Stochastic Local Search in k-term DNF Learning

Ulrich Rückert RUECKERT@INFORMATIK.UNI-FREIBURG.DE
Machine Learning and Natural Language Lab, Institute of Computer Science, University of Freiburg, Georges-Köhler-Allee, Gebäude 079, D-79110 Freiburg i. Br., Germany

Stefan Kramer KRAMER@IN.TUM.DE
Technische Universität München, Institut für Informatik/I12, Boltzmannstr. 3, D-85748 Garching b. München, Germany

Abstract

A novel native stochastic local search algorithm for solving k-term DNF problems is presented. It is evaluated on hard k-term DNF problems that lie on the phase transition and compared to the performance of GSAT and WalkSAT type algorithms on SAT encodings of k-term DNF problems. We also evaluate state-of-the-art separate and conquer algorithms on these problems. Finally, we demonstrate the practical relevance of our algorithm on a chess endgame database.

1. Introduction

This paper deals with one of the most fundamental abstractions of rule learning, namely k-term DNF learning. The task in k-term DNF learning is to induce a formula in disjunctive normal form of at most k disjuncts (terms) given positive and negative examples. This problem is at the core of DNF minimization, i.e. the problem of finding a DNF formula for a given dataset with as few terms as possible. The goal of finding small formulae is, in one or the other form, present in most practical rule learning algorithms, and related to the principle of William of Ockham. K-term DNF learning is known to be NP-complete.

K-term DNF learning is essentially a combinatorial search problem. Because of the combinatorial complexity, exhaustive search algorithms for finding solutions require huge computational resources for hard problem settings. In this paper we investigate other approaches with the goal to find solutions more efficiently. Clearly, there are some trade-offs involved: stochastic local search (SLS) algorithms are incomplete, i.e. they are not guaranteed to find a solution. Other approaches such as separate-and-conquer only find good approximations with a slightly larger number of terms than the optimal solution.

In previous work (Rückert et al., 2002) we laid the foundation for this investigation: first of all, we presented a complete exhaustive search algorithm, that can be used as a reference for other algorithms. We then used this algorithm to investigate the distribution of the search costs for randomly generated problem instances. We applied the *phase transition* framework to identify hard randomly generated problem instances, that can be used as a benchmark for the evaluation of the algorithms. Finally, we used a polynomial reduction of the k-term DNF learning problem to the satisfiability problem (SAT) to show that WalkSAT is able to solve a large fraction of the hard problems in the benchmark set. WalkSAT can be seen as a reference for native SLS algorithms, i.e. SLS algorithms that search directly in the solution space of the k-term DNF learning problem.

In this paper we continue our investigation: first of all we design a native SLS algorithm, that is able to efficiently solve hard k-term DNF learning problems without the need for encoding a k-term DNF learning problem as a SAT problem. We give a detailed analysis of the algorithm's performance compared to WalkSAT. In the same context, we evaluate the separate-and-conquer approach used in many rule-learning systems. More specifically, we investigate the average size of the learned rules for hard problems from the phase transition region in relation to the shortest possible formula. Finally, to show the practical relevance of the k-term DNF learning problem, we apply our SLS algorithm to a challenging chess-endgame problem.

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 (Kearns & Vazirani, 1994):

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

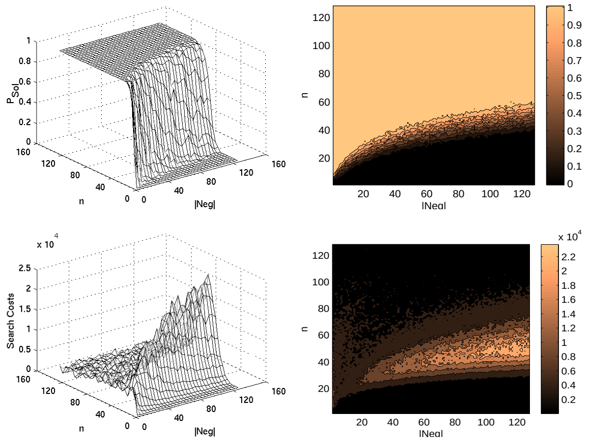

Figure 1. P_{Sol} (above) and search costs (below) plotted as 3D graph (left) and contour plot (right) for the problem settings with $k = 3$, $|Pos| = 15$, $1 \leq |Neg| \leq 128$, and $1 \leq n \leq 128$

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.

This problem is in the presented form a decision problem. Given the parameters, we would like to know, whether or not there exists a solution. The corresponding optimization problem is DNF minimization: given the positive and negative examples, we would like to find the smallest k, so that there is a formula consistent with the examples. Every algorithm for solving the k-term DNF learning problem can easily be extended to solving the DNF minimization problem (usually by adding branch and bound techniques). Thus, in the following we focus only on k-term DNF learning, even though the optimization version is much more common in practical applications.

K-term DNF learning is essentially a combinatorial search problem. Indeed, k-term DNF learning is NP-hard. There are two basic ways of solving a k-term DNF learning problem. One possibility is to search the space of all possible k-term DNF formulae directly. The other possibility is to search the usually smaller space of k-partitions of Pos. A k-partition is a partition of Pos into k disjunct subsets. Given any k-partition, one can obtain a k-term DNF formula that covers all positive examples by computing the k least general generalizations of the k partitions. The least general generalization of a set is the conjunction of all literals that are true in all the examples in the set. E.g., for the examples $e_1 = \{a_1 = 1, a_2 = 1, a_3 = 0\}$ and $e_2 = \{a_1 = 1, a_2 = 0, a_3 = 0\}$ the $lgg(e_1, e_2) = a_1 \wedge \neg a_3$, because the variable assignments to a_1 and a_3 in the examples can be combined to the literals a_1 and $\neg a_3$. The formula obtained in this manner covers all positive examples. If it does not cover any negative example, it is a solution. Furthermore, if there is no k partition whose generalization is a solution, then the k-term DNF problem does not have a solution at all. This leads to an elegant combinatorial search algorithm for solving k-term DNF learning problems, as outlined in (Rückert, 2002).

Due to the NP-hardness of the k-term DNF learning problem, exhaustive algorithms are impractical. Therefore, we are often willing to sacrifice completeness (i.e. the certainty of finding a solution, if there exists one) for better runtime behavior. In this context, *stochastic local search* (SLS) algorithms have been shown to find solutions to many hard NP-complete problems in a fraction of the time required by the best complete algorithms at the cost of incompleteness. Since the introduction of GSAT (Selman et al., 1992), there has been a lot of research on SLS algorithms, and a couple of different variants have been proposed and evaluated (Hoos, 1998; Selman & Kautz, 1993; Selman et al., 1996; McAllester et al., 1997). In this paper, we will study such stochastic local search algorithms for k-term DNF learning.

In order to evaluate SLS algorithms, we need a suitable test set of hard problem instances. If the test set contains only easy problem instances, all algorithms perform quite well. Unfortunately, large values of $|Pos|$, $|Neg|$, $n = |Var|$ and k do *not* necessarily indicate that a problem at hand is hard to solve. For example, even though the test sets used in (Kamath et al., 1992) have up to one thousand examples, they can be solved by a randomized version of the exhaustive search algorithm in partition space in less than a second (Rückert, 2002). On the other hand, some problem instances with only sixty examples can take weeks to solve. We therefore need some way to estimate the hardness of our benchmark data sets.

One way to assess problem hardness is the phase transition framework. Consider, for instance, k-term DNF learning problem instances where n, $|Pos|$, and k are fixed and $|Neg|$ is varied. 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, $|Pos|$, and k. This transition between obviously soluble and obviously insoluble problem settings resembles the phase transition in physical systems. It is, however, not obvious, which parameter influences the average hardness of the generated problem instances in which way. A first step towards the characterization of the phase transition in k-term DNF Learning has been made in (Rückert et al., 2002): Figure 1 shows $P_{Sol} =_{def} P$(“instance soluble”) and average search costs for randomly generated problem instances with $k = 3$, $|Pos| = 15$, $1 \leq |Neg| \leq 128$, and $1 \leq n \leq 128$. As can be seen, the problem instances whose average probability of being soluble is approximately 0.5 require the largest average search costs. This interrelation can be expressed using an equation which is derived according to methods from statistical mechanics. In the following we will make use of these results to evaluate Stochastic Local Search algorithms on hard problems taken from the phase transition region.

3. Stochastic Local Search

SLS algorithms differ from other approaches in that they perform a local, randomized-walk search. In its basic incarnation, an SLS algorithm starts with a randomly generated solution candidate. It then iterates in a two-step loop: in the first step it examines a fixed set of “neighboring” candidates according to some predefined neighborhood relation. Each neighbor is evaluated according to some global scoring function. In the second step the SLS algorithm selects the neighbor with the highest score as next candidate. Such a greedy hill climbing approach is obviously susceptible of getting caught in local optima. Most SLS algorithms therefore select with a fixed probability p (the so-called *noise probability*) a random neighbor instead of the neighbor with the highest score. In this way they can escape local optima through random steps. Algorithm 1 sketches the main concept.

Algorithm 1 A framework for typical Stochastic Local Search Algorithms.

```
procedure SLSearch($p$)
  $c \leftarrow$ a randomly generated candidate
  while $score(c) \neq \max$ do
    $n \leftarrow$ the set of all neighbors of $c$
    with probability $p$ do
      $c \leftarrow$ a random neighbor in $n$
    otherwise
      $s \leftarrow$ the scores of the neighbors in $n$
      $c \leftarrow$ the neighbor in $n$ with the highest score
  end while
  return $c$
end procedure
```

Technically, SLS algorithms are *Las Vegas algorithms*, i.e. nondeterministic algorithms that output a correct solution, if they terminate. Because of the non-determinism, the runtime of a Las Vegas algorithm is a random variable. Since in practice one is not willing to wait forever, SLS algorithms are usually implemented to stop after a certain maximum runtime (the so called *cutoff time*) and output “no solution found”. This yields a *Monte Carlo algorithm*, i.e. a non-deterministic algorithm which might with a certain probability output a wrong result. Sometimes the average runtime of an SLS algorithm on a specific set of problems can be improved by selecting a comparably low cutoff time and using frequent restarts (Gomes et al., 1998; Hoos, 1998).

When designing an SLS algorithm for k-term DNF Learning, one has to decide about a suitable candidate space first. Using the space of k-partitions, just as with the exhaustive search algorithm outlined in section 2 seems to be an obvious choice. Unfortunately, calculating the neighboring formulae for a given candidate in k-partition space is a relatively time consuming task, because it requires the computation of two lggs per neighbor. From our experiments (Rückert, 2002) it seems to be more time-efficient to use k-term DNF formulae as candidates and to add or remove literals to generate the neighboring candidates. A reasonable choice for the global scoring function is the number of positive and negative examples that are misclassified by the current formula: $score_P(S) =_{def} |\{x \in Pos \mid x \text{ violated by } S\} \cup \{x \in Neg \mid x \text{ satisfied by } S\}|$. Obviously, a candidate is a solution, if its score is zero. We are therefore aiming for minimal, not maximal score.

Another important design issue for SLS algorithms is the choice of the neighborhood relation and the decision rule. The decision rule specifies which of the neighbors is chosen as the next candidate. A straightforward decision rule for k-term DNF learning SLS algorithms could evaluate all formulae that differ from the current candidate by one literal and choose the neighboring formula with the lowest score. As it turns out, this approach is not very effective (Rückert, 2002). In the following we present a more target-driven decision process to boost the search.

The main idea is to concentrate on those changes that will correct the misclassification of at least one misclassified example. Assume p is an uncovered positive example. Thus,

Algorithm 2 An SLS algorithm for k-term DNF learning.

```
procedure SLSearch($k$, $maxSteps$, $p_{g1}$, $p_{g2}$, $p_s$)
  $H \leftarrow$ a randomly generated $k$-term DNF formula
  $steps \leftarrow 0$
  while $score_L(H) \neq 0$ and $steps < maxSteps$ do
    $steps \leftarrow steps + 1$
    $ex \leftarrow$ a random example that is misclassified by $H$
    if $ex$ is a positive example then
      $t \leftarrow$ { with probability $p_{g1}$: a random term in $H$
                     otherwise: the term in $H$ that differs
                     in the smallest number of literals from $ex$
      $l \leftarrow$ { with probability $p_{g2}$: a random literal in $t$
                     otherwise: the literal in $t$ whose removal
                     decreases $score_L(H)$ most
      $H \leftarrow H$ with $l$ removed from $t$
    else if $ex$ is a negative example then
      $t \leftarrow$ a (random) term in $H$ that covers $ex$
      $l \leftarrow$ { with probability $p_s$: a random literal $m$
                     so that $t \wedge m$ does not cover $ex$
                     otherwise: a random literal whose
                     addition to $t$ decreases $score_L(H)$ most
      $H \leftarrow H$ with $l$ added to $t$
    end if
  end while
  return $H$
end procedure
```

the current candidate formula c is obviously too specific: we have to remove at least one literal in order to satisfy p. Of course it does not make sense to modify a random term in c, because one might then affect a large number of currently correctly classified examples. A more sensible strategy would generalize the term t in c that differs in the smallest number of literals from p. One can then evaluate the formulae that differ in one literal in t as neighbors and choose that neighbor with the lowest score.

Similar considerations can be made for adding literals: assume n is a covered negative example. Then the current formula is obviously too general. Let t be the term that covers n. We have to add a literal to t in order to make n uncovered. Again we can evaluate generate a set of neighbors by adding one literal to t and then choose that neighbor whose score is lowest. This consideration leads to a decision rule and a neighborhood relation for the final algorithm, that use as much information as possible to guide the search: the decision rule first selects a random misclassified example e. If e is an uncovered positive example, the algorithm performs a generalization step as explained above; if e is a covered negative example, it performs a specialization step. Of course, this algorithm can get stuck in a local optimum quite quickly. It makes sense to replace each decision step with a random choice from time to time to escape those local optima. There are two decisions to be made during a generalization step and one decision for the specialization step. Thus, we have three different places to perform a random instead of an informed decision with certain probabilities. Algorithm 2 sketches the idea. Note that k is a fixed input parameter. However, the algorithm can easily be extended to solving the DNF minimization problem with varying values of k (usually by adding branch-and-bound techniques). For noisy and inconsistent data sets, the algorithm returns the formula that misclassifies as few instances as possible. This allows for an easy extension to noisy data.

4. Experiments

In this section we will empirically evaluate the performance of the SLS algorithm. For the purpose of this paper we are only concerned with compression instead of prediction: we aim at finding a compact representation of the information given in the training set, not necessarily a representation that performs well on unseen data. The characteristics of SLS with regard to prediction are the topic of a companion paper (Rückert & De Raedt, 2003).

The experiments are realized in three steps. First, we compare our SLS algorithm to other SLS algorithms applied to k-term DNF learning. As – to the best of our knowledge – there exists no other published SLS algorithm that directly works on the k-term DNF learning problem, we use a polynomial reduction of the k-term DNF problem to the well-known SAT problem. This in turn enables us to employ the widest possible range of SLS algorithms for solving the reduced k term DNF learning problem, including GSAT, WalkSat, etc. Secondly, we compare our SLS algorithm with three established rule learning systems. While separate-and-conquer does not guarantee to find the shortest possible formulae, it is a popular choice for settings in which runtime or predictive accuracy are more important than pure compression. Using the SLS algorithm we can quantify the overhead relative to the optimal formula size when using separate-and-conquer on hard problems taken from the phase transition region. Thirdly, to show the relevance of the k-term DNF learning problem, we apply our SLS algorithm to a challenging chess-endgame problem. This problem is directly relevant for practical applications, because the key challenge in endgame databases is compression. Compression in our framework corresponds to finding the minimum k-term DNF formula.

4.1. SLS on SAT-encoded k-term DNF Learning Problems

Kamath et al. present a reduction scheme to encode k-term DNF learning problems as satisfiability problems (SAT) (Kamath et al., 1992). Using this reduction one can encode

Table 1. The success rates (i.e. fraction of tries that found a solution) for various SLS algorithms running on the reduced test sets. For GSAT+Tabu we state the size of the tabu table instead of a noise level

Algorithm	Noise Level	Success Rate	Success Rate	Success Rate
		Test Set 1	Test Set 2	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%
Native SLS	0.25	100%	99.3%	82.7%
	0.5	100%	99.5%	69.2%
	0.75	100%	97.5%	36.2%

k-term DNF learning problems as satisfiability problems and then use some of the many published SLS algorithms for SAT. In (Rückert et al., 2002) we showed that WalkSAT is able to solve a large fraction of hard SAT-encoded k-term DNF learning problems. This result can be used as a reference for the native SLS algorithm 2.

We start with a survey of various established SLS algorithms for the satisfiability problem. For our experiments, we used the same three data sets as in (Rückert et al., 2002). Each data set contains one hundred soluble problem instances taken from the hard region of the phase transition. The first test set was generated with $|Pos| = |Neg| = 10$ and $n = 10$, the second with $|Pos|$=$|Neg|$= 20 and n=42, and the third one with $|Pos|$=$|Neg|$= 30 and n=180. In our initial experiments we ran ten tries per problem instance and counted the number of successful tries (i.e. tries that found a solution) for each algorithm. We used the default cutoff values: 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 and for native SLS we used a cutoff value of 50000 steps. Note that the noise level for native SLS in table 1 denotes the values for p_{g1}, p_{g2}, and p_s. Table 1 shows the fraction of successful tries for each algorithm. As can be seen, only WalkSAT and the native SLS algorithm are able to find a significant part of the solutions. Similarly distinct results can be obtained for other noise settings and cutoff values.

Comparing the performance of SLS algorithms is difficult: the performance of a given algorithm depends on the used data sets, the values for the noise probabilities, and the cutoff value. Furthermore, pure runtime is a bad indicator of an algorithm's performance, because an algorithm with comparably low runtime might output "no solution found" for many more problem instances than an algorithm with comparably bad runtime. We therefore followed the methodology outlined in (Hoos, 1998) to take this trade-off into account. First of all, we investigate the runtime behavior of WalkSAT and algorithm 2 for the problems in the third test set. We set the noise level to 0.25 for both algorithms. In figure 2 we plot P("algorithm needs less than x steps to find a solution") against x for a typical problem instance in the third data set. The upper curve shows the distribution for the native SLS algorithm, the lower curve the distribution for WalkSAT. Both curves strongly resemble an exponential distribution. According to a χ^2 test the distribution of the native algorithm is indeed an exponential distribution with an error probability of 0.15. As explained in (Hoos, 1998) this also indicates that unlike for some other SLS algorithms a low cutoff value with frequent restarts will not boost the algorithm's performance.

The graph also shows that our native algorithm is able to find a larger fraction of solutions than WalkSAT when both algorithms use the same cutoff value. Unfortunately, the number of candidate-neighbor steps is not an adequate indicator for runtime, because WalkSAT's steps are simpler than the ones of algorithm 2 and can be executed faster. We therefore identified the distribution of P("algorithm needs less than x milliseconds to find a solution") and compared the curves for both algorithms. Just as in figure 2 the resulting curves also resemble an exponential distribution. If the first curve dominates the second curve for every x-value, then the expected runtime of the first algorithm is better than the runtime of the second algorithm for all cutoff values. If one does not consider the first hundred milliseconds (because algorithm performance is arbitrary in

Table 2. The average number of terms of the induced formulae, together with standard deviation and training set accuracy.

Test Set				C5.0		RIPPER		PART	
k	$\|Pos\|$	$\|Neg\|$	n	Terms	Accuracy	Terms	Accuracy	Terms	Accuracy
3	10	10	10	3.93±1.15	86.25%	3.87±0.81	100.00%	4.15±0.84	87.85%
3	20	20	42	6.09±1.28	93.50%	4.19±0.59	100.00%	5.97±0.75	95.65%
3	30	30	180	6.41±1.25	95.38%	4.23±0.44	100.00%	6.28±0.52	97.18%
3	40	40	773	6.93±1.30	94.95%	4.29±0.46	100.00%	6.65±0.70	98.19%
4	30	30	51	7.86±1.68	93.80%	5.61±0.55	100.00%	7.86±0.80	96.08%
4	40	40	141	8.78±1.80	95.10%	5.60±0.49	100.00%	8.24±0.62	97.23%
4	50	50	387	8.62±1.82	91.38%	5.58±0.50	100.00%	8.32±0.55	98.12%
5	30	30	24	8.77±2.18	91.48%	6.98±0.70	100.00%	8.94±1.11	93.67%
5	40	40	46	9.50±2.14	93.21%	7.02±0.57	100.00%	9.98±0.90	96.49%
5	50	50	90	10.42±2.15	95.04%	7.04±0.57	100.00%	10.12±0.83	97.27%

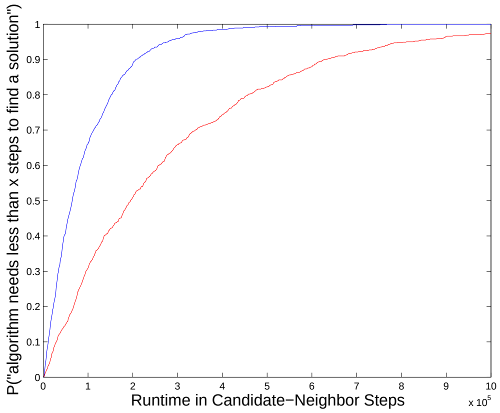

Figure 2. The probability of finding a solution plotted against the number of steps performed by the native SLS algorithm (above) and WalkSAT (below) on a hard problem instance.

the setup phase), then algorithm 2 dominates WalkSAT in seventy eight of the one hundred problem instances in the test set. We also calculated the size of the area between the two curves as a measure of the average expected speedup that is gained when using the dominating algorithm instead of the dominated algorithm. We found that the native algorithm dominates in eighty eight cases, but the speedup (as measured by the area between the two curves) is often rather small; in seventy seven cases the absolute value of the speedup is smaller than one hundred milliseconds. These results suggest that the native SLS algorithm compares favorably with the best available SLS algorithm for SAT-encoded k-term DNF learning problems. Another advantage of the native SLS algorithm is that the encoding step can be left out. A SAT-encoded k-term DNF problem contains $k \cdot (n \cdot (|Pos| + 1) + |Neg|) + |Pos|$ clauses. For example, a SAT-encoded version of the first king-rook-king endgame problem in section 4.3 contains 96045 clauses and takes 54.7 MB of disk space, when saved in the .cnf format used by WalkSAT. Generating and processing this data can take considerable amounts of time.

4.2. Rule Learning on Hard k-term DNF Learning Problems

As outlined in section 2, the general k-term DNF learning problem is NP-complete. That means that complete exhaustive search algorithms are not able to find solutions in a reasonable time frame for most problem instances. As demonstrated above, SLS algorithms can alleviate this dilemma at the cost of losing completeness. If low runtime is very important (such as for very large data sets) one is sometimes willing to also forego finding the shortest possible formula that is consistent with the examples. In such a setting one would accept finding a slightly longer-than-optimal formula in order to gain acceptable runtime. This is indeed the case for the classical rule learning setting, where high predictive accuracy and efficient runtime behavior is generally considered more important than high compression. Consequently, most rule learning algorithms utilize a separate-and-conquer approach (Fürnkranz, 1999) to find adequately short, but not necessarily optimally short formulae. In the following we investigate the degree of compression that can be achieved using this separate-and-conquer algorithm on hard k-term DNF learning problems taken from the phase transition region. This can be seen as a hard benchmark for both, the SLS algorithm and the rule learning algorithms. It also sheds some light on the inherent compromise between compression on one side and accuracy and runtime on the other side.

We examine the separate-and-conquer approach as implemented in popular rule learning algorithms: RIPPER extends the standard separate-and-conquer approach with incremental reduced error pruning and an iterated post-processing optimization step. PART avoids over-pruning by obtaining rules from partial decision trees. Additionally, we included C5.0, an improved version of the classic C4.5 rule learning algorithm. We generated ten sets of hard problem instances, all taken from the phase transition region of the problem setting space. The first three test sets are the same ones as in section 4.1: we used the complete algorithm outlined in section 2 to remove those problems that are not soluble for the stated k. For the remaining seven test sets the complete algorithm requires huge computational resources; we therefore use the SLS algorithm outlined in section 3 to remove all supposedly insoluble problem instances until each test set contains one hundred problems that are soluble for the given k.[1]

We apply each rule learning algorithm to each data set. Since we are *not* interested in high predictive accuracy but in high compression, we disable pruning for the RIPPER algorithm. C5.0 and PART do not offer any option to disable pruning. Theoretically, they should be able to find even shorter formulae, because the formulae generated by these two algorithms do not classify all examples correctly. We run C5.0 and PART with pruning, but additionally state the percentage of correctly classified examples in the results table. The average number of terms in the induced formulae and standard deviation are stated in table 2. PART and C5.0 generate formulae that are between 30% and 100% larger than necessary. RIPPER found considerably shorter formulae than C5.0 and PART. This might indicate that RIPPER's iterated post-processing step – while being computationally not too demanding – can decrease the average size of the induced formula. The pure separate-and-conquer approach as employed by PART seems to be not very well suited for compression of hard random problems. If, of course, the training set is sampled according to a more favorable distribution (e.g. the uniform distribution), separate-and-conquer can work remarkably well.

4.3. King-Rook-King Chess Endgame

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 successful application of SLS algorithms. In the following we examine, whether or not SLS algorithms can efficiently deal with structured real-world problems.

The domain of chess endgames provides an ideal testbed for DNF minimization, because here we deal with noise-free datasets with discrete attributes only, and since we have complete domain knowledge 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 (Bain, 1994; Quinlan & Cameron-Jones, 1995; Nalimov et al., 2000) and is of continuing interest (Fürnkranz, 2002). We decided to especially examine the simplest chess endgame, King and Rook versus King. In his PhD thesis Bain (Bain, 1994) 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 UCI Machine Learning Repository (Blake & Merz, 1998) and has been used as a benchmark, for instance in Quinlan's evaluation of FOIL (Quinlan & Cameron-Jones, 1995). Note that this data set is different from the easier data set for learning *legality* in the King and Rook versus King endgame.

Table 3. The number of clauses/terms learned by FOIL, GCWS, PFOIL, RIPPER, and the SLS algorithm. Each row states the number of moves required to win and the number of positions. For FOIL and GCWS we give the number of horn clauses, for PFOIL, RIPPER, and SLS the number of terms that were induced.

MOV.	POS.	FOIL	GCWS	PFOIL	RIPP.	SLS
0	27	3	6	3	3	3
1	78	13	12	21	5	4
2	246	15	20	30	10	6
3	81	17	44	47	18	6
4	198	41	89	92	37	11
5	471	55	185	221	57	17
6	592	82	-	240	67	23
7	683	114	-	349	59	31
8	1433	180	-	704	106	43
9	1712	219	-	785	132	55
10	1985	248	-	1076	153	64
11	2854	345	-	1295	183	85
12	3597	500	-	1303	180	99
13	4194	326	-	1288	168	110
14	4553	231	-	566	105	99
15	2166	68	-	87	61	39
16	390	3	-	9	9	3

The problem is stated as a repetitive application of a learning algorithm: from a database of all legal positions the algorithm learns all positions won in zero moves first. These positions are then removed from the database and the algorithm is subsequently used to learn positions won in one, two, three, etc. moves. This process is repeated until the learning algorithm separates positions won in sixteen moves from drawn positions. Originally, the problem is stated in a first-order representation. 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 positions: the location of the chessmen, the distances 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 $2 \cdot 2 \cdot 3 \cdot 8 = 86$

[1] We do not check if the problem instances are soluble for a lower k than the stated one. The test sets should therefore be considered of being soluble using the stated k or eventually a lower one.

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

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 $96 + 48 = 142$ variables.

We ran the native SLS algorithm outlined in section 3 on the problem instances for the problems in the KRK endgame suite. Table 3 shows the results. As a comparison we also state Quinlan's results with FOIL and Bain's original results with GCWS (Bain, 1994). FOIL and GCWS are ILP systems and can therefore make use of background knowledge and built-in predicates. However, it turns out that, even though FOIL and GCWS use a much more expressive representation language, they generate a larger set of rules.[3] To compare the results of the SLS algorithm with propositional algorithms we applied RIPPER, because it was able to find the smallest sets of rules in our preceding investigation. Additionally we evaluate PFOIL, a propositional version of FOIL (Mooney, 1995), to get a comparable result for a FOIL-based learning strategy. While PFOIL induces up to thirteen times more terms than necessary, RIPPER performs relatively well with up to three times as many terms as generated by the SLS algorithm. All in all, SLS clearly outperforms the other algorithms.

5. Conclusion

We presented a novel stochastic local search algorithm for finding short formulae that are consistent with sets of positive and negative examples. We evaluated this algorithm in comparison to SAT-based SLS algorithms and to traditional rule-learning approaches on hard problem instances taken from the phase transition region. We finally showed that the algorithm can be applied successfully to a real world problem. These results reveal some possible routes for further research. First of all, one can investigate SLS on other learning settings than k-term DNF learning. (Chisholm & Tadepalli, 2002) present some encouraging results for SLS in general rule learning settings. Second, we are currently evaluating the compression of separate-and-conquer-approaches on real-world problems, instead of artificially generated ones. Third, one could extend and improve the presented algorithm in various directions, such as adding noise handling or multi-valued and continuous attributes.

References

Bain, M. (1994). *Learning logical exceptions in chess*. Doctoral dissertation, Department of Statistics and Modelling Science, University of Strathclyde, Scotland.

Blake, C., & Merz, C. (1998). UCI repository of machine learning databases.

Chisholm, M., & Tadepalli, P. (2002). Learning decision rules by randomized iterative local search. *Proceedings ICML-02* (pp. 75–82). Morgan Kaufmann.

Fürnkranz, J. (1999). Separate-and-conquer rule learning. *Artificial Intelligence Review*, *13*, 3–54.

Fürnkranz, J. (2002). Personal Communication, April 2002.

Gomes, C. P., Selman, B., & Kautz, H. (1998). Boosting combinatorial search through randomization. *Proceedings AAAI-98* (pp. 431–437). Menlo Park: AAAI Press.

Hoos, H. H. (1998). *Stochastic local search - methods, models, applications*. Doctoral dissertation, Technische Universität Darmstadt.

Kamath, A. P., Karmarkar, N. K., Ramakrishnan, K. G., & Resende, M. G. C. (1992). A continuous approach to inductive inference. *Mathematical Programming*, *57*, 215–238.

Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory*. Cambridge, Massachusetts: The MIT Press.

McAllester, D., Selman, B., & Kautz, H. (1997). Evidence for invariants in local search. *Proceedings AAAI-97* (pp. 321–326). Providence, Rhode Island.

Mooney, R. J. (1995). Encouraging experimental results on learning CNF. *Machine Learning*, *19*, 79–92.

Nalimov, E. V., Haworth, G. M., & Heinz, E. A. (2000). Space-efficient indexing of chess endgame tables. *ICGA Journal*, *23*, 148–162.

Quinlan, J. R., & Cameron-Jones, R. M. (1995). Induction of logic programs: FOIL and related systems. *New Generation Computing*, *13*, 287–312.

Rückert, U. (2002). Machine learning in the phase transition framework. Master's thesis, Albert-Ludwigs-Universität Freiburg. `http://home.informatik.tu-muenchen.de/rueckert/da.pdf`.

Rückert, U., & De Raedt, L. (2003). An experimental evaluation of the covering algorithm in rule learning, to be submitted.

Rückert, U., Kramer, S., & De Raedt, L. (2002). Phase transitions and stochastic local search in k-term dnf learning. *Machine Learning: ECML 2002* (pp. 405–417). Springer.

Selman, B., & Kautz, H. A. (1993). Domain-independant extensions to GSAT : Solving large structured variables. *Proceedings IJCAI-93* (pp. 290–295).

Selman, B., Kautz, H. A., & Cohen, B. (1996). Local search strategies for satisfiability testing. In *Cliques, coloring, satisfiability: the second dimacs implementation challenge*, 521–532. AMS Series in Discrete mathematics and Theortical Computer Science 26.

Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. *Proceedings AAAI-92* (pp. 459–465).

[3] Of course, the rules induced by FOIL and GCWS are horn clauses as opposed to terms in the case of RIPPER, PFOIL and SLS.