have been plenty of other extensions to the basic GSAT architecture, some of which will be discussed later in this section.

WalkSAT was derived from GSAT with random walk. Algorithm 6 describes the main algorithm design. WalkSAT differs in three small, but important details from the GSAT based algorithms: first, it considers only variables, that occur in an unsatisfied clause. Second, its scoring function does not count the overall decrease of unsatisfied clauses. Instead, it counts the number of satisfied clauses which become unsatisfied. Third, if possible, WalkSAT takes a step that does not increase the score.

Algorithm 6 The WalkSAT algorithm.

```
procedure WalkSAT(clauses, maxFlips, maxTries, p)
  for i ← 1 to maxTries do
    t ← a random generated truth assignment
    for j ← 1 to maxFlips do
      if t satisfies clauses then
        return t
      end if
      unsat ← the set of all unsatisfied clauses
      c ← a random clause of unsat
      scores ← for each variable in c the number of clauses that are currently satisfied,
      but become unsatisfied, if the variable is flipped
      if min(scores) = 0 then
        v ← a random variable from c whose score is min(scores)
      else
        with probability p do
          v ← a random variable from c
        otherwise
          v ← a random variable from c whose score is min(scores)
        end with
      end if
      t ← t with v flipped
    end for
  end for
  return "no satisfying assignment found"
end procedure
```

A couple of extensions have been proposed to improve GSAT and WalkSAT-style SLS algorithms:

- *Weighting* [SK93, Fra96]: A weight value is assigned to each clause. The weights are regularly updated; the more frequently a clause is unsatisfied, the higher its weight. In the clause selection step, clauses with high weights are preferred.
- *Averaging* [SK93]: The most successful truth assignments are recorded. Instead of starting each try with a random truth assignment, the previously most successful assignments are randomly combined and used as a starting point for the search.

- *Tabu Search* [MSG97]: The algorithm keeps track of the t last flips in a tabu list. Variables in the tabu list are considered not at all or only with a lower probability for the next flip.

Of course, several of these strategies can be combined. As a result, more recent algorithms like Novelty [MSK97] choose the flip according to a sophisticated decision rule.

5.2 Using SLS on SAT-encoded k-term DNF Learning

An easy way to apply SLS algorithms to k-term DNF learning is to reduce a given k-term DNF learning problem to a satisfiability (SAT) problem and apply one of the many published SLS algorithms to the resulting SAT problem. In this way we can draw from a huge body of results from the area of satisfiability algorithms.

5.2.1 The Kamath et al. Test Set

In order to evaluate a particular SLS algorithm, we need an appropriate test set of problem instances. In [KKRR92] Kamath et al. introduced a reduction of k-term DNF learning to SAT. They generated a test set of 47 k-term DNF learning problem instances ranging from problems with eight variables up to 32 variables. The reduction of this test set to SAT has been widely used as a benchmark for SAT SLS algorithms [SKC94, SKC96, SLM92, SK93]. It is also part of the DIMACS Benchmark set for SAT [JT96]. Unfortunately, the test set seems to be very easy to solve. Kamath et al. used a target concept driven approach for constructing the problem instances. For each problem instance they built a random target formula. Then they uniformly generated random examples and labelled them according to the target formula. We reproduced the largest problem instances from the test set and found that our randomized complete algorithm solved all of them within a few seconds. Even worse, a propositional version of FOIL [Moo95] was able to solve them in less than a second. Obviously, the information gain heuristic works especially well for problem instances, which were generated by sampling over the uniform distribution. Clearly, this test set is too easy to be used as a hard benchmark for k-term DNF learning.

5.2.2 Results

In order to evaluate SLS algorithms on harder problem instances, we generated three test sets. As outlined in section 4.1.2 random problem instances from the wrong region of the problem space might be too easy to deliver meaningful results on the performance of an SLS algorithm. Therefore we generated problem instances only from the phase transition region of the problem setting space. We generated the positive and negative examples of the problem instances by choosing either $Var_i = 1$ or $Var_i = 0$ with the same probability for each variable $i, 1 \leq i \leq n$. Insoluble problem instances are not relevant for the evaluation of SLS algorithms, because any SLS algorithm will (correctly) return that no satisfying assignment was found. We therefore used our complete algorithm to remove insoluble instances from the three test sets. The resulting test sets contain one hundred soluble (for k=3) problem instances each. The first test set was generated with $|Pos| = 10$, $|Neg| = 10$, $n = 10$, the second one with

Algorithm	Noise Level[1]	Success Rate Test Set 1	Success Rate Test Set 2	Success Rate Test Set 3
GSAT	n/a	78.5%	0%	0%
GSAT+RandomWalk	0.25	87.2%	0%	0%
	0.5	89.3%	1.7%	0%
	0.75	56.8%	0.8%	0%
GSAT+Tabu	5	92.9%	0%	0%
	10	93.5%	0%	0%
	15	84.4%	0%	0%
WalkSAT	0.25	100%	97.5%	76.0%
	0.5	100%	98.2%	62.6%
	0.75	100%	90.5%	19.4%
Novelty	0.25	93.1%	2.8%	0%
	0.5	97.7%	4.2%	0%
	0.75	98.1%	6.7%	0%

Table 5.1: The success rates (i.e. fraction of tries that found a solution) for various SLS algorithms running on the reduced test sets.

$|Pos| = 20$, $|Neg| = 20$, $n = 42$ and the third one with $|Pos| = 30$, $|Neg| = 30$, $n = 180$. We reduced the test sets to SAT using the reduction outlined in section 2.5.

We tested GSAT, GSAT with Random Walk, GSAT with Tabu, WalkSAT, and Novelty on the SAT-encoded problems of the test sets. Each algorithm (except for GSAT) was tested with a noise parameter of 0.25, 0.5, and 0.75. We ran ten tries per problem instance and counted the number of successful tries (i.e. tries that found a solution) for each algorithm. For WalkSAT and Novelty each try was cut off after 100000 flips, for the GSAT based algorithms, we chose a cutoff value of 20 times the number of variables in the corresponding SAT problem. Table 5.1 shows the fraction of successful tries for each algorithm. On the hardest test set only WalkSAT yielded reasonable results. GSAT and its derivatives failed even on the second test set. Though WalkSAT and GSAT+RandomWalk are conceptually very similar, their results differ strongly (similar results have been found in [SKC94]). It seems that k-term DNF learning especially benefits from WalkSAT's bias towards steps, which do not break any currently satisfied clause. This behaviour ensures that the structural dependencies between the already satisfied clauses remain intact once they are found.

5.3 Using SLS on k-term DNF Learning

The SAT-encoding of a k-term DNF learning problem has $k \cdot (|Var| \cdot (|Pos|+1)+|Neg|)+|Pos|$ clauses. For practical problems, the number of examples might be very large and values of k in the range of 100 to 300 are common (e.g. [QCJ95]). Thus, the SAT-encodings of some k-term DNF learning problems might be too large to be used in practice. We are therefore interested in SLS algorithms which work directly on k-term DNF learning problems.

[1]For GSAT+Tabu we state the size of the tabu table instead of a noise level.

5.3.1 Motivation

When constructing an SLS algorithm, we have to make a couple of design decisions:

- Which search space should we use? As explained in section 3.1, we can choose between formula space and partitioning space for k-term DNF learning. The search space structure (much more than its size) determines the practicability of using SLS algorithms. Search spaces with many and deep local optima are usually less suitable, search spaces with only a few shallow local optima are better suited.
- Which neighbourhood relation and scoring function should we use? Usually, each search space has its "natural" neigbourhood relation and scoring function, but in some cases we might be able to choose between two or more possibilities. A good relation assigns not too many (or too few) neighbours to each instance and the neighbours of a given instance should be computable efficiently. Since the scoring function is called from the innermost loop, an efficient implementation is crucial. In the best case, we might be able to optimize the algorithm to store the scores in a lookup table. This can dramatically speed up the overall algorithm performance.
- Which algorithm strategies should we use? In particular, should we use WalkSAT or GSAT-style algorithms? How should we escape local optima? Does Averaging or Weighting add any benefit?

Unfortunately, there are no easy answers to these questions. As Hoos puts it: "To date, designing stochastic search algorithms and applying them successfully to hard combinatorial problems is as much an art or a craft as it is a science" [Hoo02]. An extensive framework on how each design decision might influence algorithm performance has not been developed yet (though [Hoo98] takes first steps in this direction). In the following, however, we will try to answer some of the mentioned questions for the k-term DNF learning problem.

5.3.2 GSAT-style Algorithms

We start with the most basic SLS algorithms: a GSAT-style search in partitioning and formula space with random walk. Note that the greedy version is the same as setting the noise parameter to zero in the version with random walk. In formula space a "natural" neigbourhood relation is $\{(F, G) \mid F, G$ are formulae $\wedge\ F$ and G differ only in one literal$\}$. According to this relation we have two possible flips per variable, because each variable can occur negated, unnegated, or not at all in a formula. Therefore we have to examine $2 \cdot n \cdot k$ neighbours per search step. An obvious choice for the scoring function is $score(S) = |\{x \in Pos \mid x \text{ violated by } S\} \cup \{x \in Neg \mid x \text{ satisfied by } S\}|$. Algorithm 7 sketches the design.

The partitioning space is a subset of the formula space containing only the least general formulae. A "natural" flip is to calculate the partitions $Cov_i(F)$ of the current formula F, then move a positive example from one partition to another partition and build the resulting formula. This rather complex neighbourhood relation assigns $(k-1) \cdot |Pos|$ neighbours to each instance. Since least general formulae always cover all positive examples, we simply use the number of covered negative examples as a scoring function. The pseudo code is given

Algorithm 7 A GSAT-like algorithm for stochastic local search in formula space.

```
procedure SLSearch(maxFlips, maxTries, p)
  for i ← 1 to maxTries do
    f ← a randomly generated formula
    for j ← 1 to maxFlips do
      if score(f) = 0 then
        return f
      end if
      with probability p do
        v ← a random variable
        flip ← a random kind of flip
      otherwise
        scores ← score(f with var flipped) for all variables var
        v ← a variable whose score is min(scores)
        flip ← the kind of flip of v that minimized the score
      end with
      f ← f with v flipped according to flip
    end for
  end for
  return "no formula found"
end procedure
```

Algorithm 8 A GSAT-like algorithm for stochastic local search in partitioning space.

```
procedure SLSearch(maxFlips, maxTries, p)
  for i ← 1 to maxTries do
    part ← a randomly generated partitioning
    for j ← 1 to maxFlips do
      if score(lgs(part)) = 0 then
        return lgs(part)
      end if
      with probability p do
        pos ← a random positive example
        target ← a random partition
      otherwise
        scores ← score(lgs(part')) for all partitionings part' that differ from part in the
        location of one example
        pos ← a positive example whose score is min(scores)
        target ← the target partition of pos so that the score is min(scores)
      end with
      part ← part with the positive example pos moved to partition target
    end for
  end for
  return "no formula found"
end procedure
```

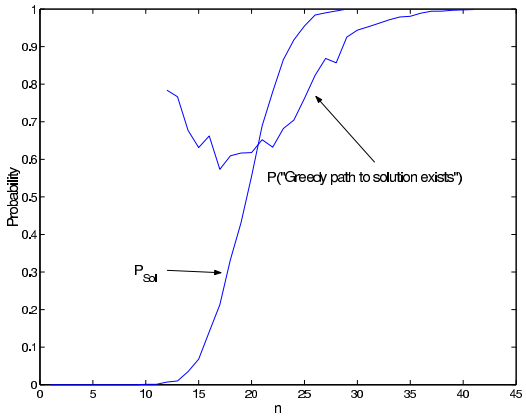

Figure 5.1: The fraction of partitionings that have a greedy path to a solution, plotted against n for $k = 3$, $|Pos| = 9$, and $|Neg| = 100$

in algorithm 8. We ran both algorithms on the test sets described in section 5.2.2. Table 5.2 shows the results for the formula space, table 5.3 the results for the partitioning space. We give the run time on a Pentium III 1GHz class computer and the success rate, i.e. the fraction of successful tries. The results indicate that greedy search is not efficient in both search spaces. In the partitioning space, having 25% noise increases the success rate up to 700% over the greedy version.

A short look on the test runs revealed that, in fact, many tries got stuck in local optima rather quickly. It seems that there are lots of local optima in both spaces. Since the partitioning space for problems with only a few positive examples is relatively small, we can examine this space in a more detailed way. A greedy GSAT-like algorithm starts its search at a random location in this space and repeatedly performs "greedy" steps, which minimize the score most. If there are two or more steps with the same minimal score, the algorithm chooses one at random. Thus, even if there is an order of steps which lead from the starting point to a solution, the algorithm might take a wrong step and end in a local optimum. For some starting locations, all possible greedy steps end in a local optimum and there are no greedy steps leading to a solution. We examined for how many starting points there is a "greedy path" from the starting point to a solution.

We generated soluble (for $k = 3$) problem instances with $|Pos| = 9$, $|Neg| = 100$, and $12 \leq n \leq 128$. The size of the resulting partitioning spaces is $S(9, 3) = 3025$. For each problem instance we searched through the whole search space and examined for each partitioning, whether or not there is a way to reach a solution by performing only greedy steps (i.e. steps that choose only flips with minimal score). In figure 5.1 we plot the fraction of partitions with a greedy path to a solution. Each data point represents the average over one hundred problems. We also plot P_{Sol} to identify the position of the phase transition.

The figure indicates that for problems in the phase transition region, on average only 57% of all partitionings have a greedy path to a solution. That means that 43% of the tries of a greedy algorithm have no chance of finding a solution. For the remaining 57% of tries there is a chance, but no guarantee that a solution is found. For problems outside the phase transition, the situation is much better: almost all partitions have a greedy path to a solution.

Test Set	Noise Level	MaxFlips	Run Time	Success Rate
1	0%	100	5.7 s	48.3%
	25%	100	4.7 s	41.6%
	50%	100	3.5 s	29.2%
	75%	100	2.1 s	13.7%
2	0%	200	151 s	1.6%
	25%	200	114 s	1.3%
	50%	200	76 s	0.9%
	75%	200	19 s	0%
3	0%	1000	192 min	0%
	25%	1000	144 min	0%
	50%	1000	96 min	0%
	75%	1000	48 min	0%

Table 5.2: The success rates and run times for a GSAT-like search in formula space.

As a result, greedy algorithms are suited, if at all, only for easy problems outside the phase transition. For non-greedy algorithms a good strategy for escaping local optima is crucial.

Another interesting observation is the fact that the number of neighbours in formula space depends on the number of variables n in the problem setting. This is unfortunate because for problems in the phase transition region n grows exponentially with $|Pos|$ (see section 4.2.2). The search in partitioning space considers only $(k-1) \cdot |Pos|$ possible flips and is thus much more efficient for large problem instances in the phase transition. A possible approach to overcome this limitation is to select the flipping candidates more carefully. We will examine such a strategy more closely in section 5.3.4.

5.3.3 Weighting and Averaging

In [SK93] Selman et al. propose weights and averaging as a method to boost the performance of GSAT. We empirically examined if those extensions affect the performance of our GSAT-like algorithms. We modified our search in partioning space to incorporate the two extensions:

- Weighting: initially we assign the weight 1 to every negative example. After each try we increment the weights of all negative examples that remained covered at the end of the try. When calculating the scores for the potential flips, we simply add up the weights of all covered negative examples. Thus, more frequently covered negative examples influence the score stronger than negative examples that remain unsatisfied most of the time. In this way we hope to adapt the search to the potential asymmetries in the problem instance.

- Averaging: during each try the algorithm records the best generated partitioning. After the first try we do not start with a completely random partitioning. Instead, we assign those positive examples to the same partition, which have been in the same partition in the best partitionings from the previous two tries. In this way some of the information gathered during the last two tries is reused.

Test Set	Noise Level	MaxFlips	Run Time	Success Rate
1	0%	100	0.9 s	87.6%
	25%	100	0.7 s	96.3%
	50%	100	0.7 s	94.1%
	75%	100	0.8 s	65.2%
2	0%	1000	80 s	30.2%
	25%	1000	40 s	72.9%
	50%	1000	30 s	70.1%
	75%	1000	28 s	13.8%
3	0%	5000	50 min	7.1%
	25%	5000	28 min	51.1%
	50%	5000	23 min	25.6%
	75%	5000	14 min	0.4%

Table 5.3: The success rates and run times for a GSAT-like search in partitioning space.

We tested the modified algorithm on the same test sets. Table 5.4 shows the results for the search with weights, table 5.5 shows the results for the search with averaging. In both cases, there was no significant speedup. Clearly, k-term DNF learning does not benefit from averaging and weights.

5.3.4 WalkSAT-style Algorithms

Given the good performance of WalkSAT on the SAT-encoded problem instances in section 5.2.2, one might assume that algorithms based on a WalkSAT-like architecture would outperform their GSAT-based counterparts in k-term DNF learning, too. In the following we will examine this hypothesis further.

In contrast to GSAT, WalkSAT only considers variables that are contained in a (randomly chosen) unsatisfied clause. In this way, WalkSAT focuses its efforts more on the remaining "troublemakers" than on unrelated variables, that may momentarily yield a better score when flipped. Thus, it utilizes the information contained in the unsatisfied clauses to guide its search. Unfortunately, the situation is more complicated in k-term DNF learning. When searching in partitioning space, we can determine, which negative examples are covered by which term of the current formula. However, we do not get any information on which positive examples need to be removed from the partition in order to make the covered negative example uncovered. Thus, we do not know, which positive examples are "troublemakers" and which ones are not. One has to rely on the GSAT-like approach of simply considering all possible flips. We therefore did not experiment with WalkSAT-like search in partitioning space.

The situation is better when searching through formula space. First of all, we can simply compare the number of uncovered positive examples and covered negative examples. If the number of uncovered positive examples is larger than the number of covered negative examples, the current formula is likely to be too specific. As a remedy we might want to generalize the formula by removing one or more literals. In the opposite case the formula is probably too general, so we should specialize the formula by adding literals. In fact we even get precise

Test Set	Noise Level	MaxFlips	Run Time	Success Rate
1	0%	100	0.8 s	90.7%
	25%	100	0.6 s	95.0%
	50%	100	0.7 s	93.7%
	75%	100	0.7 s	68.5%
2	0%	1000	85 s	25.9%
	25%	1000	40 s	73.0%
	50%	1000	30 s	70.5%
	75%	1000	28 s	14.6%
3	0%	5000	51min	5.7%
	25%	5000	27 min	51.9%
	50%	5000	23 min	26.4%
	75%	5000	14 min	0.4%

Table 5.4: The success rates and run times for a GSAT-like search in partitioning space with weights.

Test Set	Noise Level	MaxFlips	Run Time	Success Rate
1	0%	100	0.7 s	89.1%
	25%	100	0.7 s	97.2%
	50%	100	0.6 s	95.5%
	75%	100	0.7 s	70.0%
2	0%	1000	79 s	30.4%
	25%	1000	34 s	75.0%
	50%	1000	28 s	70.0%
	75%	1000	29 s	15.8%
3	0%	5000	49 min	9.3%
	25%	5000	26 min	54.0%
	50%	5000	23 min	25.7%
	75%	5000	14 min	0.7%

Table 5.5: The success rates and run times for a GSAT-like search in partitioning space with averaging.

information on how to fix the misclassification of an example. If we have an uncovered positive example p, we can replace one term t in the formula with $lgg(t, p)$ to fix the example. If we have a covered negative example, we just have to add or modify one suitable literal in each covering term. Thus, we have exact information on why the current formula fails and how we can fix the problems. These considerations lead to algorithm 9. Note that the algorithm uses two noise parameters: one for the generalization step and one for the specialization step.

Algorithm 9 A WalkSAT-like algorithm for stochastic local search in formula space.

```
procedure SLSearch(maxFlips, maxTries, p_G, p_S)
  for i ← 1 to maxTries do
    f ← a randomly generated formula
    for j ← 1 to maxFlips do
      if score(f) = 0 then
        return f
      end if
      mis ← the set of all uncovered positive or covered negative examples
      ex ← a random example from mis
      if ex is a positive example then
        with probability p_G do
          t ← a random term from f
        otherwise
          t ← the term in f which minimizes score(f with t replaced by lgg(f, ex))
        end with
        f ← f with t replaced by lgg(f, ex))
      else if ex is a negative example then
        t ← a (randomly selected) term from f that covers ex
        with probability p_S do
          l ← a random literal m so that t ∧ m does not cover ex anymore
        otherwise
          l ← a random literal m so that score(t ∧ m) is minimal
        end with
        f ← f with l added to t
      end if
    end for
  end for
  return "no formula found"
end procedure
```

We ran the resulting algorithm on the three test sets. The experiments indicated that the algorithm's performance depends much more on the choice of the noise parameter than with the GSAT-based algorithms. Changing the noise parameter from 25% to 50% reduces the success rate by half with algorithm 8, but the success rate of the WalkSAT based algorithm with 50% noise is only 16% of the success rate with 25% noise. To estimate the optimal noise parameters, we ran the algorithm on the three test sets and varied the noise parameters in steps of 5% from 0% up to 30%. Table 5.6 shows the results. To allow for a better comparison, we also state the results of the GSAT-based search in partitioning space as a reference. The WalkSAT-based search performs good, but still slightly worse than the GSAT based search

Test Set	Noise Level	MaxFlips	GSAT-based Run Time	GSAT-based Success Rate	WalkSAT-based Run Time	WalkSAT-based Success Rate
1	0%	100	0.9 s	87.6%	0.9 s	44.8%
	5%	100	0.8 s	90.6%	0.8 s	58.2%
	10%	100	0.8 s	92.3%	0.9 s	65.8%
	15%	100	0.8 s	94.3%	0.9 s	68.5%
	20%	100	0.8 s	94.5%	0.8 s	71.2%
	25%	100	0.7 s	96.3%	0.8 s	70.3%
	30%	100	0.7 s	94.8%	0.8 s	69.9%
	50%	100	0.7 s	94.1%	0.7 s	54.6%
	75%	100	0.8 s	65.2%	0.7 s	20.0%
2	0%	1000	80 s	30.2%	55 s	15.4%
	5%	1000	71 s	39.6%	41 s	51.5%
	10%	1000	63 s	48.2%	37 s	59.7%
	15%	1000	51 s	60.9%	33 s	65.7%
	20%	1000	46 s	67.5%	31 s	67.6%
	25%	1000	40 s	72.9%	32 s	63.8%
	30%	1000	36 s	74.8%	31 s	61.2%
	50%	1000	30 s	70.1%	33 s	29.8%
	75%	1000	28 s	13.8%	21 s	0.8%
3	0%	5000	50 min	7.1%	43 min	7.8%
	5%	5000	45 min	15.1%	31 min	48.0%
	10%	5000	40 min	26.0%	28 min	62.0%
	15%	5000	34 min	37.3%	27 min	63.8%
	20%	5000	31 min	46.8%	27 min	61.6%
	25%	5000	28 min	51.1%	28 min	53.3%
	30%	5000	25 min	54.4%	29 min	45.8%
	50%	5000	23 min	25.6%	28 min	8.5%
	75%	5000	14 min	0.4%	15 min	0%

Table 5.6: The success rates and run times for a WalkSAT-like search in partitioning space.

in partitioning space on the first two test sets. On the third test set, though, it outperforms the GSAT based algorithm.

5.3.5 Optimization

The test sets we used in the subsequent sections have relatively few examples. In real-world problems, like the King-Rook-King endgame benchmark (see section 5.4), the number of provided examples can be tens of thousands and more. Additionally, the number of features per example might be higher than the number of variables in our randomly generated test cases. Since the scoring functions of SLS algorithms count (either directly or indirectly) the number of misclassified examples and the scoring function is calculated for each neighbour in each search step, there seems to be a huge computational performance problem. In fact, when applying algorithm 9 to the first King-Rook-King endgame problem (28057 examples),

we measure an average 3.6 steps per second on a Pentium III 1 GHz class computer. This is too slow to be used in practice.

The obvious bottleneck in SLS algorithms is the calculation of the scoring function. An easy way to avoid this bottleneck is to precalculate the scores for all neighbouring instances and store them in a table. Only if a flip is actually performed, the algorithm has to recalculate some selected parts of the table. This technique has been successfully applied to GSAT and WalkSAT: Kautz's highly optimized implementation of WalkSAT performs about 1500 steps per second on the same computer. We applied similar optimization techniques to algorithm 9. In its original formulation the algorithm is not very well suited to the mentioned optimization techniques. Therefore we had to introduce a couple of modifications:

- We used two scoring functions: the generalization score counts the number of uncovered negative examples which become covered during a generalization step. The specialization score counts the number of covered positive examples that become uncovered during the generalization of a literal. The algorithm saves the precomputed scores for each literal in each term in two lookup tables. Note that this scoring function resembles WalkSAT's scoring function; it considers less examples than GSAT's global scoring function.

- We do not consider "sideway" specialization (i.e. making a negated literal unnegated and vice versa) anymore. Sideway specialization is computationally demanding, because it can affect the coverage of all examples in the worst case. In contrast, adding or removing a literal to a term only affects those examples, which do not satisfy the added or removed literal. Therefore, the algorithm can skip the evaluation of all other examples. On startup the algorithm builds a lookup table on which example is covered by which literal. In each flip one can use this table to find all examples, which might get covered or uncovered and therefore influence the updated scores.

- The algorithm keeps track of how many terms are satisfied by each example. Only if a positive example satisfies exactly one term, a change of a literal in this term affects the scores. If a positive example is covered by two or more terms, changing one of the two terms does not change the score, because the positive example remains covered. Similarly, the algorithm keeps track of the number of literals which keep an example from being covered by a term. The specialization score only needs to be updated, if there is one literal left, which prevents the coverage of a negative example.

- The algorithm uses reverse-lookup lists to speed up adding or removing examples to or from the lists of uncovered positive and covered negative examples.

Note that on average this optimized version needs more steps to solve a problem than the original algorithm. However, this disadvantage is easily made up by the improved performance. Table 5.7 shows the average number of flips per second when solving a problem instance of the third test set on a Pentium III 1GHz class PC. The optimized version performs the 26 fold amount of flips per second than the original version. The simple structure of SAT allows for even more aggressive optimization; WalkSAT performs an additional 68% more flips per second than our direct algorithm. The SLS algorithms for search in partitioning space are not very well suited for those optimization techniques. Moving a positive example from one

Algorithm	Flips per Second
Original	2213
Optimized	58824
WalkSAT	99010

Table 5.7: The average number of flips performed by various algorithms on a problem instance of the third test set.

partition to another can in the worst case affect the coverage of all examples, so the algorithm has to examine all examples in each step.

5.4 The King-Rook-King Endgame Benchmark

In the previous sections we gave empirical evidence that SLS algorithms can be successfully applied to hard random k-term DNF learning problem instances. However, real world problems are not "random". Instead, they often feature an (unknown) inherent structure, which might hinder or prevent the application of SLS algorithms. In the following we will examine, whether or not SLS algorithms can efficiently deal with a structured real-world problem.

The domain of chess endgames provides an ideal testbed for k-term DNF learning, since here we deal with noise-free datasets with discrete attributes only, and we are more interested in compression than in predictivity. Finding a minimum theory for such endgame data was also a goal for previous research in this area [Bai94, QCJ95, NHH00] and is of continuing interest [Für02], but has not been tackled since then, partly due to the complexity of the task. We decided to especially examine the simplest chess endgame, King and Rook versus King. In his PhD thesis Bain [Bai94] studies the task of learning to predict the minimum number of moves required for a win by the Rook's side. This problem is available from the "Machine Learning Repository" and has been used as a benchmark, for instance in Quinlan's evaluation of FOIL [QCJ95].

The problem is stated as a repetitive application of a learning algorithm: from a database of all legal positions the algorithm learns first all positions won in zero moves. These positions are then removed from the database and the algorithm is subsequently used to learn positions won in one, two, three, etc. moves. To make the problem setting suitable for our algorithm, we had to express the given chess situation as a truth value assignment. We decided to extract the most basic information from the given chess situations: the positions of the chessmen, the distance between them and their position with regard to the two diagonals[2]. Since the values for position and distance for each of the two dimensions are in the range one to eight, we express positions and distances using $2 \cdot 3 \cdot 8 = 48$ Boolean variables.The (symmetrical) distance to the two diagonals for the three chessmen requires $3 \cdot 2 \cdot 8 = 48$ Boolean variables. Thus, we encode a chess situation in $48 + 48 + 48 = 142$ variables.

We ran the optimized version of our algorithm from section 5.3.5 on the encoded problem instances for the first six problems in the KRK endgame suite. Table 5.8 shows the results. As a comparision we also state Quinlan's results with FOIL and Bain's original results with

[2]This is important information because the kings can move and cover diagonally.

Moves	Positions	FOIL (Clauses)	GCWS (Clauses)	PFOIL (Terms)	SLS Algorithm (Terms)
zero	27	3	6	3	3
one	78	13	12	21	4
two	246	15	20	30	10
three	81	17	44	47	9
four	198	41	89	92	25
five	471	55	185	219	80

Table 5.8: The number of clauses/terms learned by FOIL, GCWS, PFOIL and the SLS algorithm from section 5.3.5.

GCWS [Bai94]. However, one has to be cautious when comparing the results directly: Bain and Quinlan used First-Order learners. Thus, the learned formulae are conjunctions of Horn clauses (i.e. a form of CNF), not DNF as with our algorithms. Additionally, the first-order learners can use any combination of the relations "not", "less than", and "difference" to extract the relevant parameters of the chess situations. In contrast to this our SLS algorithm can base its decision only on the 142 precomputed features. To compare our results with a more related algorithm we implemented the propositional version of FOIL named PFOIL [Moo95], and let it learn DNF formulae from the same 142 features than our SLS algorithm.

The results are encouraging: the algorithm found for all problem instances formulae which are considerably smaller than the ones learned by PFOIL. For almost all learned formulae the number of terms was smaller than the number of clauses in the CNF formulae learned by GCWS and FOIL[3]. Only with the sixth problem FOIL generated a more compact formula. We assume that this is due to the fact that FOIL can make use of the "less than" relation to rely its decision on minimal distances between the chessmen. This information is not available to our SLS algorithm.

5.5 Conclusions

In this chapter we examined if and how SLS algorithms can be used to learn DNF formulae with as few terms as possible. From our results there is some evidence that SLS algorithms can be successfully applied to hard random problems and structured real world problems. We also shed some light on the question on which algorithms or strategies should be used.

The first observation is that encoding a problem in SAT and using WalkSAT works surprisingly well. This supports the conjecture in [Hoo98] that using SLS on SAT-encoded problems is competitive with the search in a native search space. The only disadvantage of this approach is the large memory footprint of the generated SAT problems. While this might not be an issue for problems with small k and relatively few examples, it can be the prohibiting factor for large real-world problems.

[3]Comparing the number of clauses in a CNF formula with the number of terms in a DNF formula is a questionable practice. Unfortunately, to the best of our knowledge, there are no published results on learning DNF formulae from the KRK data. We therefore present this comparison as a cautious "indication of trend".

We also examined strategies like weighting and averaging, different scoring functions and search spaces and architectures (GSAT and WalkSAT). It seems that a WalkSAT-based search in formula space yields the best results. Though we can not give any clear recommendations for the design of successful SLS algorithms, the following observations might give some good "rules of thumb":

- Focusing on the misclassified examples to guide the search seems to be an important part of the success of the WalkSAT-based SLS algorithms.
- Greedy search is clearly not competitive. An SLS algorithm for k-term DNF learning needs some more sophisticated way to escape local optima.
- The application (and the applicability) of efficient optimization techniques can make the difference between failure and success of an SLS algorithm. It is more efficient to choose scoring function and neighbourhood relation in a way that the resulting algorithm can be optimized aggressively instead of using more sophisticated, but less optimizable strategies.

Chapter 6

Summary and Outlook

Phase transitions and SLS algorithms are lively fields of research. In this chapter we summarize our results and give some possible directions for further research.

6.1 Summary

In this thesis we studied the application of stochastic local search (SLS) algorithms on the k-term DNF learning problem. We discussed the relevance of the problem for propositional concept learning and proved that the problem is trivial for k being larger than certain upper bounds.

We then presented five complete algorithms for the problem. We evaluated those algorithms and found that a recursion over the positive examples in partitioning space yielded the best performance. For problems in the "obviously soluble" region, randomized algorithms can boost the search considerably, because the distribution of the search costs for varying search orders is heavy-tailed.

We examined the k-term DNF learning problem in the phase transition framework and found that there is indeed a phase transition. We empirically derived an equation which successfully predicts the location of the phase transition, and we showed that methods from statistical mechanics can be used to describe the shape of the transition.

Finally, we examined stochastic local search algorithms. Using the complete algorithms from chapter 3, we showed that the Kamath et al. test set is not suitable for evaluating SLS algorithms. Therefore we built our own test set, using hard problem instances from the phase transition region. With this test set we were able to examine a range of published SLS algorithms for the satisfiability problem. Our results indicate that WalkSAT was able to find solutions for the largest fraction of SAT encoded problem instances. We then presented and evaluated SLS algorithms for search in the two "native" search spaces of k-term DNF learning. On our test set a WalkSAT-like search in formula space yielded the best results. A speed-optimized version of this algorithm was able to find solutions to the king-rook-king chess endgame problem, which were considerably more compact than the solutions found by a propositional version of FOIL. This result hints at the potential applicability of the investigated methodology for practical problems.

6.2 Outlook

The contributions of this thesis were focused on its main aim: evaluating the application of SLS algorithms on hard k-term DNF learning problems. However, we touched a range of other interesting problems and revealed some other possible routes for further research:

- In chapter 3 we presented some complete algorithms. We evaluate different search styles, but we do not examine if and how heuristics or certain optimization techniques could be successfully applied to those algorithms. Also, we do not investigate the randomized algorithm in detail: the optimal value of the factor, which increases the size of the searched area after each try, seems to depend on the distribution of the search costs. A more detailed analysis of this distribution might boost the search further or even enable the accurate prediction of the average search costs for a given problem setting.

- We empirically investigated the phase transition for problem settings with $k \leq 5$. Due to the huge computational complexity, we could not examine the phase transition for problems with $k > 5$. Since empirical research is always restricted to observable problems sizes, only an elaborate analytical investigation of the phase transition in k-term DNF learning could shed some light on problems with very high k.

- The field of SLS algorithms is too large to be examined exhaustively in this thesis. For example, we did not examine SLS techniques like tabu lists [MSG97] or more sophisticated approaches to weighting [Fra96]. Our experiments give only an rough indication of the average performance of the presented algorithms. A more detailed discussion should give the actual distribution of the search costs depending on the noise parameter and the type of the problem instances [Hoo98], so that a more accurate prediction of the expected search costs for a given problem instance is possible. There are also some other methods like estimating the noise parameter during search [MSK97] or learning evaluation functions to boost the search [BM00], which could be examined in the k-term DNF learning setting.

- While we successfully applied SLS algorithms to a selected real-world problem, it is an open question, if and how SLS algorithms have to be modified in order to yield good results on a broad range of real-world problems. The presented (or similar) methods seem to be applicable whenever a compact summarization of data is desirable. This is the case for many tasks faced in the field of Knowledge Discovery and Data Mining.

We hope that the results in this thesis prove to be useful for future research on phase transitions and SLS algorithms.

Bibliography

[Bab79] Laszlo Babei. Monte carlo algorithms in graph isomorphing testing. Technical Report DMS 79-10, Universite de Montreal, 1979.

[Bai94] Michael Bain. *Learning Logical Exceptions in Chess*. PhD thesis, Department of Statistics and Modelling Science, University of Strathclyde, Scotland, 1994.

[BM00] J. Boyan and A. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112, 2000.

[CKT91] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[CKT92] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Computational complexity and phase transitions. In *Proceedings of the Workshop on Physics and Computation. Los Alamitos, Calif.*, pages 63–68, 1992.

[Dom99] Pedro Domingos. The role of occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4):409–425, 1999.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Fra96] Jeremy Frank. Weighting for godot: Learning heuristics for GSAT. In *AAAI/IAAI, Vol. 1*, pages 338–343, 1996.

[Für02] J. Fürnkranz. Personal Communication, April 2002, 2002.

[GBS99] A. Giordana, M. Botta, and L. Saitta. An Experimental Study Of Phase Transitions In Matching. In T. Dean, editor, *16th International Joint Conference On Artificial Intelligence (Ijcai99)*, pages 1198–1203. Morgan Kaufmann, 1999.

[GSCK00] Gomes, Selman, Crato, and Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *JAR: Journal of Automated Reasoning*, 24, 2000.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 431–437, Menlo Park, July 26–30 1998. AAAI Press.

[Gu92] Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin*, 3(1):8–12, 1992.

[GW96a] Ian P. Gent and T. Walsh. The TSP phase transition. *Artificial Intelligence*, 88:349–358, 1996.

[GW96b] Ian P. Gent and Toby Walsh. Phase transitions and annealed theories: Number partitioning as a case study. In *European Conference on Artificial Intelligence*, pages 170–174, 1996.

[HB00] Holger H. Hoos and Craig Boutilier. Solving combinatorial auctions using stochastic local search. In *AAAI/IAAI*, pages 22–29, 2000.

[Hoo98] Holger H. Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, Technische Universität Darmstadt, 1998.

[Hoo02] H. Hoos. Announcement for the course cpsc 532d - stochastic search algorithms (spring 2002), 2002.

[JT96] D. S. Johnson and M. A. Trick, editors. *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KKRR92] A. P. Kamath, N. K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende. A continuous approach to inductive inference. *Mathematical Programming*, 57:215–238, 1992.

[KMS97] H. Kautz, D. McAllester, and B. Selman. Exploiting the variable dependency in local search, 1997.

[KS94] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 27 1994.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts, 1994.

[LMSK63] J. D. C. Little, K. G. Murty, D. W. Sweeny, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Moo95] Raymond J. Mooney. Encouraging experimental results on learning CNF. *Machine Learning*, 19(1):79–92, 1995.

[MSG97] Bertrand Mazure, Lakhdar Saïs, and Éric Grégoire. Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 281–285, Menlo Park, July 27–31 1997. AAAI Press.

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence, Rhode Island, 1997.

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distribution of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.

[NHH00] E. V. Nalimov, G. McC. Haworth, and E. A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000.

[PS96] David M. Pennock and Quentin F. Stout. Exploiting a theory of phase transitions in three-satisfiability problems. In *AAAI/IAAI, Vol. 1*, pages 253–258, 1996.

[QCJ95] J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):287–312, 1995.

[SD96] Barbara M. Smith and Martin E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):155–181, 1996.

[Seb94] Roberto Sebastiani. Applying GSAT to non-clausal formulas (research note). *Journal of Artificial Intelligence Research*, 1:309–314, 1994.

[SK93] Bart Selman and Henry A. Kautz. Domain-independant extensions to GSAT : Solving large structured variables. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 290–295, 1993.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.

[SKC96] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Johnson and Trick [JT96], pages 521–532.

[SLM92] Bart Selman, Hector Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 459–465, 1992.

[Val84] L. G. Valiant. A theory of the learnable. In *ACM Symposium on Theory of Computing (STOC '84)*, pages 436–445, Baltimore, USA, April 1984. ACM Press.

[Web96] Geoffrey I. Webb. Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417, 1996.

[Wei02] E.W. Weisstein. Eric Weisstein's World of Mathematics, http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html, April 2002, 2002.

[WH94] C. P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70(1-2):73–117, 1994.

[Yug95] N. Yugami. Theoretical Analysis of Davis-Putnam Procedure and Propositional Satisfiability. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence, (JCAI-95)*, pages 282–288, 1995.