Phase Transitions and Stochastic Local Search in k-Term DNF Learning

Ulrich Rückert, Stefan Kramer, and Luc De Raedt

Machine Learning and Natural Language Lab, Institute of Computer Science
University of Freiburg
Georges-Köhler-Allee, Gebäude 079
D-79110 Freiburg i. Br.
Germany
{rueckert, skramer, deraedt}@informatik.uni-freiburg.de

Abstract. In the past decade, there has been a lot of interest in phase transitions within artificial intelligence, and more recently, in machine learning and inductive logic programming. We investigate phase transitions in learning k-term DNF boolean formulae, a practically relevant class of concepts. We do not only show that there exist phase transitions, but also characterize and locate these phase transitions using the parameters k, the number of positive and negative examples, and the number of boolean variables. Subsequently, we investigate stochastic local search (SLS) for k-term DNF learning. We compare several variants that first reduce k-term DNF to SAT and then apply well-known SLS algorithms, such as GSAT and WalkSAT. Our experiments indicate that WalkSAT is able to solve the largest fraction of hard problem instances.

1 Introduction

The study of phase transitions of NP-complete problems [15, 3, 6, 7] has become quite popular within many subfields of artificial intelligence in the past decade. However, phase transitions have not yet received a lot of attention within the field of machine learning. Indeed, so far, there are only a few results that concern inductive logic programming [11, 8, 9]. The existence of phase transitions in inductive logic programming is not surprising because inductive logic programming is known to be computationally expensive, especially due to the use of θ-subsumption tests [21], which are NP-complete.

In this paper, we study an important class of boolean formulae, i.e. k-term DNF, and show that phase transitions also occur in this propositional framework. The task in k-term DNF learning is to induce a DNF formula with at most k disjuncts (or terms) that covers all positive and none of the negative examples (this is the consistency requirement). Examples are boolean variable assignments. Learning k-term DNF is of practical relevance because one is often interested in finding the smallest set of rules that explains all the examples. This criterion is motivated by the principle of William of Ockham. Various practical machine learning systems employ the covering algorithm in the hope of finding small rule-sets. Moreover, k-term DNF has some interesting computational properties. It has polynomial sample complexity (in the PAC-learning sense) but the consistency problem is hard [14]. The polynomial sample complexity implies that only a polynomial number of examples is needed in order to converge with high probability to a good approximation of the concept. On the other hand, the computation of complete and consistent concepts (the consistency problem) cannot be done in polynomial time (unless $RP = NP$). The combination of these properties makes k-term DNF the ideal class of formulae to start an investigation of phase transitions in boolean learning.

The contributions of this paper are as follows. First, we show that phase transitions exist for learning k-term DNF. This result is not surprising because of the hardness of the consistency problem. Secondly, we locate the phase transitions that arise in k-term DNF. Thirdly, we introduce the use of stochastic local search methods for learning hard k-term DNF problems. Stochastic local search algorithms approximate the optimal solution at much lower computational costs. Well-known examples of stochastic local search algorithms for SAT include GSAT and WalkSAT. Finally, our experiments demonstrate that these stochastic local search algorithms are effective.

This paper is organized as follows: Section 2 introduces k-term DNF learning, and Section 3 identifies and localizes the phase transition in k-term DNF learning. Subsequently, Section 4 presents stochastic local search that is based on the reduction of k-term DNF learning to the satisfiability problem (SAT) and compares variants thereof on test sets of hard problem instances. Finally, Section 5 discusses further work, related work and concludes.

2 K-term DNF Learning

A k-term DNF formula is a disjunction of k terms, where each term is a conjunction of literals. E.g. $(a_1 \wedge \neg a_2 \wedge a_3) \vee (a_1 \wedge a_4 \wedge a_5)$ is a 2-term DNF with the terms $(a_1 \wedge \neg a_2 \wedge a_3)$ and $(a_1 \wedge a_4 \wedge a_5)$.

The k-term DNF learning problem can now be formalized as follows [14]:

Given

– a set of Boolean variables Var,
– a set Pos of truth value assignments $p_i : Var \rightarrow \{0, 1\}$,
– a set Neg of truth value assignments $n_i : Var \rightarrow \{0, 1\}$, and
– a natural number k,

Find a k-term DNF formula that is consistent with Pos and Neg, i.e. that evaluates to 1 ($true$) for all variable assignments in Pos and to 0 ($false$) for all variable assignments in Neg.

We can make a few observations about this problem. First, for $k = |Pos|$, we have a trivial solution F, where each term in F covers exactly one positive example. Obviously, we are only interested in problem instances with $1 \leq k < |Pos|$. Second, if we know a solution F for a given k_{min}, we can easily derive solutions for any $k > k_{min}$. That means we can safely weaken the condition “formula needs to have exactly k terms” to “formula needs to have at most k terms”.

Finally, assume that we have discovered a solution F for a given problem instance. Upon closer inspection we might discover 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 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: $Cov_i =_{def} \{p \in Pos|$ the ith term of F 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:

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

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

The *least general specialization* is then:

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

One can show that $lgs(F)$ is a solution if F is a solution. As a consequence, a problem instance P has a solution if and only if it has a solution, that is a *least general specialization* (Proof omitted).
We can leverage these considerations 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. More precisely:

1. Recursively enumerate all possible partitionings of Pos into k pairwise disjunct subsets P_i. This can be done by starting with an empty partitioning and adding one positive example per recursion step.
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 all positive examples have been added and the resulting F did not cover any negative examples, F is a solution.

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 $k = 2$. 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.

The size of this search space for a k-term DNF learning problem with n positive examples is the number of possible partitionings of Pos into k pairwise disjunct nonempty subsets. This is the Stirling number of the second kind $S(n, k) = \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n$ [28]. 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 k-term formula space in some settings, especially for low k and high $|Pos|$.

3 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 sketched in Section 2.

The search costs for finding a solution using the complete algorithm for such a randomly generated problem instance obviously depend on all of the four parameters. For instance, when keeping n, $|Pos|$, and k fixed and varying $|Neg|$, one would expect to have – on average – low 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 costs should be high. Similar considerations can be made about n and $|Pos|$, but it is not obvious, to which degree each parameter affects solubility and average search costs.

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 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 $P_{Sol} \approx 0.5$. If the methods from statistical mechanics can be applied to k-term DNF learning, we should be able to identify a "control parameter" α describing the location of the phase transition [6, 15]. If finite-size scaling methods hold, we should be able to express P_{Sol}

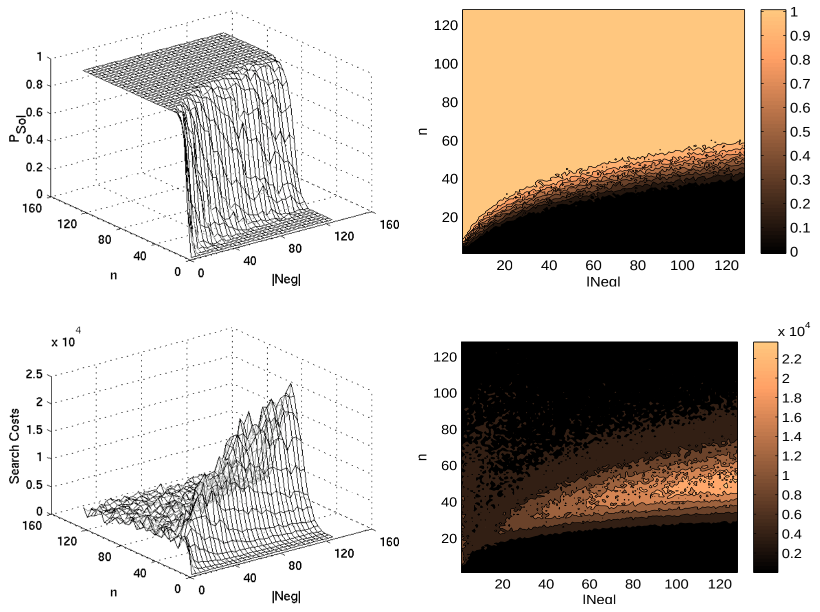

Fig. 1. P_{Sol} (above) and search costs (below) plotted as 3D graph and contour plot for the problem settings with $k = 3$, $|Pos| = 15$, $1 \leq |Neg| \leq 128$, and $1 \leq n \leq 128$

as a function of this control parameter around some critical point α_c [6, 2]:

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

The term $\frac{\alpha-\alpha_c}{\alpha_c}$ mimics the "reduced temperature" $\frac{T-T_c}{T_c}$ in physical systems, while the term $N^{\frac{1}{\nu}}$ provides the change of scale. As with many other NP-complete problems [7], we expect f to be the cumulative distribution function of a normal distribution.

Figure 2 shows that P_{Sol} increases rapidly with the number of variables n. 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. 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 new (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,

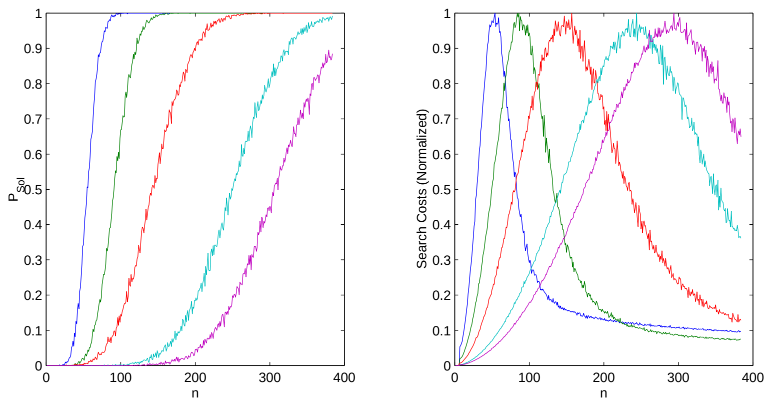

Fig. 2. P_{Sol} and search costs for problem settings with $k = 2$ and $(|Pos|, |Neg|)$ being – from left to right – (10, 100), (12, 60), (16, 20), (16, 60), and (16, 100)

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 in most cases. 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:
$$n_c \approx \beta^{\gamma(|Pos|,k)} \qquad (2)$$

– **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 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 1, n_c grows with $\log |Neg|$. 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:

$$n_c \approx (a \log_2 |Neg|)^{\gamma(|Pos|,k)} \tag{3}$$

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 n_c for a set of

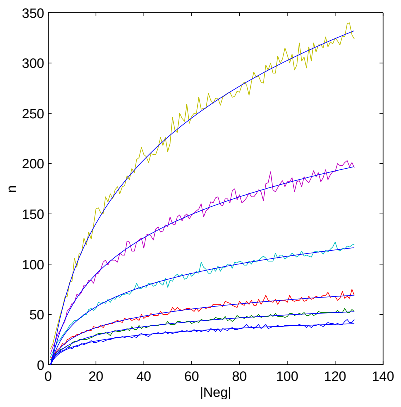

Fig. 3. The location of n_c for $k = 2$ depending on $|Neg|$ for $|Pos|$ being – from bottom to top – 8, 9, 10, 12, 14, and 16

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 estimated a and $\gamma(|Pos|, k)$ using non-linear least square function regression (Nelder-Mead). We found that γ can be approximated very well by a linear function of $|Pos|$: $\gamma(x) =_{def} b \cdot x + c$. Figure 3 shows the computed and the approximated value of n_c for k=3 and $|Pos| \in \{9, 12, 15, 18\}$. Table 1 shows the values of a, b, and c for $2 \leq k \leq 5$.

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

Hypothesis:

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

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

In order to put our hypothesized n_c to the test, we now check whether equation 1 adequately describes the phase transition. We computed P_{Sol} for $k = 3$, $|Pos| \in$

Table 1. The values of a,b, and c for determining n_c depending on the number of terms k

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

$\{10, 12, 14, 16, 18\}$, 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 shows P_{Sol} for some selected problem settings, plotted against n and $\alpha(n) =_{def} \frac{n-n_c}{n_c}$. 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 $N^{1/\nu}$. 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 [7, 18].

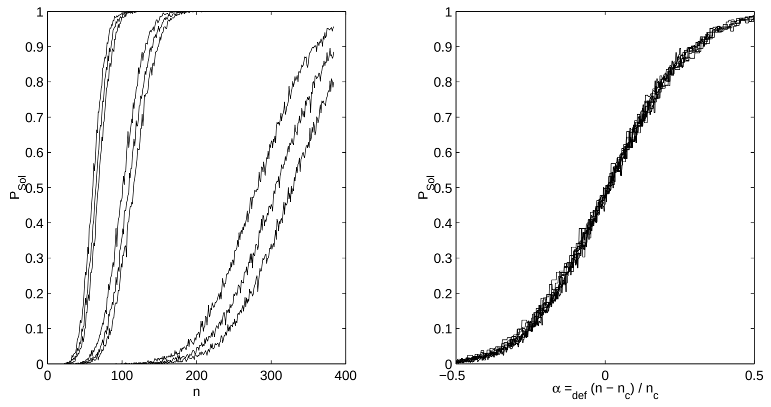

Fig. 4. P_{Sol} for $k = 2$ and $(|Pos|, |Neg|)$ being – from left to right – (10, 80), (10, 100), (10, 120), (12, 80), (12, 100), (12, 120), (16, 80), (16, 100), and (16, 120), plotted against n and α

4 Stochastic Local Search

Most NP-complete problems can be formulated as a search through the space of possible solutions. For the hard problem instances the size of the search space is extremely large, and, as a consequence, complete algorithms require huge computational resources. Thus, for most practical problems, we are gladly willing to sacrifice completeness (i.e. the certainty of finding a solution, if there is one) for adequate runtime behaviour. Though being incomplete, stochastic local search (SLS) algorithms have been shown to find solutions for many hard NP-complete problems in a fraction of the time required by the best complete algorithms. Since the introduction of GSAT [23], there has been a lot of research on SLS algorithms, and a couple of different variants have been proposed and evaluated [12, 24, 25, 17].

There are two main properties of SLS algorithms: first, instead of doing a systematic search through the whole instance space, an SLS algorithm starts a local search at a randomly selected location and restarts at different random locations, if it does not find a solution within a given time frame. Second, the search is local in a sense that it steps only to neighbouring instances. During its search an SLS algorithm usually favours those instances, that minimize some predefined global evaluation function. To be successful, an SLS algorithm needs to have some way of escaping or avoiding local optima. Often, this is achieved by performing randomized steps from time to time (the so called "noise"[1]).

An easy way to apply SLS algorithms to k-term DNF learning is to reduce a given

Table 2. The success rates (i.e. fraction of tries that found a solution) for various SLS algorithms running on the reduced test sets

Algorithm	Noise Level[2]	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%

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 [13] 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

[1] Note that this type of noise is different from the noise we use in machine learning!

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

for SAT SLS algorithms [26, 24, 23, 25]. 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 labeled them according to the target formula. We reproduced the largest problem instances from the test set and found that even our complete algorithm solved all of them within a few seconds. Even worse, a propositional version of FOIL [19] 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.

In order to evaluate SLS algorithms on harder problem instances, we generated three test sets, all taken from the phase transition region of the problem setting space. Each test set contains one hundred soluble (for k=3) problem instances. The first test set was generated with $|Pos| = 10$, $|Neg| = 10$, $n = 10$, the second one with $|Pos| = 20$, $|Neg| = 20$, $n = 42$ and the third one with $|Pos| = 30$, $|Neg| = 30$, $n = 180$. We reduced the test sets to SAT using the reduction from [13]. The resulting SAT problems describe the desired solution F using $2 \cdot |Var| \cdot k$ variables. They use another $|Pos| \cdot k$ auxiliary variables to express, which positive example is covered by which term. The constraints put on those variables by the positive and negative examples are encoded in $k \cdot (|Var| \cdot (|Pos| + 1) + |Neg|) + |Pos|$ clauses. We tested a range of known SLS algorithms on the SAT-encoded problems of the test sets (see [12] for a description of the algorithms). 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 2 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 [26]). 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 Conclusion

In the preceding sections we examined the NP-complete problem of k-term DNF learning. In Machine Learning we are not so much interested in the decision problem, but much more in the corresponding optimization problem: DNF minimization. As with many other NP-complete problems, an algorithm for the decision problem can be easily generalized to solving the optimization problem (usually by adding branch and bound techniques) and vice versa [7]. In fact, we found that the presented SLS algorithm was able to quickly find a solution for all $k' > k_{min}$. This is also supported by the results in Section 2, where we identified the location of the phase transition: if a problem instance is located in the phase transition for a given k, the corresponding decision problem instances for all $k' > k$ are in the "obviously soluble" region.

The DNF minimization problem settings seems to be at the core of most propositional concept learning settings, in the sense that:

1. Learning problems with discrete (and even with continuous-valued) 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.
2. Most propositional concept learners use representations that are subsets of or equivalent to DNF formulae, e.g. decision trees or disjunctive sets of rules.
3. Most concept learning algorithms include a bias towards a short representation of the hypotheses. While this might not necessarily increase the predictive accuracy, it is commonly considered as a desirable property [27, 4].

We showed that SLS algorithms can be successfully applied to hard randomly generated problem instances. Problem instances that are sampled from a uniform (or near uniform) distribution, can usually be solved in less than a few seconds by the presented SLS algorithm. The examples in the test cases were generated randomly; they do not follow a particular distribution. We would therefore expect that "real world" problems, which are obtained according to an (unknown) distribution are much easier to solve than the presented hard problem instances. However, it is not yet clear, whether or not SLS algorithms can efficiently deal with more structured problems. We are currently evaluating SLS algorithms for problem sets in the domain of chess endgames, such as the problem of predicting the minimum number of moves before a win on the Rook's side in the KRK (King-Rook-King) endgame. 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 [1, 22, 20] and is of continuing interest [5], but has not been tackled since then, partly due to the complexity of the task.

Another field of interest is the application of SLS algorithms in different learning settings. SLS algorithms can easily be adapted to tolerating noise by introducing some noise threshold for the score. Whenever the score of a formula falls below this threshold, the remaining uncovered examples are considered as noise.

Finally, we have to emphasize that stochastic search in general has been used before in Machine Learning (see, e.g., [10, 16]). However, to the best of our knowledge, this is the first attempt to introduce algorithms from stochastic local search (SLS) into propositional Machine Learning. Reducing the problem of k-term DNF to SAT, we can draw from a huge body of results from the area of satisfiability algorithms. In this sense, we hope that this work can stimulate further work along these lines in Machine Learning.

References

1. Bain, M.E. (1994) Learning logical exceptions in chess. PhD thesis. Department of Statistics and Modelling Science, University of Strathclyde, Scotland
2. Barber, M. N. (1983) Finite-size scaling. *Phase Transitions and critical phenomena*, Vol. 8, 145-266, Academic Press
3. Cheeseman, P., Kanefsky, B., and Taylor, W. M. (1991). Where the really hard problems are. *Proceedings of the 12th IJCAI*, 331-337

4. Domingos, P. (1999) The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, Vol. 3, Nr. 4, 409-425
5. Fürnkranz, J. (2002) Personal communication
6. Gent, I. P., and Walsh, T. (1995). The number partition phase transition. *Research report 95-185, Department of Computer Science, University of Strathclyde.*
7. Gent, I. P., and Walsh, T. (1996). The TSP phase transition, *Artificial Intelligence*, 88, 1-2, 349-358
8. Giordana, A., Saitta, L., Sebag, M., and Botta, M. (2000) Analyzing Relational Learning in the Phase Transition Framework. *Proc. 17th International Conf. on Machine Learning*, 311-318
9. Giordana, A., Saitta, L. (2000) Phase Transitions in Relational Learning. Machine Learning, 41(2), 217-25
10. Giordana, A., Saitta, L., and Zini, F. (1994) Learning disjunctive concepts by means of genetic algorithms. *Proc. 11th International Conf. on Machine Learning*, 96-104
11. Giordana, A., Botta, M., and Saitta, L. (1999) An experimental study of phase transitions in matching. *IJCAI 1999*, 1198-1203.
12. Hoos, H. H. (1998) Stochastic local search - methods, models, applications, PhD Thesis, Technische Universität Darmstadt
13. Kamath, A.P., Karmarkar, N.K., Ramakrishnan, K.G., and Resende, M.G.C. (1991). A continous approach to inductive inference. *Mathematical Programming*, 57, 1992, 215-238.
14. Kearns, M. J., and Vazirani, U.V. (1994). An introduction to computational learning theory. Cambridge, MA: MIT Press
15. Kirkpatrick, S., and Selman, B. (1994). Critical behavior in the satisfiability of random boolean expressions. *Science*, 264, 1297-1301
16. Kovacic, M. (1994) MILP – a stochastic approach to Inductive Logic Programming. *Proc. 4th International Workshop on Inductive Logic Programming*, 123–138
17. McAllester, D., Selman, B., and Kautz, H. (1997) Evidence for invariants in local search. *Proceedings of the 14th National Conference on Artificial Intelligence*, 321-326
18. Mitchell, D., Selman, B., and Levesque, H. (1992) Hard and easy distributions of SAT problems. *Proceedings of the 10th National Conference on Artificial Intelligence*, AAAI Press/ MIT Press, San Jose, CA, 459-465
19. Mooney, R. J. (1995) Encouraging experimental results on learning CNF. *Machine Learning*, Vol. 19, 1, 79-92
20. Nalimov, E.V., Haworth, G.McC., and Heinz, E.A. (2000) Space-efficient indexing of chess endgame tables. *ICGA Journal*, Vol. 23, Nr. 3, 148-162
21. Plotkin, G. D. (1970) A note on inductive generalization. Machine Intelligence 5, Edinburgh University Press, 153-163.
22. Quinlan, J.R., and Cameron-Jones, R.M. (1995) Induction of logic programs: FOIL and related systems. *New Generation Computing*, Vol. 13, 287-312
23. Selman, B., and Kautz, H. A. (1992) A new method for solving hard satisfiability problems. *Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, CA, 440-446
24. Selman, B., Kautz, H. A., and Cohen, B. (1993) Local search strategies for satisfiability testing. *Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, CA
25. Selman, B., and Kautz, H. A. (1993) Domain-independent extensions to GSAT: solving large structured satisfiability problems. *Proceedings of IJCAI 93*, 290-295
26. Selman, B., Kautz, H. A. and Cohen, B. (1994) Noise strategies for improving local search. *Proceedings of the 14th National Conference on Artificial Intelligence*, 337-343
27. Webb, G. J. (1996) Further evidence against the utility of Occam's razor. *Journal of Artificial Intelligence Research*, Vol. 4, 397-417
28. Weisstein, E. W. (2002) Stirling number of the second kind, http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html