Parameter Adjustment Based on Performance Prediction: Towards an Instance-Aware Problem Solver

Frank Hutter
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, BC, V6T 1Z4, Canada
hutter@cs.ubc.ca

Youssef Hamadi
Microsoft Research
7 JJ Thomson Ave
Cambridge, CB3 0FB, UK
youssefh@microsoft.com

December 27, 2005
Technical Report
MSR-TR-2005-125

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

Tuning an algorithm's parameters for robust and high performance is a tedious and time-consuming task that often requires knowledge about both the domain and the algorithm of interest. Furthermore, the optimal parameter configuration to use may differ considerably across problem instances. In this report, we define and tackle the *algorithm configuration problem*, which is to automatically choose the optimal parameter configuration for a given algorithm on a per-instance base. We employ an indirect approach that predicts algorithm runtime for the problem instance at hand and each (continuous) parameter configuration, and then simply chooses the configuration that minimizes the prediction. This approach is based on similar work by Leyton-Brown et al. [LBNS02, NLBD$^+$04] who tackle the algorithm selection problem [Ric76] (given a problem instance, choose the best algorithm to solve it). While all previous studies for runtime prediction focussed on tree search algorithm, we demonstrate that it is possible to fairly accurately predict the runtime of SAPS [HTH02], one of the best-performing stochastic local search algorithms for SAT. We also show that our approach automatically picks parameter configurations that speed up SAPS by an average factor of more than two when compared to its default parameter configuration. Finally, we introduce sequential Bayesian learning to the problem of runtime prediction, enabling an incremental learning approach and yielding very informative estimates of predictive uncertainty.

1 Introduction

The last decade has seen a dramatic rise in our ability to solve combinatorial optimization problems in many practical applications. Amongst the most important reasons for this development are considerable advances in algorithms that exploit special domain-specific characteristics. However, peak performance usually comes at a loss of generality since the optimal search strategy differs across problem domains. Moreover, state-of-the-art algorithms for combinatorial optimization, be they based on tree search or local search, tend to have a fair amount of parameters whose optimal settings differ on an instance-by-instance base.

Recent years have seen a number of approaches to construct portfolios of algorithms that choose the algorithm that is expected to perform best for each problem instance [LL98, LBNS02, NLBD$^+$04, GHBF05]. There have also been a number of approaches to automatically find the best default parameter configuration for a single algorithm [KJ95, BSPV02, Hut04, SM05, ADL06].

However, to our best knowledge, there exist only precious little approaches for configuring an algorithm's parameters on a per-instance base automatically, that is, taking into account the characteristics of the problem instance at hand and past experience on similar instances. We call this problem the *algorithm configuration problem.* The only solution approach to this problem we are aware of is the Auto-WalkSAT framework [PK01] which is confined to a small domain and does not perform well for all classes of SAT instances. In this report, we introduce a general framework for solving the algorithm configuration problem. Our approach is based on predicting the runtime of the parametric algorithm at hand for all possible parameter configurations (in the case of continuous parameters there are infinitely many), and to find a configuration that approximately minimizes this prediction. When our runtime predictions are accurate, this configuration indeed minimizes runtime. We show experimental results confirming that automatically chosen instance-specific parameter configurations for a local search algorithm can outperform its best overall parameter configuration.

We furthermore introduce the use of Bayesian techniques to the problem of runtime prediction, which enables a simple incremental learning approach and more importantly the quantization of uncertainty in our estimates of runtime. These techniques widen the applicability of runtime prediction to include a number of practically relevant scenarios.

Our contributions in this report are manifold, the most important ones being:

- We demonstrate that it is possible to predict the (median) runtime of local search algorithms for SAT fairly accurately. Even though Leyton-Brown et al. [LBNS02, NLBD$^+$04] have demonstrated that the runtime of branch and bound algorithms for SAT and other NP-hard problems can be predicted, to this end no equivalent study was performed for local search.[1]

- We show how to predict the runtime of an algorithm with continuous parameters and use this methodology to implement an effective and general approach to solve the algorithm configuration problem.

- Using the above approach, we automatically tune the parameters of the stochastic local search SAT algorithm SAPS [HTH02] on a per-instance base. This achieves a reduction in runlength on unseen test instances by more than a factor of two on average when compared to the SAPS default parameter configuration.

- We employ a simple Bayesian approach to introduce incremental learning and a quantization of predictive uncertainty to the field of runtime prediction, widening its practical applicability considerably.

The rest of this report is structured as follows. Following a comprehensive discussion of related work in Section 2, the report is loosely split into two parts. In the first part, Section 3 details the use of regression techniques in runtime prediction and Section 4 demonstrates its applicability to predict the median runtime of local search algorithms. Section 5 then shows how to predict the runtime for different parameter configurations of an algorithm, and Section 6 applies this approach to the SAPS algorithm. Part two covers a fairly independent extension of runtime prediction methodology. It starts with some practical considerations about real-world scenarios in Section 7, motivating incremental learning approaches that can deal with multiple domains. Section 8 then gives the basics of sequential Bayesian linear regression which is demonstrated in Section 9 to offer a solution to the problems

[1] Indeed, Kevin Leyton-Brown stated in personal communication that they have conducted preliminary (and unpublished) inconclusive experiments that rather suggested runtime prediction to be infeasible for local search algorithms.

arising in the above real-world scenarios. Finally, Section 10 concludes the report and gives pointers for future work.

2 Related Work

Over the last decade, a considerable number of approaches towards automated parameter tuning and algorithm selection have been proposed. We group these into three distinct categories, namely approaches for finding the best default parameter configuration of a single algorithm across a set of given instances; portfolio approaches that choose between several algorithms on a per-instance base before actually running the algorithms; and approaches that alternate between different problem solving strategies during the execution of an algorithm. We discuss each of these approaches in detail in the following sections. Our present work on algorithm configuration is most similar in spirit to the approaches for algorithm selection we review in Subsection 2.2.

2.1 Finding the best parameter configuration for a problem distribution

The most general problem in automated parameter tuning is to find the best (default) parameter configuration for a particular algorithm and problem domain. We define the *best default parameter configuration problem* as the one that minimizes the overall expected runtime (or maximizes overall quality in an optimization problem) across the problem instances of a domain when the same parameter configuration is to be used for all instances. There is an abundant number of applications for algorithms that automatically find a good default parameter configuration, be it in industry (where the default parameter configuration may vary for different problem domains) or simply in algorithm development (where researchers spend much of their precious time optimizing their algorithms' parameters). Approaches that find the best default parameter configuration have the potential to improve parametric algorithms for *any* kind of problem. All that is required for this approach to be applicable is a distribution over (or a set of) problem instances and a (possibly black-box) parametric algorithm. However, it should be emphasized that a general optimization tool will necessarily fall short of specialized optimizers which can exploit knowledge about the problem domain and the algorithms to be optimized.

In this section we review a number of quite heterogeneous approaches for finding the best default parameter configuration. When the heuristics employed by a CSP solver are viewed as its parameters, automated parameter tuning for CSP solving dates back to the MULTI-TAC system by Minton [Min93, Min96] . MULTI-TAC requires as input a number of generic heuristics as well as a specific problem domain and a distribution over instances. It adapts the generic heuristics to the problem domain and automatically generates domain-specific LISP programs implementing them. A beam search is then used in order to choose the best LISP program where each program is evaluated by running it on a number of problem instances sampled from the provided distribution.

There is a number of fairly recent approaches for automated parameter tuning. These approaches are quite heterogeneous, but all of them optimize a certain objective function that is implicitly defined via the performance of the algorithm on a distribution of problems. Probably the biggest challenge in automatic parameter tuning is the great number of possible parameter configurations to choose from. Even if all algorithm parameters are discrete, there are exponentially many possible configurations, and since continuous parameters lead to an infinite number of choices all approaches we are aware of discretize continuous parameters.[2] In contrast, the approach we present in this paper works directly in the continuous space, rendering discretization superfluous.

Birratari et al. [BSPV02] introduce a racing algorithm for configuring metaheuristics. This algorithm takes as input a finite number of parameter configurations for an algorithm and a problem distribution. It runs the algorithm with each single parameter configuration on a number of instances. After all configurations have been run for an instance, a non-parametric statistical test is used in order to filter out configurations that are significantly worse than others. This process is iterated until a cutoff time is reached and only a small number of good configurations is left. Experiments with 256 different parameter configurations (all combinations of 4 continuous parameters discretized to 4 values each) show promising results in the domain of the MAX-MIN ant system for solving the travelling salesman problem.

Hutter [Hut04] introduced an iterated local search (ILS) algorithm for automated parameter tuning, called ParamILS. Taking the same input parameters as the above mentioned racing algorithm, ParamILS starts with some default parameter configuration and tries to improve it locally. This can be seen as performing a greedy local search, where performance is measured by running the algorithm with the respective configuration on a number of instances, and the objective function combines the number of instances solved and time taken to do so. Once in a local minimum of this implicit cost surface, a number of parameter values are perturbed at random to arrive at a new starting point for a greedy local search. Interestingly, ParamILS was used to tune the parameters of another ILS algorithm that solved the most probable explanation problem in Bayesian networks. That ILS algorithm had a large number of continuous and discrete parameters which led to over one million possible discretized parameter configurations, rendering a racing approach infeasible. ParamILS found parameter configurations that were considerably better than the ones previously found manually.

The CALIBRA system [ADL06] combines fractional experimental design with a local search mechanism. Being limited to at most five free parameters, it starts off by evaluating each parameter configuration in a full factorial design with two values per parameter. It then iteratively homes in to good regions in parameter space by employing fractional experimental designs that evaluate nine configurations around the so far best performing configuration. The grid for the experimental design becomes finer in each iteration, which provides a nice solution to the automatic discretization of continuous parameters. Once a local optimum is found and cannot be refined anymore, the search is restarted (with a coarser grid) by combining some of the best configurations found so far, but also introducing some noise for sake of diversification. The experiments reported in [ADL06] show great promise in that CALIBRA was able to find parameter settings for six independent algo-

[2] Only the fractional experimental design approach paired with local search in [ADL06] discretizes with increasingly fine levels of granularity. It thereby avoids many problems associated with discretization.

rithms that matched or outperformed the respective originally proposed parameter configurations.

Finally, Srivastava and Mediratta [SM05] use a decision tree classification approach to classify the space of possible parameter configurations into good and bad configurations for a domain. Neither instance-specific nor domain-specific characteristics are part of the decision tree. Once more, all parameters are discrete or discretized (which is here described as an advantage since by means of discretization the algorithm can handle "diverse parameter types") and the experiments cover scenarios with less than 200 possible parameter configurations. Their approach requires the user to define a so-called performance criterion (such as "the algorithm solves the problem within five minutes"). A training instance is then called learnable if the performance criterion is met by some parameter configuration and missed by some other one. Predictably, the typically large inter-instance variability in runtimes even for uniform domains complicates this classification based on a fixed user-defined performance criterion, already rendering a third of the training instances unlearnable. Moreover, the complete learning process has to be repeated whenever the performance criterion is modifed. The learned decision tree is used in order to choose the presumably best parameter configuration for a domain by one of three possible "ranking functions" which all aim at choosing a leaf in the decision tree with many good and few bad configurations. Unfortunately, the experimental evaluation presented in [SM05] is completely inconclusive since the only comparison made is to the *worst* performing parameter configuration, as opposed to the best performing one or at least to the default parameter configurations all the tested algorithms come with.

In summary of these recent approaches, there are three possibilities to handle the exponential blowup in possible parameter configurations. Birratari et al. [BSPV02] try to minimize the number of times an algorithm is run with poor configurations. This can work if the number of possible configurations is fairly small and the number of runs necessary to statistically distinguish good from bad parameter settings is also small. The iterated local search by Hutter [Hut04] and the CALIBRA system [ADL06] concentrate on well-performing regions of parameter space and completely ignore regions that do not seem promising enough. This scales up to much larger numbers of possible configurations, but there is a possibility that an interesting region is missed alltogether.[3] Finally, model-based approaches based on machine learning try to represent the whole parameter space compactly. So far, the only machine learning work for finding good default parameter configurations we are aware of is the decision tree approach by Srivastava and Mediratta [SM05], but this did not convince us yet. We plan to study the use of machine learning for this problem closer in the future.

Some related work from the machine learning literature is presented in [KJ95]. In supervised machine learning, problems are very different than in combinatorial optimization, the aim being to build a predictive model which is typically parameterized by a number of *model parameters*. These model parameters are fitted or learned from a training data set, and it is very important not to confuse them with

[3] For many algorithms, there exist strong connections between the algorithm parameters which may lead to symmetries in parameter space. In the presence of such symmetries, focussing on a representative sub-region may actually be a very sensible strategy.

the *algorithm parameters* we deal with in this paper.[4] Model parameters are an artifact which Bayesians would optimally like to integrate over, and even in classic machine learning it does not make any sense to adapt these model parameters on a per-instance base. However, it *does* make sense to separately fit model parameters for distinct problem domains.

The objective in (classical parametric non-Bayesian) machine learning for many data sets can be seen as finding the best parameter setting for each of the data sets. In [KJ95], Kohavi and John demonstrate that it is possible to automatically find good settings for the parameters of the popular decision tree learning algorithm C4.5. The methodology they use is very similar to the ones applied for finding the best default parameter setting for combinatorial search algorithms (cf. the above mentioned iterated local search in parameter space in [Hut04]): for each of the 33 data sets they studied, they perform a best-first search in parameter space, where each parameter setting is evaluated by running cross validation on the training set. Their approach finds different good parameters for each of their data sets, and they significantly outperform the overall best default parameter setting of C4.5.

To finalize the discussion of best default parameter configurations, we would like to emphasize that this problem is a very worthwhile area of study due to its generality. Note that automated parameter tuning in this setting is by no means limited to search algorithms. In contrast, the problem formulation applies to any field of algorithmic study where a set of arbitrary parameters needs to be set in order to achieve the best overall performance on a distribution of problems. Unfortunately, the performance of a single default parameter configuration depends heavily on the homogenity of instances within a problem domain as well as on the sensitivity of algorithms to their parameters. In Section 5, we introduce a novel approach that tunes algorithm parameters conditional on the current instance to be solved. This approach is more powerful than finding the best default parameter configuration, but it is also less general since it requires some features to distinguish instances from one another.

2.2 Algorithm selection on a per-instance base

Sometimes it is not easy to tell ahead of time which domain a particular problem comes from. For example, if a SAT solver is used as a "General Problem Solver" to solve problem encodings from a variety of domains, it may be regularly confronted with SAT instances of unknown origin. Similarly, the problem modelling may change over time such that even problem instances from the same domain can become very heterogeneous.

In such a scenario, using only the single best algorithm for a domain will not lead to peak performance. This is due to the fact that the more diverse instances are, the more the optimal solution approaches tend to differ. The problem a practitioner faces for each new problem instance is the *algorithm selection problem* [Ric76], that is, given a problem instance to select the most appropriate algorithm to solve it.

[4] For readers with a background in Bayesian machine learning, it may help to view algorithm parameters as decision variables that need to be instantiated, whereas model parameters are random variables. Even in machine learning there are a number of algorithm parameters that need tuning, even though they may be as trivial as the step size in a gradient descent optimizer or the number of EM steps to be performed.

Our work in this paper is very related to this problem, with the only difference that we select one of infinitely many parameter configurations for a single algorithm instead of selecting one of finitely many algorithms (with fixed parameter settings).

This section reviews a number of recent approaches for tackling the algorithm selection problem. One simple yet effective approach goes back to Donald Knuth [Knu75] who proposed to estimate the efficiency of backtrack search algorithms with the following Monte Carlo approach. First, sample a number of paths from the root of a search tree to some of its leaves by uniformly sampling a branch at each choice point. Appropriately weighted, these sampled paths can yield an estimate of the size of the search tree or the time necessary for a complete traversal. Lobjois and Lemaître [LL98] build on this work in order to estimate the same quantities for Branch and Bound algorithms for Max-CSP with various propagation methods. For each of a number of algorithms, they sample search trajectories by picking branches uniformly at random. However, in contrast to Knuth's method, they perform algorithm-specific propagations upon instantiation of a variable and keep track of the upper bound on solution cost (where the objective is to minimize cost), cutting off branches whose cost already exceeds the upper bound. For some problem instances, costly propagations pay off due to a massive reduction of the size of the explored search tree, whereas for other instances, additional propagations simply pose an overhead. Exactly this difference can be detected and exploited automatically by Lobjois' and Lemaître's method: for each algorithm, they compute an estimate of runtime by multiplying the estimated size of the search tree with the estimated time spent in each node. They then run the algorithm with minimal expected runtime. Their experiments with four Branch and Bound algorithms show that this method, which they call *Selection by Performance Prediction*(SPP), is for most instances only a bit slower than the optimal algorithm for that instance, and that SPP outperforms every single algorithm on average. However, SPP is only applicable for tree search algorithms and requires access to and modification of the algorithms' source codes.

A quite related approach is presented in the work of Leyton-Brown et al. [LBNS02, NLBD$^+$04]. For each algorithm of interest, they build a so-called *empirical hardness model*, which predicts the algorithm's runtime given a problem instance. Amongst others, empirical hardness models can be used for algorithm selection by simply choosing the algorithm with lowest predicted runtime. They are based on supervised machine learning, more precisely linear regression, and are applicable whenever meaningful features are available for the domain of interest that can be computed quickly and that are predictive of runtime. Apart from being more general, this approach differs from SPP in that it uses identical features to predict the performance of different algorithms. An empirical hardness model for a tree search algorithm could easily include the two features SPP uses (predicted size of search tree and cost per node) on top of other features. This would likely improve accuracy of the hardness model somewhat (it would very likely outperform SPP), but this has not been done so far, probably in part because Leyton-Brown et al.'s research focussed on black-box algorithms for which SPP features are not computable. Since our work on algorithm configuration builds on the one by Leyton-Brown et al. we detail their approach in Section 3.

Another method for algorithm selection by Gebruers et al. [GHBF05] comes from the field of case-based reasoning, a supervised classification method. Instead of predicting the runtime for each algorithm using regression, this method simply classifies problem instances according to which algorithm should best be used to solve them. This discriminative method does not suffer the overhead of having to predict runtime, which may be quite a challenging task. In contrast, it can directly model the decision boundary, which may be much simpler. However, the indirect approach via predictions of runtime has a number of very appealing advantages. Firstly, having an estimate of an algorithm's runtime can be useful in its own right, for example in a larger automated reasoning system that can allocate resources depending on expected problem hardness. Predictions about the runtime can also be interesting in scientific discovery when we try to find out what makes problem instances hard. Secondly, if runtime prediction is slightly off then the best algorithm may still be selected since runtime predictions only need to be accurate in a *relative* sense when solely used for selecting the best algorithm. Mistakes are more likely when two algorithms' runtimes are very close, such that mistakes are more likely when they don't really matter. On the other hand, when an algorithm performs very poorly for an instance, only a great prediction error can lead to it being predicted to be the best algorithm – an unlikely situation. Thirdly, probabilistic estimates, decision theoretic concepts, active learning and the like are more straight-forwardly integrated in the runtime prediction setting. We do indeed implement probabilistic estimates in this paper and plan to implement the other concepts in future work. Furthermore, runtime prediction is much more suitable in an incremental learning scenario because runtime predictions can already be improved from the execution of a single algorithm on a new instance. On the other hand, classification algorithms need to run many if not all algorithms on a new instance in order to subsequently use it as a training example. Finally, runtime prediction yields independent predictors for each algorithm which remain unchanged when new algorithms join the portfolio. In contrast, classification algorithms would have to relearn everything from scratch in this case.[5]

This Section so far dealt with methods to solve the algorithm selection problem. In this paper, we apply similar techniques to solving the algorithm configuration problem. The only other solution approach to the algorithm configuration problem we are aware of is the Auto-WalkSAT algorithm by Patterson and Kautz [PK01]. This approach builds on work on invariants in local search by McAllester et al. [MSK97]. That paper performed repeated runs of WalkSAT, studying the mean and standard deviation of the resulting solution qualities and in particular the ratio of the two, which they dubbed the *invariant ratio.* They showed empirical evidence that the optimal noise parameter settings for algorithms in the WalkSAT family tend to be close to the noise setting that minimizes the invariant ratio, plus 10%. Auto-WalkSAT exploits this property by explicitly searching for the noise setting that approximately minimizes the invariant ratio, and simply using this noise setting plus 10%. This very simple algorithm performs well for a large range of instances, finding different noise settings close to the optimal one for random, circuit test, graph colouring and planning problems. However, in the case of logistics problems, the optimal noise parameter setting is the noise setting that

[5] It may be possible to devise a scheme to only compare new algorithms against the so-far best algorithm for each problem instance, but this would mean having to construct and maintain a large number of pairwise classificators, which is clearly undesirable both conceptually and computationally.

minimizes the invariant ratio *minus* 30%, such that Auto-WalkSAT performs very poorly. Unfortunately, the relationship between invariant ratio and optimal noise setting is a purely empirical one, which does not hold for every instance type, as is demonstrated by its failure in the logistics domain.

Nevertheless, Auto-WalkSAT demonstrates that in well defined special domains it is possible to construct mechanisms that automatically and directly select a good performing parameter configuration for each problem instance. One can imagine a more involved machine learning algorithm that employs many instance features (such as the ones we use in this paper, but also the invariant ratio), to directly predict a good or even optimal parameter configuration for each instance. This direct approach may perform better than our indirect approach via runtime prediction since the functions to learn may be easier and since correlations between features and optimal parameter values (such as the one used in Auto-WalkSAT) can be exploited. On the other hand, more than one algorithm parameter and different types of parameters may complicate direct predictions. This is because correlations between the various parameters must be captured to find the best joint configuration of parameters, and because we are not aware of any off-the-shelf machine learning solutions with mixed discrete/continuous output. Further, the previously mentioned advantages of runtime prediction also speak for this indirect approach, which is why we chose to study it in this paper.

2.3 Online approaches

We distinguish our work on a priori algorithm configuration from a large class of competing approaches, namely those that tune algorithm parameters *during the search*. In the context of this paper, we also include algorithms in this class that switch between algorithms during the course of the search. We refer to the problems these algorithms are tackling as the *online* algorithm configuration and selection problems as opposed to the above *a priori* variants in which one commits to using a particular parameter configuration or algorithm *before* running the algorithm on an instance. Obviously, the online variants have more potential than the a priori ones since they are free to *reactively* exploit history information about what happened in previous phases of the search. However, since many decisions have to be made during the course of a search, the computational efficiency of the learning component becomes an important issue, which leads to research focussing on simple and often hard-coded "learning" rules that do not have much in common anymore with traditional machine learning approaches (one exception is the promising and principled work on reinforcement learning for search). Our approach is to first establish a solid baseline by tackling the less powerful but more general a priori algorithm configuration problem by means of state-of-the-art machine learning approaches. We hope that the lessons learned may carry over to help tackling the more complicated problem of reactively tuning algorithm parameters and choosing algorithms.

A borderline case in our distinction between a priori algorithm configuration and tuning parameters during the algorithm execution is the work by Horvitz et al. [HRG$^+$01]. They employ a decision tree to classify runs of a randomized algorithm into long and short runs and exploit it by cutting off runs that are predicted to be long, letting the others continue. This can considerably improve the average-case performance of algorithms with heavy-tails. Even though this approach makes a decision during the execution of the algorithm (cut/continue), we do not really view it as an online approach. This is due to the fact that it does not scale to drawing multiple decisions during an algorithm run, unless one is willing to learn a new predictor for each single decision to be drawn. In order to scale to the "real" online case, one would like to learn a single classifier which can handle both features that have been computed in the beginning of the search and in latter stages of the search, either by extending the feature vector as the search progresses or by updating it. Both approaches are somewhat problematic and scaling up to the "real" online case with supervised machine learning techniques is an open research problem.

It is hard to attribute the improvements an algorithm incrementally achieves on a problem during the course of the search to single decisions or series of decisions made by the algorithm. This problem calls for the use of reinforcement learning. The STAGE algorithm by Boyan and Moore [BM00] incrementally learns an function $f(\cdot)$ that predicts for every search state s the quality of the local optimum region reached by a basic local search method when started in s. STAGE alternates between such local search phases and pertubation moves that lead the search out of locally optimal regions and to states from which the next local search phase is predicted to reach good local optima. For this perturbation, STAGE employs a secondary search over search states that optimizes with respect to the learned function $f(\cdot)$, that is, with respect to the predicted objective function value after the next local search phase. Due to its alternation of local searches and pertubations, STAGE can be seen as an iterated local search (ILS) algorithm [HS04] with a principled pertubation. The original results presented in [BM00] were very encouraging, but unfortunately, we are not aware of any follow-up work that showed STAGE to outperform state-of-the-art metaheuristics, in particular ILS with simple random perturbations.

Lagoudakis and Littman [LL00] apply reinforcement learning to solve the algorithm selection problem for each recursive subinstance of an order statistic and a sorting problem.[6] For the sorting problem, they demonstrate that their algorithm automatically discovers that for large problems QuickSort performs very well while small instances are more readily solved by InsertionSort due to a lower constant factor in complexity. Their learned algorithm combines the two sorting algorithms by automatically choosing one of them based on the size of the subinstance. The choice of QuickSort for a (sub)instance results in two subinstances for which the decision between the two algorithms has to be made again. This process is iterated until the subinstances are small enough to be predicted easier to solve with InsertionSort than with QuickSort. This combined algorithm demonstrates very effectively that different algorithms can be optimal in different phases of problem solving.

Lagoudakis and Littman also apply this methodology to the problem of selecting branching rules in the DPLL procedure for SAT solving [LL01]. As in the above order statistic and sorting problems, the only state information used in the reinforcement learning algorithm is instance size, in the SAT domain expressed as the number of free variables. Experimental results show that for a variety of domains, after a fairly involved training phase they recover the performance of the single best branching rule for a test set from the respective domain by choosing one of the best

[6] In tree search algorithms, subinstances are the subtrees resulting from partial instantiations. If one decision is made for each such recursive subinstance, this is a perfect example for the "real" online case, whereas drawing only one decision per instance (and then sticking to this decision for all sub-instances) would not qualify as such.

(almost indistinguishable) branching rules at every node. They also show an experiment on artificially constructed instances that consist of two disjoint subinstances, one with 40 variables that allows for deep unit propagations, and one with 20 random variables. For this type of instances, it pays off in the beginning (first 40 levels of the tree) to use an expensive probing mechanism to select the variable assignment that leads to the most unit propagations. For the last 20 variable assignments (relating to the random subinstance), this strategy does not pay off and a simpler but faster branching rule should be employed instead. Experimental results show that this rule is recovered automatically by the reinforcement learning algorithm, and that it thus clearly outperforms the single best branching rule per instance. These results suggest that it is possible to outperform the single best algorithm for an instance if one can find a compact state representation that still discriminates between the possible actions. For the artificial instances in [LL01], instance size alone suffices, but for general problem instances and the general online algorithm selection problem this is an open research problem.

We now move on to a number of approaches that use simple and efficient estimators or heuristics to improve efficiency during tree searches for CSP. For the problem of selecting the next variable to branch on in a CSP, Sillito [Sil00] uses Knuth's idea [Knu75] of estimating the size of the search tree by sampling (Lobjois and Lemaître [LL98] already based their SPP algorithm on this idea – cf. Section 2.2). For each variable with small enough domain, he estimates the size of the complete search tree when that variable is instantiated next, and then picks the variable leading to the smallest predicted tree.

The *quickest first principle*(QFP) [BTW95] by Borrett et al. employs a number of algorithms of increasing power but also increasing complexity. Simple instances can often quickly be solved by simple strategies, while harder instances lead to so-called *thrashing* of simple approaches, that is, extensive exploration of useless subtrees. QFP exploits this idea by starting the search with a simple algorithm, using a heuristic thrashing detector to cut off the algorithm and start a more complex one once thrashing is detected. Experimental results suggest that this approach is able to solve easy problems quickly and to reduce the so-called exceptionally bad behaviour of algorithms by cutting them off once they begin to thrash.

In the setting of tree search for CSP, the Adaptive Constraint Engine (ACE) [EF01, EFW$^+$02, SLE04] by Epstein et al. uses a set of so-called advisors, heuristics that vote for possible actions during the search. These votes are taken at every decision point during the search and effectively compose the variable- and value-selection heuristics of ACE. Weights for the advisors are learned *after* solving each problem by examining the trace and appraising which choices were optimal. However, during algorithm execution ACE's strategies can still vary even if the advisor's weights do not change. Furthermore, depending on the ratio of instantiated variables, it employs three distinct stages in the search for which different heuristics turn out to be advisable. In the experimental analysis of [EFW$^+$02], ACE considerably reduced the number of backtracks necessary to solve a number of CSP instances, but the frequent voting required considerable time such that the overall computational time of ACE did not yet challenge straight-forward applications of standard heuristics. The recent application of so-called fast and frugal reasoning [SLE04] (which more or less amounts to repeating previous decisions without renewed voting) limited this computational overhead.

Low-knowledge algorithm control is an approach by Carchrae and Beck [CB04, CB05] to build reactive algorithm portfolios for combinatorial optimization problems. Assuming all algorithms in the portfolio to be anytime algorithms that continuously improve a lower bound on solution quality, it distinguishes itself from other approaches by using the development of solution quality for each of the algorithms as its sole feature. This makes it very general since it does not depend on any information about the domain under consideration and the particular algorithms involved. It starts off by running each algorithm in the portfolio for a given amount of time. Subsequently, it prioritizes the algorithms according to how much they improved their respective incumbent solution in the past, which can be seen as a type of learning by reinforcement (though it is quite different from mainstream reinforcement learning). This process is iterated during the whole course of the search, such that one never solely commits to a single algorithm[7]: an initially strong algorithm will eventually be assigned less runtime once it yields diminishing returns. This robustness paired with the algorithm's generality make it an extremely promising candidate for deployment in practice. Obviously, however, this general approach cannot perform as well as specialized algorithms that take into account the domain and algorithms under consideration. It would be very interesting to extend the algorithm with the capability to use additional domain knowledge if this is known anyways – this could lead to the best of both worlds: a well performing general algorithm whose performance increases once domain-knowledge is taken into account.

The local search community has developed a great variety of approaches for adaptation in local search algorithms, some of which can be seen as tackling the online algorithm configuration problem. The *reactive search* framework by Battiti and Brunato [BB05] uses a history-based approach to decide whether the search is trapped in a small region of the search space, and make a diversification move more likely when trapped. For example, in reactive tabu search, the tradeoff between intensification (more intensely searching a promising small part of the search space) and diversification (exploring other regions of the search space) is made via the tabu tenure, the number of steps for which a modified variable is tabu, that is, cannot be changed again after a modification. When search stagnation is detected, reactive tabu search increases the tabu tenure exponentially, and otherwise slowly decreases it.

A very similar mechanism is used in an adaptive noise mechanism for WalkSAT by Hoos [Hoo02]. Instead of the tabu tenure, Adaptive WalkSAT controls its noise parameter. The noise is increased if no improvement in the objective function value has been observed for too long a time, and it is decreased otherwise. Adaptive WalkSAT Novelty$^+$, introduced in 2002, is still amongst the best performing local search algorithms for SAT.

A similar reactive variant has also been implemented for the SAPS algorithm by Hutter et al. [HTH02] (the algorithm we use for our experiments in this paper). Reactive SAPS, or RSAPS, adapts the probability of performing a smoothing step, where smoothing corresponds to an intensification of the search. Since the optimal parameter setting may change during the course of the search, in principle, this strategy has the potential to achieve better performance than any fixed parameter

[7] Another advantage of this approach is that it lends itself nicely for parallelization. Iterated sequential runs of all algorithms could also be implemented as threads with different priority.

setting. In practice this is true for some instances, but overall, SAPS still shows more robust performance with its default parameter settings than RSAPS.

Let us conclude this discussion of related work by reiterating that our present work on instance-specific algorithm configuration addresses a problem related to, but more powerful than the default parameter configuration problem covered in Section 2.1. For this purpose, we use technology similar to the one used for the instance-specific algorithm selection problem tackled by the approaches in Section 2.2. This is less powerful but more general than the *online* algorithm selection or configuration approaches covered in this Section. We hope to extend our probabilistic machine learning approach to this setting in future work.

3 Empirical hardness models

In this section, we describe the concept of empirical hardness models as introduced by Leyton-Brown et al. [LBNS02, NLBD$^+$04]. In subsequent sections, we extend this research in various directions.

Leyton-Brown et al. [LBNS02, NLBD$^+$04] use a supervised Machine learning approach to predict the runtime of various algorithms for combinatorial auctions and SAT solving. In an offline training phase, each algorithm A of interest is run on a representative *training set* of problem instances $\{s_1, \ldots, s_N\}$. For each instance s_n, A's runtime r_n^A is recorded and a set of so-called *features* $\boldsymbol{x}_n = [x_{n1}, \ldots, x_{nK}]^T$ is computed. These features are domain-dependent and can comprise a large variety of instance characteristics. For example, in the SAT domain, [NLBD$^+$04] employs 91 features, ranging from simple ones (such as the number of variables, clauses and their ratio), to more complex ones, such as an estimate of the size of the DPLL search space [LL98] and local search probes.

After collecting features $\boldsymbol{x}_n$ and runtime r_n^A for each training instance s_n, a function $f^A(\cdot)$ can be learned that, given the features $\boldsymbol{x}_n$ of an instance, approximates A's runtime r_n^A. This function $f(\cdot)$ can also be applied to the features of yet unseen instances s_{N+1} to yield a prediction of the time A would take to solve it. The art of machine learning is to learn functions on the training set that generalize well to unseen instances (instances in the *test set*). [LBNS02, NLBD$^+$04] refer to the predictive function $f^A(\cdot)$ as an *empirical hardness model* for the algorithm A, since it models the empirical hardness of various instances for the algorithm.

In portfolio design based on empirical hardness models [LBNS02, NLBD$^+$04], the task is, given a portfolio of algorithms $\mathcal{A} = \{A_1, \ldots, A_P\}$ to pick the best algorithm $A \in \mathcal{A}$ on a per-instance basis. In order to achieve this, the above methodology is applied for every algorithm A in the portfolio. In an offline training phase, a separate function $f^A(\cdot)$ is learned for each algorithm A, that, given the features $\boldsymbol{x}_n$ of an instance, approximates A's runtime r_n^A on that instance. After these functions have been learned, they can easily be used to pick the presumably best algorithm for solving a previously unseen problem instance s_{N+1}. First, the features $\boldsymbol{x}_{N+1}$ are computed; then, for each algorithm A, the function f^A is evaluated on the instance features $\boldsymbol{x}_{N+1}$, and the algorithm A^* with minimal predicted runtime $f^{A^*}(\boldsymbol{x}_{N+1})$ is chosen to solve the problem.

Different machine learning algorithms can be employed for this runtime prediction problem. Although supervised classification algorithms can be employed to classify runtime into several bins (e.g., short, medium, and long), this approach has important shortcomings, which are discussed in detail in [LBNS02, NLBD$^+$04]. The more intuitive alternative approach employed in [LBNS02, NLBD$^+$04] is based on supervised regression. A large variety of regression algorithms from the Machine Learning literature can be used for this problem, each with its own advantages and disadvantages. [LBNS02, NLBD$^+$04] choose to employ the approach of linear regression, which is attractive due to a very low computational complexity for training and prediction, as well as its conceptual simplicity and ease of implementation. Its main disadvantage is a comparably low predictive accuracy which may be surpassed by more complex algorithms – however, Kevin Leyton-Brown stated in personal communication that they tried more complex algorithms, to no avail. Since we also restrict ourselves to (Bayesian) linear regression algorithms in this paper, we cover linear regression in some detail in the following section.

3.1 Performance prediction with linear regression

This section covers the basics of linear regression and ridge regression, and will set the stage for our subsequent Bayesian linear regression approach. This is standard material covered in many articles and textbooks. We nevertheless include it here in order to yield a self-contained report that is comprehensible without prior exposure to Machine Learning. Our exposition is mainly based on [Bis06].

[NLBD$^+$04] uses a simple Machine Learning technique called *linear regression* that restricts the learned predictive functions $f(\cdot)$ to be linear:

$$f_{\boldsymbol{w}}(\boldsymbol{x}_n) = w_0 + \sum_{k=1}^{K} w_k x_{nk},$$

where $\boldsymbol{w} = [w_0, \ldots, w_k]^T$ are free parameters of the function and we make the function's dependence on these parameters explicit by the subindex $\boldsymbol{w}$. Note that $f_{\boldsymbol{w}}(\cdot)$ is linear in both the features $\boldsymbol{x}_n$ and the parameters $\boldsymbol{w}$. This very simple model may not be flexible enough to do accurate predictions since linear functions of the features are not very expressive. However, the beauty of linear regression is that it doesn't actually require the target function to be linear in the features – all that it requires is *linearity in the parameters* $\boldsymbol{w}$. Hence, we can introduce a vector of so-called *basis functions* $\boldsymbol{\phi} = [\phi_1, \ldots, \phi_D]$ which can include arbitrarily complex functions of *all* features $\boldsymbol{x}_n$ of an instance s_n.[8] The linear regression model is then formulated as

$$f_{\boldsymbol{w}}(\boldsymbol{x}_n) = w_0 + \sum_{d=1}^{D} w_d \phi_d(\boldsymbol{x}_n).$$

Note that the simple case of linear functions of the features is just a special case of this general formulation: here, the number of basis functions D equals the number of features K, and the dth basis function just picks the dth feature: $\phi_d(\boldsymbol{x}_n) = x_{nd}$. Also note that the parameter w_0 is not multiplied by a basis function. It serves as an offset (or bias) parameter that is implicitly multiplied by a "dummy" feature that is constantly 1. If we introduce a constant basis function $\phi_0 = 1$, we can write

[8] Linear regression with basis functions is also sometimes referred to as *basis function regression.*

the linear regression model more compactly as

$$f_{\boldsymbol{w}}(\boldsymbol{x}_n) = \sum_{d=0}^{D} w_d \phi_d(\boldsymbol{x}_n) = \boldsymbol{w^T} \boldsymbol{\phi}(\boldsymbol{x}_n),$$

where $\boldsymbol{\phi}(\boldsymbol{x}_n) = [\phi_0(\boldsymbol{x}_n), \ldots, \phi_D(\boldsymbol{x}_n)]^T$. Remember, that in the context of empirical hardness models we want the function $f^A_{\boldsymbol{w}}(\boldsymbol{x}_n)$ to be a good predictor for the runtime r^A_n of algorithm A on problem instance s_n. This is achieved by fitting the free parameters $\boldsymbol{w}$ such that $f^A_{\boldsymbol{w}}(\boldsymbol{x}_n) \approx r^A_n$ for all instances s_n in the training set. More precisely, $\boldsymbol{w}$ is set such as to minimize some *loss-function.* The standard choice for this is mean squared prediction error (MSPE) on the training set:

$$loss_{ls}(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} (f^A_{\boldsymbol{w}}(\boldsymbol{x}_n) - r^A_n)^2, \tag{1}$$

where the index *ls* stands for least squares. The minimization of this function can be performed analytically as follows, leading to the globally optimal parameter vector $\boldsymbol{w}_{ls}$. Taking the gradient of $loss_{ls}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$ and equating to zero yields the equation

$$\sum_{n=1}^{N} r^A_n \boldsymbol{\phi}(\boldsymbol{x}_n)^T - \boldsymbol{w^T} \left(\sum_{n=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_n) \boldsymbol{\phi}(\boldsymbol{x}_n)^T \right) = 0.$$

Solving this for $\boldsymbol{w}$ directly yields the so-called *normal equations* for the least squares problem:

$$\boldsymbol{w}_{ls} = (\boldsymbol{\Phi^T \Phi})^{-1} \boldsymbol{\Phi^T r}^A, \tag{2}$$

where $\boldsymbol{r}^A = [r^A_1, \ldots, r^A_N]^T$ and we stacked the $D+1$ basis functions for all N training instances into the so-called *design matrix* $\boldsymbol{\Phi}$:

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^T \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^T \end{bmatrix}.$$

Thus, after a bit of algebra, finding the parameter vector $\boldsymbol{w}_{ls}$ of a linear model that minimizes MSPE on the training set comes down to evaluating the term $\boldsymbol{w}_{ls} = (\boldsymbol{\Phi^T \Phi})^{-1} \boldsymbol{\Phi^T r}^A$ (which can be implemented in 1 line of Matlab code). This process is also referred to as *training* the linear model. The computational complexity of this training procedure is very small: it is dominated by the cost of multiplying the DxN matrix $\boldsymbol{\Phi^T}$ by the NxD matrix $\boldsymbol{\Phi}$ (which will take time $O(D^2N)$) and by the inversion of the DxD matrix $\boldsymbol{\Phi^T \Phi}$ (which will take time $O(D^3)$). When reporting results, it is often convenient to report the square root of MSPE; this is called the root mean squared (prediction) error, RMSE.

At test time, A's runtime on a yet unseen problem instance s_{N+1} can be predicted by simply evaluating the learned function $f^A_{\boldsymbol{w}}(\cdot)$ at the features $\boldsymbol{x}_{N+1}$ of the new instance. Since $f^A_{\boldsymbol{w}}(\boldsymbol{x}_{N+1}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_{N+1})$, this evaluation simply computes the inner product of two $(D+1)$-dimensional vectors, which takes time $O(D)$.

One problem of standard linear least squares regression is that $\boldsymbol{w}_{ls}$ may contain excessively large weights. These large weights are the results of fitting some of the noise in the training data (also referred to as *overfitting*) and will lead to poor generalization on new unseen test data. This problem can be easily dealt with by minimizing an alternative loss function

$$loss_{ridge}(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{w}}^{A}(\boldsymbol{x}_n) - A_n)^2 + \lambda \boldsymbol{w}^T\boldsymbol{w}, \tag{3}$$

which penalizes large parameter values by adding a *regularization term* $\lambda\boldsymbol{w}^T\boldsymbol{w}$.[9] Conveniently, the global optimum of this modified loss function can still be found analytically. This is done in what is called *ridge regression*. In direct analogy to the standard result for linear regression, setting the gradient of $loss_{ridge}(\boldsymbol{w})$ to zero and solving for $\boldsymbol{w}$ leads to the ridge solution

$$\boldsymbol{w}_{ridge} = (\lambda I + \boldsymbol{\Phi}^{\boldsymbol{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\boldsymbol{T}}\boldsymbol{A}, \tag{4}$$

where I denotes the $(D+1)$-dimensional identity matrix. Note that $\boldsymbol{w}_{ridge}$ is almost identical to $\boldsymbol{w}_{ls}$, with the only difference that $\boldsymbol{w}_{ridge}$ adds a constant λ to the diagonal of $\boldsymbol{\Phi}^{\boldsymbol{T}}\boldsymbol{\Phi}$ before inverting it. Another frequently used motivation for adding a small constant λ to the diagonal of the matrix is that there are otherwise numerical problems with the inversion. In practice, the performance of ridge regression depends strongly on the actual value of λ, with too low values leading to overfitting (like in standard linear regression) and too large values leading to overly flat functions that are not flexible enough to fit the data. Thus, in practice, the regularization parameter λ must be carefully chosen via cross-validation.

3.2 Feature engineering for runtime prediction

In the last section, we stressed that linear regression functions only have to be linear in the parameters and that arbitrary basis functions of all features can easily be used. Leyton-Brown et al. [LBNS02, NLBD$^+$04] exploit this and study several alternative sets of basis functions. In the simplest case, they employ one basis function per feature and thus learn functions that are linear in the features. They also study the use of additional basis functions that consist of the pairwise products of all input features. To be more precise, given K input features, the simple model (which is linear in the input features) would employ K basis functions of the form $\phi_k(\boldsymbol{x}_n) = x_{nk}$. The second type of model would additionally employ $\binom{K}{2}$ basis functions $\phi_d(\boldsymbol{x}_n) = x_{ni}x_{nj}$ for all pairs of input features $\langle i, j\rangle$. Since this second model effectively learns a quadratic function of the features, [LBNS02, NLBD$^+$04] refer to it as a "quadratic model", whereas they call the first type of model "linear model". We will refer to both types of regression models as linear regression, just with different basis functions.

Leyton-Brown et al. [LBNS02, NLBD$^+$04] further employ extensive *feature selection* methods in order to pick the most predictive basis functions from a large

[9] This loss function penalizes large weights quadratically. An alternative we do not cover here it to penalize the absolute (non-squared) values of $\boldsymbol{w}$. Such a penalty is chosen in *lasso regression*, which leads to many of the parameters going to zero, effectively performing feature selection for free. We plan to compare this approach against our current approaches for feature selection in future work.

set.[10] The easiest example for feature selection is *forward selection*, which starts off with an empty feature set and greedily adds features that maximally improve performance.

In order to prevent overfitting, performance of a feature subset is measured by training a model on the training set and evaluating its resulting root mean squared error (RMSE) on a separate validation set. This validation set is usually taken to be some held out part of the training set that is not used for training. In what is called *cross validation*, the training data is partitioned into K *folds* of roughly the same size, and this procedure is performed K times, each time using all but one fold as training set and the remaining fold as validation set. The cross validation RMSE can then be computed as the mean of the RMSEs for each of the folds; the standard deviation of the RMSE in the folds can also be computed. Finally, the special case where every data point is a fold by itself is called *leave-one-out* cross validation. Compared to using only one validation set, cross validation has the advantage of making better use of the possibly small training set, but it may also be prone to overfitting if used to optimize several parameters.

[LBNS02, NLBD$^+$04] employ a simple fixed split of the training set into training and validation set, and demonstrate that using small subsets of basis functions already yields results comparable to using all of them. We observe a similar phenomenon for our data.

4 Feasibility of empirical hardness models for SLS algorithms

In this section, we show that empirical hardness models are not limited to complete solvers but can also be learned for stochastic local search (SLS) algorithms. This has never been demonstrated before, and indeed, Kevin Leyton-Brown stated in personal communication that they conducted preliminary (and unpublished) inconclusive experiments that rather suggested the opposite. We start this section by introducing the SAT domain and the SLS algorithm SAPS, and subsequently show that we can predict the median runtime of SAPS on unseen instances quite accurately.

4.1 Stochastic Local Search for SAT

So far, we have motivated our discussion with very general problem scenarios from Constraint Programming (CP). While we see our approach as generally applicable, we focus our experiments on a much smaller domain, namely a single stochastic local search (SLS) algorithm for solving the propositional satisfiability (SAT) problem. This deserves some justification. Firstly, we restrict our initial experiments to the SAT domain since SAT is the prototypical and best studied $\mathcal{NP}$-hard problem [HS04, TH04]. While SAT algorithms tend to be simpler and cleaner than algorithms for other $\mathcal{NP}$-hard problems, SAT exhibits characteristics representative for many combinatorial problems. Moreover, there exist an abundant number of freely

[10] Since these feature selection techniques operate on the basis functions directly, the term "basis function selection" may be more appropriate, but we stick with the term feature selection since it is the most commonly used term.

available SAT instances, which is central in learning empirical runtime models with the methodology of [LBNS02, NLBD$^+$04]. Lastly, empirical runtime models have already been studied for SAT, such that strong predictive features already exist and need not be engineered from scratch [NLBD$^+$04].

We chose the domain of SLS algorithms [HS04] since empirical hardness models have thus far not been learned for this important domain and we wanted to study their applicability. Furthermore, SLS algorithms tend to be parameterized algorithms whose performance depends crucially on a good parameter setting. An automated way of parameter tuning will be an important contribution that promises to significantly relieve researchers by handling the necessary engineering work in an automated and principled fashion. Finally, we chose the SAPS algorithm [HTH02] as one particular instance of SLS algorithms since for several reasons. Firstly, SAPS is still amongst the state-of-the-art algorithms for many hard random SAT problems [HS04]. Secondly, SAPS has three continuous parameters that interact closely. Finally, we have already seen evidence in the past that the optimal parameter setting of SAPS is instance-dependent [HTH02], a fact that we exploit by setting its parameters conditional on the instance characteristics.

SAPS is a dynamic local search (DLS) algorithm, that means it dynamically modifies the evaluation function during the progress of the search. Like most DLS algorithms for SAT, SAPS is a clause weighting algorithm, i.e. it assigns a weight to each clause and at each step aims to minimize the sum of the weights of unsatisfied clauses. SAPS starts off by assigning a weight of one to each clause. After this, it iterates weighted search steps and weight update steps. First, a series of weighted search steps leads SAPS into a local minimum of the evaluation function, i.e. a search state without neighbours with lower evaluation function value. SAPS's neighbourhood relation is a simple one-flip neighbourhood, i.e. two search states are neighbours if and only if they differ in the assignment to exactly one variable. Once in a local minimum, SAPS performs a weight update step, in which the weights of all unsatisfied clauses are multiplied by a factor $\alpha > 1$ (*scaling*). Afterwards, with some probability P_{smooth}, all clause weights are *smoothed* towards their mean by setting them to $w \leftarrow w \cdot \rho + (1 - \rho) \cdot \overline{w}$, where ρ is a factor between zero and one. The three quantities α, ρ, and P_{smooth} are the parameters, and their default setting is $\langle \alpha, \rho, P_{smooth} \rangle = \langle 1.3, 0.8, 0.05 \rangle$. Intuitively, there is a strong interconnection between all these parameters. When α is high, the differences in clause weights grows quickly, whereas when ρ is small they are smoothed out quicker (in the extreme $\rho = 0$ all differences in clause weights vanish in a single smoothing step). The connection between ρ and P_{smooth} is even stronger. As discussed in [HTH02], a decreased smoothing probability P_{smooth} can be countered by a stronger smoothing (i.e., a lower ρ). Due to this tight connection between P_{smooth} and ρ, we chose to fix P_{smooth} to its default value 0.05 and only set α and ρ to their most promising value for each instance automatically.

Since our experiments are restricted to the SAPS algorithm which cannot prove unsatisfiability, we only employ satisfiable SAT instances. All benchmark instances we employ can be found online. They represent three fundamentally different classes of instances:

`SAT04-random` comprises all solvable random instances from the SAT04 competition. In the competition, SAPS solved 144 of the 150 instances, placing second after a robust variant of WalkSAT Novelty$^+$ with an adaptive setting of the noise parameter (AdaptNovelty$^+$).

`BIGMIX` comprises 139 instances that were handpicked from the online repository SATLIB[11]. They comprise uniform random instances, random instances with controlled backbone size, backbone-minimal sub-instances, as well as encodings of blocksworld and logistics planning, flat graph colouring, factorization, inductive inference, and parity problems. One constraint in choosing instances was that they be satisfiable and solvable by SAPS with default parameters within one hour.

`UF100` comprises 1000 hard uniform 3-SAT instances from the phase transition region. All instances in this set have been generated using *exactly* the same random instance generator. Nevertheless, they differ in hardness (average time typical solvers take to solve them) by up to two orders of magnitude. This is the hardest problem class for an approach based on runtime prediction because all instances are structurally very similar.

We now shift our attention from the data sets of interest to the features of individual instances that we employ in order to predict empirical hardness on a per-instance base.

`BasicFeatures` The 46 features in this feature set are a subset of the 91 SAT features used in [NLBD$^+$04]. They range from simple features (such as the number of variables, clauses and their ratio) to more complex estimates of the size of the DPLL search space [LL98] and some local search probes. We only left out features which were constant for all instances, whose computation took an unreasonably long time for some large instances (such as some cluster graph-based features), or whose computation often failed alltogether (such as the features based on linear programming).

`SLSF` This feature set comprises all the features in `BasicFeatures`, and twelve more that are computed from multiple trajectories of the SAPS algorithm. There are already some local search features in `BasicFeatures` that compute characteristics of SAPS. We do not see a fundamental difference between our features and those. Our intuition was, however, that additional features from local search probes may be quite beneficial for predicting the runtime of local search algorithms. Thus, we put in all additional features we expected to be of potential use. The feature selection mechanism is then expected to detect the really important features and thus, additional informative features should at least not be harmful. The cost we pay for additional features is a slightly longer training time (since the feature selection takes longer) and the cost of computing the features for each instance.

Features based upon the SAPS algorithm may be more beneficial to predicting SAPS behaviour than the behaviour of any other algorithm. They are thus arguably not as general as most other features in `BasicFeatures`. However, they have two advantages: firstly, since they may capture some of the dynamic behaviour of the algorithm of interest on the particular instance of interest, they are inherently more powerful than features which do not execute the

[11] `http://www.satlib.org`

algorithm. Secondly, the dynamic features we compute are quite general in that they can be computed for any local search algorithm for SAT. In order to compute these twelve additional features, we run SAPS $N = 10$ times for $M = 1000$ search steps each and gather the following statistics from each run:

Mean age of flipped variables The *age* of a variable is the number of steps for which it has not been flipped, and if a variable has never been flipped during a trajectory, its age is the total number of steps performed so far. The statistic we use to describe a run is the age of variables at the time when they are flipped, averaged over all flips of variables. Intuitively, when the mean age of flipped variables is low, then the search concentrates on flipping a few variables often, that is, stays in a small part of the search space. When it is large, intuitively, the coverage of the search space is larger.

Percent local minima This statistic is just the number of local minima encountered during a search trajectory, divided by the total number of search steps. The same local minimum is counted multiple times if it is visited multiple times.

Correlation length A trajectory $s_1, \ldots, s_m$ of m search states defines a sequence of objective function values $g_1, \ldots, g_m$. For such a sequence with m values, Hoos and Stützle[HS04] define the *(empirical) autocorrelation function* as

$$r(i) = \frac{1/(m-i) \cdot \sum_{k=1}^{m-i}(g_k - \bar{g}) \cdot (g_{k+i} - \bar{g})}{1/m \cdot \sum_{k=1}^{m}(g_k - \bar{g})^2},$$

where $\bar{g}$ is the mean of $g_1, \ldots, g_m$. The value $r(i)$ is the empirical correlation coefficient between the objective function values i steps apart in the search trajectory. [HS04] states that for random trajectories, this function typically decays exponentially as $r(i) = e^{-i/l}$, where l is the (empirical) correlation length (also known as autocorrelation length) and can be defined as $\frac{1}{ln|r(1)|}$ if $r(1) \neq 0$. We use this quantity l as a statistic despite the fact that our trajectories are highly non-random. If it is high, this is an indication for a relatively smooth cost surface (even after taking several steps, the objective function has not changed too much), whereas if it is low the landscape is fairly rugged (i.e., even a few steps can lead to large changes in objective function value).

Scaled version of correlation length We also include a scaled version of the autocorrelation length as a feature, namely $s = 1/(1 - l)$.

For each of these four statistics, we compute the mean and median value across the N runs, as well as the variation coefficient (standard deviation divided by mean). These 12 values are added as additional features.

`LLSF` stands for "large local search features", and this set contains the same features as `SLSF`, with the one difference that the twelve dynamic features are computationally less efficient but more precise estimates of the quantities under question. Instead of 10 runs of 1000 search steps each in `SLSF`, `LLSF` executes 100 runs of 10000 steps each.

The approach of runtime prediction we discuss in this report may further benefit from some preprocessing on the SAT instances. For example, preprocessing has been combined with SLS algorithms for SAT in [APSS05]. Next to making some instances easier to solve, this preprocessing may reduce the problem instance to its combinatorial core, rendering our automatically derived features more predictive of problem hardness. We expect a future study of such effects to be quite illumniating.

4.2 Feasibility of runtime prediction for SLS algorithms for SAT

In this section, we evaluate to which extent the previous approach of [LBNS02, NLBD$^+$04] generalizes to predict the runtime of SLS algorithms for SAT, in particular the SAPS algorithm. Remember that we only use satisfiable SAT instances, so what we really study here is the hardness of the model finding variant for SAT.

In experiment 1, we demonstrate that empirical hardness models yield good predictive performance for the median runtime of 1000 SAPS runs for each of our data sets. This simply applies the linear regression software of Leyton-Brown et al. [LBNS02, NLBD$^+$04] as an off-the-shelf algorithm with default parameters. Subsequently, we employ cross validation to tune the regularization parameter δ for our domains to further improve results; experiments 2 and 3 do this for linear and quadratic models, respectively. These two experiments aim at finding good default settings for the regularization parameter and at comparing the power of linear and quadratic models in our domain. Next, experiment 4 studies the impact of different data normalizations, solely in order to make sure that we use the best possible normalization for our domain. Finally, experiment 5 studies the improvements in predictive performance when we use some additional local search features. We trade off predictive accuracy and time needed for the feature computation, and pick one instance feature set that will be used in all subsequent experiments in this report.

Experiment 1 (Applicability of runtime prediction for SLS algorithms)**.** In this experiment, we study the applicability of runtime prediction for SLS algorithms for SAT, in particular for predicting the runtime of the SAPS algorithm with default parameter setting. We illustrate the predictive performance on our data sets `SAT04-random`, `UF100`, and `BIGMIX`. Each data set was divided into three subsets: a training set comprising 65% of the instances, a validation set with 15% of the instances, and a final test set holding 20% of the instances. Since SAPS is a randomized algorithm with exponentially distributed runtimes, we only aim to predict the logarithm of median runtime over a number of N runs per instance. For each data set, we thus ran SAPS N times on each instance of the training set, and learned a function that predicts their median runtime given the instance's features (here, we employ feature set `SLSF`).

The resulting full models whose predictive performance we report on employs all features that are not constant in the training set. We also evaluated the performance of models learned on subsets of features. For each data set, in order to construct a good feature subset, we ran forward selection on the training set and evaluated the performance of the learned models on the validation set. We then used the subset that achieved lowest overall validation set RMSE. For each data set, features that were constant in the training set were dropped; for the data sets

SAT04-random, UF100, and BIGMIX, there were six, twelve, and three such constant features, respectively.

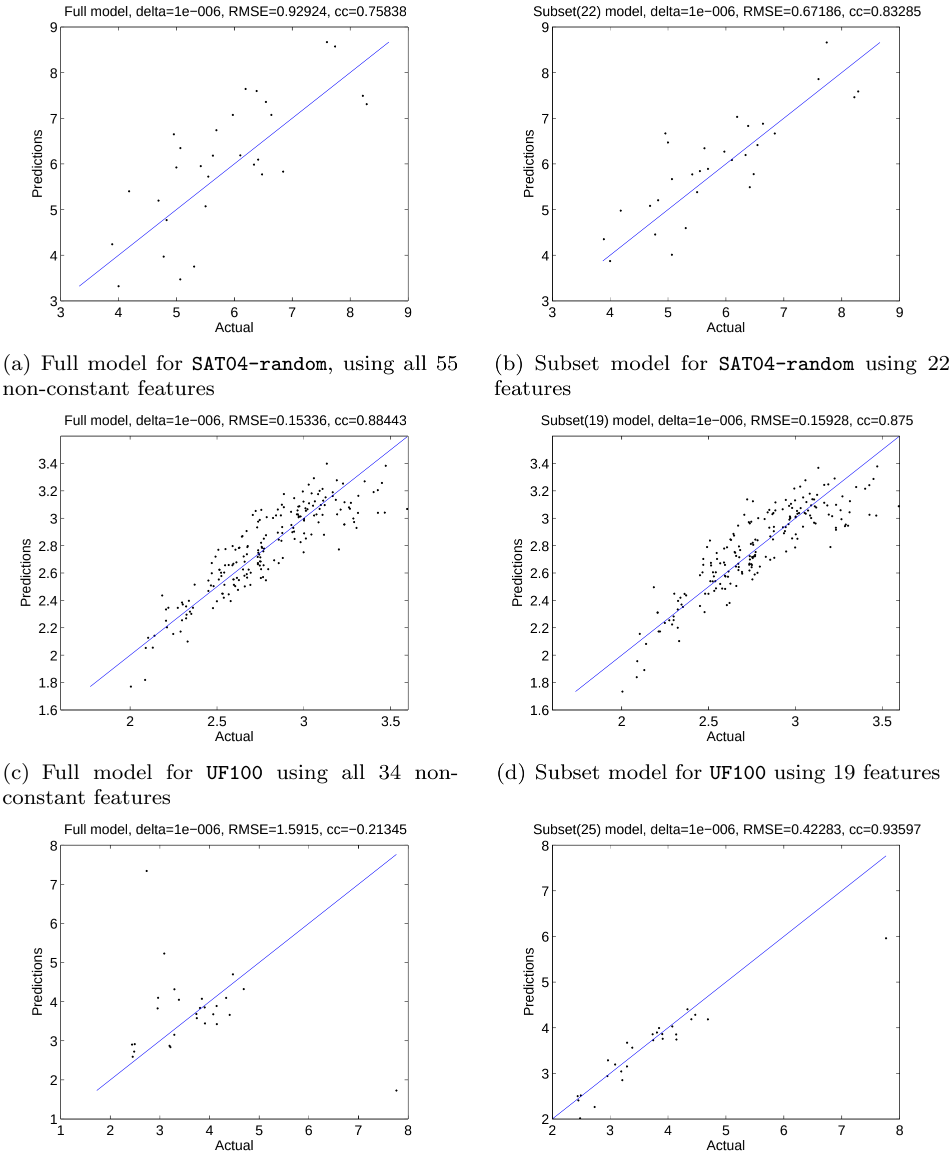
For data set SAT04-random, Figure 1(a) shows the predictive performance on the testset obtained with a linear model employing the remaining 55 features (we refer to models using all non-constant features as *full models*). Figure 1(b) shows the performance of a linear model learned on an automatically constructed subset of 22 features. Note that this *subset model* achieves significantly better accuracy than the full model (RMSE 0.93 vs. 0.67). Also notice the strong correlation (correlation coefficients 0.75 and 0.83) between actual and predicted runlength which demonstrates our ability to predict median runlengths of SLS algorithms based on previous performance. Although the median SAPS runlengths vary by over four orders of magnitude across the test set, predictions of the subset model are typically well within an order of magnitude of the actual runlength.

We now turn our attention to the data set UF100. All instances in UF100 have the same number of variables and clauses and were generated with the same instance generator. Thus, it does not come as a surprise that twelve out of the 46 features were constant in the training set, and were thus removed. Although generated with the same generator, hardness of the instances in in UF100 still varies by about two orders of magnitude. As can be seen in Figures 1(c) (full model) and 1(d) (subset model), the absolute predictive error is very low (RMSEs around 0.155), but we observe a slight systematic trend that hard instances are predicted to be easier than they actually are. Also note that for this data set, there is almost no difference in the performance of the full model and the subset model.

Finally, we turn to what is arguably our hardest data set. BIGMIX is fairly small and on top of this contains very heterogenous instances. It is thus to be expected that runtime prediction will not be easy for this data set. Figures 1(e) and 1(f) show the predictive accuracy of a full model and a subset model for this domain. Note that this data set exemplifies overfitting in the full model which results in very poor predictive performance, and that the subset model achieves much better accuracy (full models can also achieve high performance, but need a stronger regularization – we will revisit this issue in experiment 2). Notice that the performance of the subset model is indeed quite good. Although the predictive accuracy for the hardest instance is not good, it is clearly recognized as much harder than the rest. Based on the very sparse training set, we cannot hope to do much better performance. Also, the overall RMSE of the subset model (0.42) is remarkably low and the correlation coefficient between predicted and actual runlength is very high (0.93).

Since we saw evidence of overfitting in the previous experiment 1 (especially for the data set BIGMIX, and since overfitting in linear regression can be countered by an increased regularization, in the next experiment we evaluate the dependency of cross validation and test error on the regularization constant δ.

Experiment 2 (Varying the regularization constant δ)**.** Here, we employ cross-validation to study the impact of the regularization constant δ on predictive performance for each of our data sets. For this purpose, we merge training and validation set. In K-fold cross validation, this set is then divided into K folds of equal size. Each of these folds is used as a “test set” in turn, with the rest of the instances divided into training and validation set at random in a ratio of 70-30. We report



(a) Full model for `SAT04-random`, using all 55 non-constant features

(b) Subset model for `SAT04-random` using 22 features

(c) Full model for `UF100` using all 34 non-constant features

(d) Subset model for `UF100` using 19 features

(e) Full model for `BIGMIX` using all 34 non-constant features

(f) Subset model for `BIGMIX` using 25 features

Figure 1: Predicted runlength vs. actual runlength for SAPS, using full models and subset models on our three data sets. All models use feature set `SLSF` and regularization parameter $\delta = 10^{-6}$. This plot belongs to experiment 1.

the mean and variance of RMSE for each value of the regularization constant δ, and for both the full and the subset model (as before, the size of the subset model is adaptively chosen.). We also report the real test error (using the original training, validation, and test sets) for each value of δ, but we do not tune with respect to this value in order not to peek at the test set.

The results of this experiment are given in the upper half of Table 1. We observe that full models favour higher regularization parameters (around $\delta = 1$) than the subset models, which show best performance for values of δ around 10^{-2}. Also note that the subset models are very robust w.r.t. δ, whereas for two of the three data sets(`SAT04-random` and `BIGMIX`) the full model only performs good for a small range of δ. Due to this sensitivity of full models, we only consider (automatically determined) subset models in subsequent experiments. We will also keep the regularization constant of linear models fixed at a value of $\delta = 10^{-2}$.

Experiment 3 (Varying the regularizer for quadratic models)**.** This experiment repeats experiment 2 using quadratic basis functions. Again, we vary the regularization parameter δ and for each data set report performance of the full quadratic model and of an automatically determined subset model.

The result is presented in the lower half of Table 1. In particular, we notice that predictive accuracy does not increase significantly when compared to the model that only uses linear basis functions. Since the feature selection process and cross-validation are quite computationally expensive, we disregard the use of quadratic basis functions from now on.

Experiment 4 (Data normalization)**.** In this experiment, we study the effects of different normalizations on the data. One common normalization for continuous data is to make it have zero mean and unit variance by subtracting the mean and dividing by the standard deviation. A second normalization is a transformation of the data to the unit interval [0,1] by subtracting the minimum and then dividing by the maximum. Using linear basis functions, we employ leave-1-out cross validation to study the effects of these normalizations compared to not normalizing the data at all. Table 2 shows the results. The predictive accuracy achieved with the different normalizations is almost indistinguishable, except in the case of data set `BIGMIX` without normalization. In that case, the cross validation RMSE is huge, which is due to a few very badly predicted data points. Thus, we conclude that, for our data sets, normalization is not crucial but can prevent infrequent large errors. Normalization to mean zero and unit variance is just as good as normalization to the unit interval, and we perform the former one in subsequent experiments.

Experiment 5 (Importance of the employed features)**.** In this experiment, we evaluate how predictive accuracy varies if we use different feature sets. For each combination of data set and feature set, we ran cross-validation in order to find the best value of δ when an automatically found subset of the features were used. Table 3 shows the results. Interestingly, performance with feature set `SLSF` is not much better than with feature set `BasicFeatures`, whereas using feature set `LLSF` leads to big improvements. Only for data set `UF100` is feature set `SLSF` significantly better than feature set `BasicFeatures`, but `LLSF` performs even much better. Actually, for this last data set `UF100`, we do not entirely trust our newly introduced local search features. Remember that in order to compute the extra features in `SLSF`, we run

δ	SAT04-random			BIGMIX			UF100		
	K	subset model	full model	K	subset model	full model	K	subset model	full model
Linear basis functions									
10^{-8}	22	0.61 ± 0.12 0.67	0.92 ± 0.20 0.93	25	1.30 ± 2.44 0.42	6.09 ± 11.92 2.23	19	**0.15 ± 0.01** 0.16	0.16 ± 0.01 0.15
10^{-5}	22	0.62 ± 0.10 0.67	0.80 ± 0.19 0.93	25	**0.52 ± 0.28** 0.42	3.15 ± 4.20 1.05	17	**0.15 ± 0.01** 0.16	**0.15 ± 0.01** 0.15
10^{-3}	22	0.61 ± 0.13 0.67	0.74 ± 0.15 0.87	19	0.59 ± 0.40 0.43	0.86 ± 0.49 0.88	19	**0.15 ± 0.01** 0.16	0.16 ± 0.01 0.15
10^{-2}	25	**0.59 ± 0.08** 0.67	0.70 ± 0.10 0.80	22	0.56 ± 0.36 0.42	0.66 ± 0.50 0.76	19	**0.15 ± 0.01** 0.16	**0.15 ± 0.01** 0.15
10^{-1}	27	0.63 ± 0.09 0.62	0.65 ± 0.05 0.72	24	0.58 ± 0.28 0.45	0.51 ± 0.24 0.67	22	**0.15 ± 0.01** 0.16	**0.15 ± 0.01** 0.15
10^{0}	14	0.63 ± 0.11 0.60	**0.60 ± 0.08** 0.64	26	**0.52 ± 0.28** 0.52	**0.46 ± 0.27** 0.62	24	0.16 ± 0.01 0.16	**0.15 ± 0.01** 0.15
10^{1}	30	0.90 ± 0.25 0.90	0.92 ± 0.23 0.92	21	0.61 ± 0.30 0.76	0.65 ± 0.32 0.79	34	0.17 ± 0.01 0.17	0.17 ± 0.01 0.17
10^{2}	29	3.10 ± 0.36 3.17	3.11 ± 0.37 3.17	24	1.99 ± 0.29 2.21	2.00 ± 0.28 2.14	25	0.43 ± 0.01 0.40	0.43 ± 0.01 0.41
Quadratic basis functions									
10^{-5}	48	0.86 ± 0.33 0.71	1.22 ± 0.26 1.03	46	1.50 ± 2.19 1.32	1.27 ± 0.57 1.55	80	0.16 ± 0.01 0.16	0.43 ± 0.04 0.44
10^{-3}	71	0.81 ± 0.29 0.76	1.09 ± 0.29 1.03	63	2.64 ± 5.83 0.72	1.88 ± 1.46 1.54	80	0.16 ± 0.01 0.16	0.41 ± 0.04 0.40
10^{-2}	79	0.76 ± 0.13 0.81	1.13 ± 0.30 1.01	70	0.78 ± 0.50 0.44	1.39 ± 0.88 1.46	80	0.16 ± 0.01 0.16	0.34 ± 0.06 0.31
10^{-1}	80	**0.69 ± 0.17** 0.78	0.87 ± 0.24 0.93	80	0.57 ± 0.37 0.76	1.12 ± 1.04 1.12	80	0.16 ± 0.01 0.16	0.22 ± 0.02 0.21
10^{0}	80	0.70 ± 0.18 0.60	0.80 ± 0.12 0.73	80	**0.55 ± 0.25** 0.54	**0.66 ± 0.29** 0.71	80	**0.15 ± 0.01** 0.16	**0.18 ± 0.01** 0.17
10^{1}	80	0.77 ± 0.19 0.65	**0.76 ± 0.14** 0.64	80	0.61 ± 0.40 0.41	1.06 ± 0.81 0.50	80	0.16 ± 0.01 0.16	0.20 ± 0.02 0.21
10^{2}	80	1.16 ± 0.28 1.13	1.06 ± 0.27 1.05	80	1.05 ± 0.19 1.00	0.95 ± 0.18 0.89	80	0.40 ± 0.01 0.40	0.47 ± 0.03 0.46
10^{3}	80	2.22 ± 0.42 2.35	2.06 ± 0.42 2.11	80	2.43 ± 0.31 2.69	2.34 ± 0.29 2.57	80	1.31 ± 0.05 1.29	1.29 ± 0.05 1.28

Table 1: Performance of linear models with linear basis functions (upper part) and quadratic basis functions (lower part) for different values of the regularizer δ. For each data and each value of δ, we give the size K of the subset model, as well as the performance of the subset model and the full model. The performance is measured by three numbers, namely the mean RMSE in 10-fold cross validation ± its standard deviation across the 10 folds (top row), as well as the test error (bottom row). We typeset the best cross validation results for each data set in bold face (the test set cannot be used to tune parameters, we only use it to report results). The upper part of this table belongs to experiment 2, the lower part to experiment 3.

	Mean zero, $\sigma^2 = 1$		Between zero and one		No normalization	
	Full model	Subset model	Full model	Subset model	Full model	Subset model
SAT04-random	0.49 ± 0.35 0.71	0.45 ± 0.34 0.64	0.47 ± 0.36 0.71	0.45 ± 0.34 0.64	0.47 ± 0.34 0.67	0.46 ± 0.35 0.68
BIGMIX	0.36 ± 0.37 0.71	0.45 ± 0.91 0.39	0.36 ± 0.42 0.71	0.41 ± 0.49 0.39	1.29 ± 7.58 0.39	0.96 ± 4.49 0.36
UF100	0.20 ± 0.06 0.20	0.20 ± 0.06 0.20	0.20 ± 0.06 0.20	0.20 ± 0.05 0.20	0.21 ± 0.06 0.21	0.21 ± 0.06 0.21

Table 2: Predictive performance for different normalizations of the data. For each normalized data set, a full linear model and a subset model were learned and evaluated with leave-one-out cross validation. We report the RMSE, averaged over all cross validation folds and its standard deviation, as well as the test set RMSE. The regularization parameter used was $\delta = 10^{-1}$. This table belongs to experiment 4.

	BasicFeatures				SLSF				LLSF			
	δ	N	K	Performance	δ	N	K	Performance	δ	N	K	Performance
SAT04-random	10^{-3}	N	12	0.59 ± 0.10 0.64	10^{-2}	N	25	0.59 ± 0.08 0.67	10^{-2}	N	15	0.48 ± 0.07 0.53
BIGMIX	10^{0}	N	20	0.47 ± 0.30 0.53	10^{0} or 10^{-5}	N	26	0.52 ± 0.28 0.52	10^{-2}	N	20	0.35 ± 0.19 0.29
UF100	10^{-8} to 10^{-2}	N	25	0.21 ± 0.02 0.20	10^{-8} 10^{-1}	N	22	0.15 ± 0.01 0.16	10^{-8} to 10^{0}	N	26	0.06 ± 0.01 0.06

Table 3: Comparison of our different feature sets in terms of performance for all data sets. For each data set and feature set, we ran cross validation in order to choose the best value of δ. For each such combination, we report the total number N of features, the number K of features chosen by the subset model and the performance in terms of cross-validation RMSE mean and standard deviation (upper row) and test set RMSE (lower row). For each δ, the subset of features to use was chosen individually. This table is part of experiment 5.

SAPS 10 times for 1000 steps each. If SAPS successfully solves an instance during this feature computation, it is in theory possible to encode its runtime until solution in the local search features. This would lead to "cheating" in the runtime prediction: first, SAPS solves the instance during the feature computation, say in 500 steps, and then you predict SAPS to take around 500 steps. Since the SAPS runlength for instance in UF100 lies between 100 and 4,000 steps, there is considerable potential for such "cheating" behaviour. This is not the case for the other two instance sets since the runlength there reaches up to many millions of steps (400 million steps for the hardest instance in SAT04-random), such that instances can't usually be solved in the feature computation phase.

At this point, we have to note that the computation of the feature set LLSF (and sometimes also SLSF) did take a long time for some large instances. Because of this sometimes excessive computation, we did not use these new local search features in subsequent experiments. However, we plan to study computationally bounded ways of computing features such as the ones in LLSF. In experiment 5, we have seen conclusive evidence that the new local search features in LLSF significantly improve predictive performance.

5 Automatic parameter setting based on empirical hardness models

In this section, we demonstrate how to extend the previous approach for runtime prediction by Leyton-Brown et al. [LBNS02, NLBD$^+$04] (described in Section 3) to predict the runtime of a single algorithm with various parameter configurations. This generalization allows us to achieve an instance-specific parameter tuning for any algorithm which can significantly outperform its best default configuration.

We have already seen how regression approaches can be employed in order to predict the runtime of a given algorithm A on a given instance s_n. In Section 3, this was done by learning a function that maps s_n's features $\boldsymbol{x}_n$ to A's runtime r_n^A. More specifically, we introduced a vector of basis functions $\boldsymbol{\phi} = [\phi_0, \ldots, \phi_D]^T$ and learned a parameter vector $\boldsymbol{w} = [w_0, \ldots, w_D]^T$ that, when multiplied with the values of these basis functions for instance s_n, yielded an approximation of A's runtime for this instance: $f_{\boldsymbol{w}}^A(\boldsymbol{x}_n) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_n) \approx r_n^A$. We saw that when we stack the $D+1$ basis functions for all training instances $\{s_1, \ldots, s_N\}$ into the Nx$(D+1)$ design matrix Φ and collect all runtimes in the Nx1 target vector $\boldsymbol{r}$, then the parameter vector $\boldsymbol{w}$ of a ridge regression function can be found by Equation (4). This parameter vector yields the optimal approximation $\Phi \boldsymbol{w} \approx \boldsymbol{r}$ in a regularized least squares sense (see Section 3). Thus, all we have to do in the training phase is to determine the appropriate basis functions, build the design matrix Φ and the target vector $\boldsymbol{r}$, and finally invoke Equation (4) to fit the parameters $\boldsymbol{w}$. The test phase is even easier. Given a new instance s_{N+1}, we simply compute its basis functions and multiply them with the learned weight vector $\boldsymbol{w}$ to get the predicted runtime $f_{\boldsymbol{w}}^A(\boldsymbol{x}_{N+1}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_{N+1}) \approx r_{N+1}^A$.

Having reviewed this regression approach, let's start this section with a straightforward application of regression to picking the best parameter configuration for a single algorithm on a per-instance base. This simple application is analogous to the portfolio approach by Leyton-Brown et al. [LBNS02, NLBD$^+$04] (see Section 3): we basically treat an algorithm with P different possible parameter configurations as P different algorithms. This simplistic approach may be beneficially applied to algorithms with a small number of possible parameter configurations, but it does not scale to many parameters or to the more interesting case of continuous parameters. We mainly use this simple version to build intuition, and subsequently move on to the automatic tuning of continuous parameters.

Say, we want to automatically set the parameters of some parametric algorithm A to achieve peak performance on a per-instance base. Further say, there are P possible configurations for A's parameters with finite but possibly large C. (Note that this can always be achieved by discretizing continuous parameters, but also that the number of parameter configurations grows exponentially with the number of algorithm parameters.) We denote algorithm A with parameter configuration c as $A[c]$. In the case of a finite number C of parameter configurations c, the methodology introduced in Section 3 applies directly as follows.

In the training phase, we run $A[c]$ for each possible parameter configuration c on each of the problem instances $s_1, \ldots, s_N$ and collect the runtimes $\boldsymbol{r}^c = [r_1^c, \ldots, r_N^c]^T$. We also compute the features $\boldsymbol{x}_n$ for each instance s_n and stack their basis function values $\boldsymbol{\phi}(\boldsymbol{x}_n)^T$ into the design matrix Φ. We then learn a linear function $f_{\boldsymbol{w}}^c$ that maps features to runtimes, that is, we apply Equation (4) to fit the function's parameter vector $\boldsymbol{w}$. The resulting function is $f^c_{\boldsymbol{w}}(\boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_n)^T \boldsymbol{w}$.

In the test phase, given a new instance s_{N+1}, all that is left to do is to compute the instance features $\boldsymbol{x}_{N+1}$ and the basis function values $\boldsymbol{\phi}(\boldsymbol{x}_{N+1})$, and to evaluate the learned function $f^c_{\boldsymbol{w}}(\boldsymbol{x}_{N+1})$ for each possible parameter configuration c. We can then pick the configuration c^* with the lowest predicted runtime $f^{c^*}_{\boldsymbol{w}}(\boldsymbol{x}_{N+1})$.

However, there are significant shortcomings to this approach. We already hinted on the most important problem, namely that there may be prohibitively many possible parameter configurations. If there are L parameters each of which can take M values there will be $C = M^L$ possible configurations. Especially in the case of continuous parameters, M will be quite large if we want to discretize the parameter space in a reasonably fine-grained fashion.

A large number of parameter configurations C leads to several problems for this approach. Firstly, in order to learn a function $f^c_{\boldsymbol{w}}$ for each possible parameter configuration c, we need to run $A[c]$ for each of the N training instances. For applications with large C, large N or a large average algorithm runtime, this can be prohibitively expensive. Furthermore, at test time one has to evaluate C functions which may be expensive for large C and a large number of basis functions (albeit a single function evaluation in linear regression only has complexity $\mathrm{O}(D)$, where D is the number of employed basis functions). Finally, a last problem of the described simple approach is that it learns independent functions for the C parameter configurations, making it impossible to use a known runtime of $A[c]$ in order to inform the runtime predictions for $A[c']$ for $c' \neq c$.[12]

Our new approach for tuning continuous algorithm parameters naturally takes care of these problems using a very simple method. Instead of learning a separate function $f^c_{\boldsymbol{w}}(\cdot)$ for each possible parameter configuration c, we learn a single function $g_{\boldsymbol{w}}(\cdot,\cdot)$ that has c in its argument list. Given the set of features $\boldsymbol{x}_n$ of an instance s_n and a parameter configuration c, $g(\boldsymbol{x}_n, c)$ will yield a prediction of r^c_n, $A[c]$'s runtime on instance s_n. The main advantage of this approach is that it can generalize to yet unseen parameter configurations as well as to yet unseen instances. The details of this approach are as follows.

In the training phase, for each training instance s_n we run A with a set of parameter configurations $\boldsymbol{c}_n = \{c_{n,1}, \ldots, c_{n,k_n}\}$ and collect the corresponding runtimes $\boldsymbol{r}_n = [r_{n,1}, \ldots, r_{n,k_n}]^T$. We also compute s_n's features $\boldsymbol{x}_n$. The key change to the previous approach is that now the parameter configuration is treated similarly to the features. We define a new set of basis functions (still called $\boldsymbol{\phi}$) whose domain now consists of the cross product of features and parameter configurations. For each instance s_n and parameter configurations $c_{n,j}$, we will have a row in the design matrix that contains $\boldsymbol{\phi}(\boldsymbol{x}_n, c_{n,j})^T$, that is, the design matrix now contains n_k rows for every training instance. The target vector $\boldsymbol{r} = [\boldsymbol{r}_1^T, \ldots, \boldsymbol{r}_N^T]^T$ just stacks all the runtimes on top of each other.

[12] Note that this contradicts our intuition gathered from manual parameter tuning. Think of an algorithm A with a single continuous parameter b: if we observe the runtimes $A[b] = \langle 10s, 40s, 90s, 100s, 100s \rangle$ for $b = \langle 1.1, 1.2, 1.3, 1.4, 1.5 \rangle$, respectively, these observed runtimes should clearly inform our predictions of $A[b = 1.6]$. Intuitively, for a nicely-behaved parameter b, a reasonable prediction for $A[b = 1.6]$ would be something around 100s since runtime appears to have levelled off at $b = 1.4$. Obviously, this is not the whole story, but at the very least, our prior prediction should be somewhat modified in the light of the observed evidence. If we employ separate independent functions for predicting A's runtime with each parameter configuration, no such modification of the prior prediction is possible.

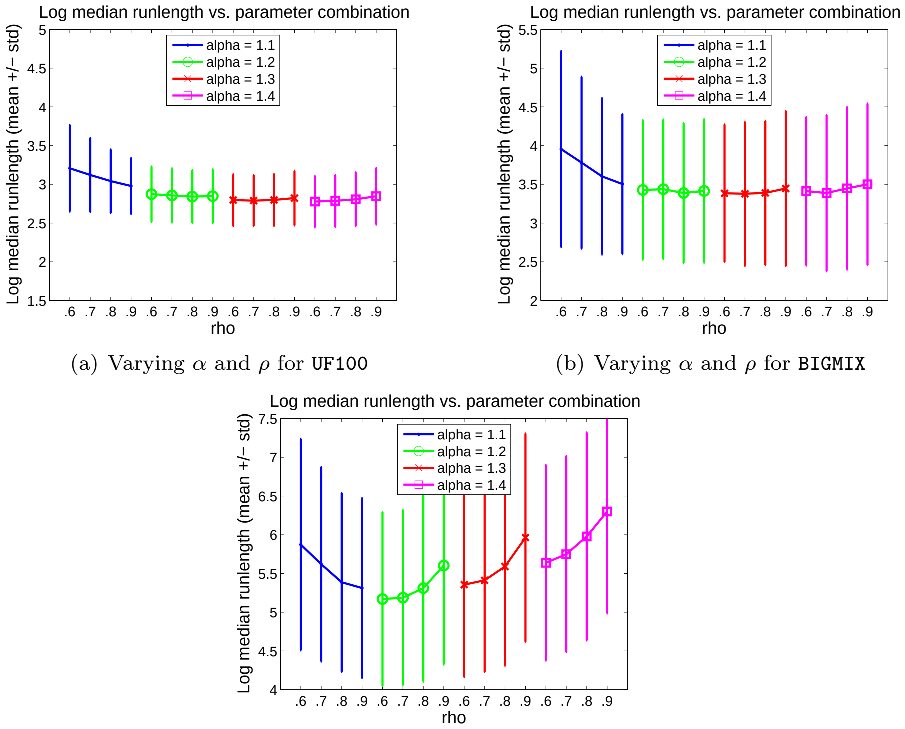
We then learn a single function $g(\cdot, \cdot)$ to predict A's runtime given the features of an instance and a parameter setting c. Once more, this comes down to the application of Equation (4) in order to learn a parameter vector $\boldsymbol{w}$. The final function has the form $g_{\boldsymbol{w}}(\boldsymbol{x}_n, c) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_n, c)$. The test phase for this approach is slightly more interesting than before. Given a new instance s_{N+1}, one computes its instance features $\boldsymbol{x}_{N+1}$ as usual, but in order to predict a runtime by evaluating function $g_{\boldsymbol{w}}(\boldsymbol{x}_{N+1}, c)$ one needs a parameter configuration c. The aim at test time is to find the optimal parameter configuration c^* for the new test instance, that is, the parameter configuration that minimizes expected runtime. If the number of possible parameter configurations is small, one can just evaluate $g_{\boldsymbol{w}}(\boldsymbol{x}_{N+1}, p)$ for every configuration. For a larger number of configurations, more complicated methods need to be applied, such as gradient descent for continuous parameters. It is important to note, though, that the evaluation function for this search is still very cheap: since it only consists of an inner product of two $(D+1)$-dimensional vectors, it only takes time $O(D)$. In particular, algorithm A does not have to be executed at all during the course of this search.

Note that we have a lot of freedom in the training phase. The number k_n of parameter configurations picked for instance s_n is not restricted to be constant across different instances s_n, and we are free to choose the parameter configurations $\boldsymbol{c}_n$ for each instance at will. Some choices of $\boldsymbol{c}_n$ will yield more informative training data than others, but how to maximize the utility of the training phase is an open research problem. This question is closely related to the fields of experimental design in statistics and active learning in machine learning. However, there are also important differences, for example, that running an algorithm with a suboptimal parameter setting is much more expensive than running it with its best parameter settting. We plan to address these concerns in future work.

In summary of this section, we would like to stress that our approach does *not* simply apply the methodology of [LBNS02, NLBD$^+$04] to choose from a set of finite possible parameter configurations by building a predictive model f^p for each configuration separately, but that it builds *one* predictive model g that includes the parameter configuration. This approach also works for an infinite number of possible parameter configurations, removing the need for a discretization of continuous parameters. An equivalent approach to ours for algorithm selection would be to build *one* predictive model for *all* algorithms that simply employs the used algorithm as one of its features. Even though standard linear regression would cease working upon inclusion of this categorical variable, hierarchical models [TJBB04, Jor06] that share parameters could still be employed and promise to yield better performance.

6 Experiments for SAPS with automatic parameter setting

In this section, we study how well empirical hardness models capture performance differences with different parameter settings for the SAPS algorithm. Throughout this section, we consider 16 different parameter settings, namely all combinations of $\alpha \in \{1.1, 1.2, 1.3, 1.4\}$ and $\rho \in \{0.6, 0.7, 0.8, 0.9\}$. We also evaluate the performance of a SAPS variant that automatically chooses between these parameter settings on a per-instance base.



(a) Varying α and ρ for `UF100`

(b) Varying α and ρ for `BIGMIX`

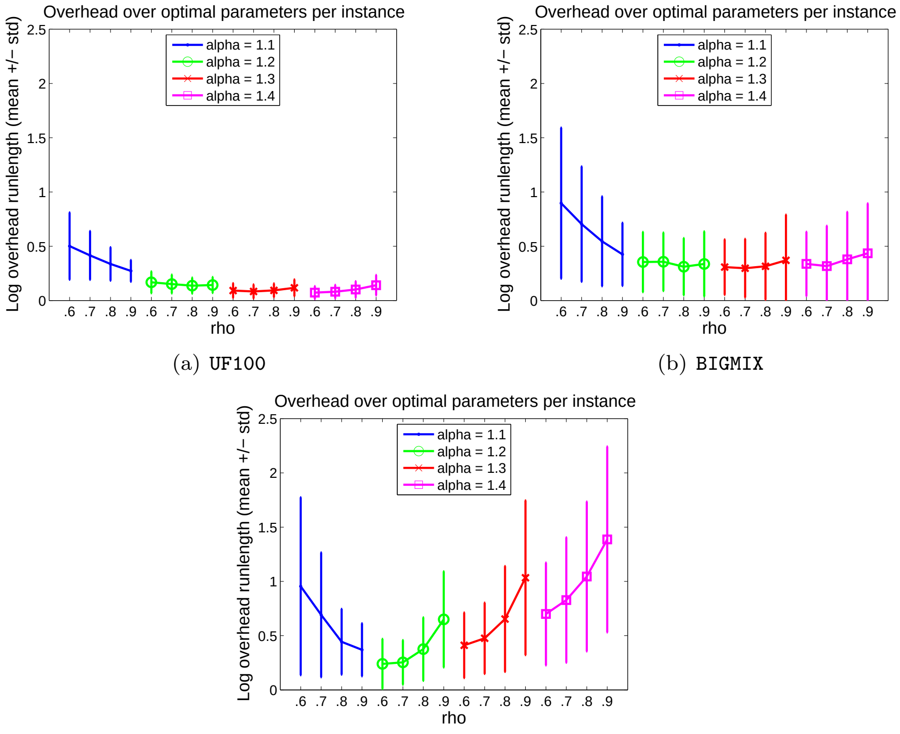
(c) Varying α and ρ for `SAT04-random`

Figure 2: True SAPS log median runlength vs. its algorithm parameters α and ρ for all our instance sets. Medians are taken over 100 runs for `UF100` and 10 runs for the other instance sets. We plot the mean ± standard deviation across all instances in the respective instances set.

6.1 Predictive accuracy for different parameter settings

In view of the experimental results for linear regression with linear and quadratic basis functions in Section 4.2, we only use linear basis functions in the instance features. However, since we do want to learn expressive models for the importance of the parameter settings, we employ all multiplicative combinations of parameters up to a power of four as basis functions. The resulting 4-th order polynoms in parameter space performed better than linear and quadratic functions and were more robust than 8-th order polynoms in experiments we omit. In order to be able to learn instance-specific functions of the parameters, we also use basis functions for all multiplicative combinations of the parameter monomials and the linear instance features. For each experiment, we then employ the methodology sketched out in Section 3.2 in order to choose a good subset of 40 basis functions for the respective data.

Figure 2 demonstrates the dependence of the SAPS algorithm on its parameters. This plot shows SAPS median log runlength for different parameter settings, averaged over all instances in the respective instance set plus/minus one standard



(a) `UF100` (b) `BIGMIX`

(c) `SAT04-random`

Figure 3: Log overhead in median runlength over the optimal parameter setting on a per instance base for our three data sets. Medians are taken over 100 runs for `UF100` and 10 runs for the other instance sets. We plot the mean ± standard deviation across all instances in the respective data set.

deviation. As we can see, the dependence on the parameters is not very strong for instance set `UF100`, stronger for `BIGMIX`, and the strongest for `SAT04-random`. For the latter one, average runlength across all instances differs by about one order of magnitude depending on the used parameter setting.

Notice that the large error bars in Figure 2 are due to inter-instance differences. In order to study how much the best parameter setting varies across instances, Figure 3 shows the overhead over the best parameter setting for each instance, averaged across all instances in the data set, plus/minus one standard deviation. We notice that the overhead is very small for most parameter settings in the `UF100` domain, and that the minimal overhead over the best parameter setting per instance is also small for the two other data sets. For example, it is around $10^{0.22} \approx 1.66$, such that we can never hope to improve the best fixed parameter setting by more than a factor of 1.66 for these data sets, not even with a perfect predictor that leads to optimal parameter choices for each instance. We attribute this effect in part to the coarse grid we laid over parameter space but mostly to the uniformity of our data sets. Notice, however, that the best fixed parameter setting differs across our three data sets. We believe that if there was more variation in the instances to be solved (as will be the case in most practical applications) there would also be a higher variation in optimal parameter settings, leading to a larger possible gain for an automatic parameter setting. In practice, our predictors will unfortunately be far from optimal, further reducing possible speedups. As we will see, this is especially true in our case since our training data is very sparse and noisy which inevitably leads to imperfect predictions.

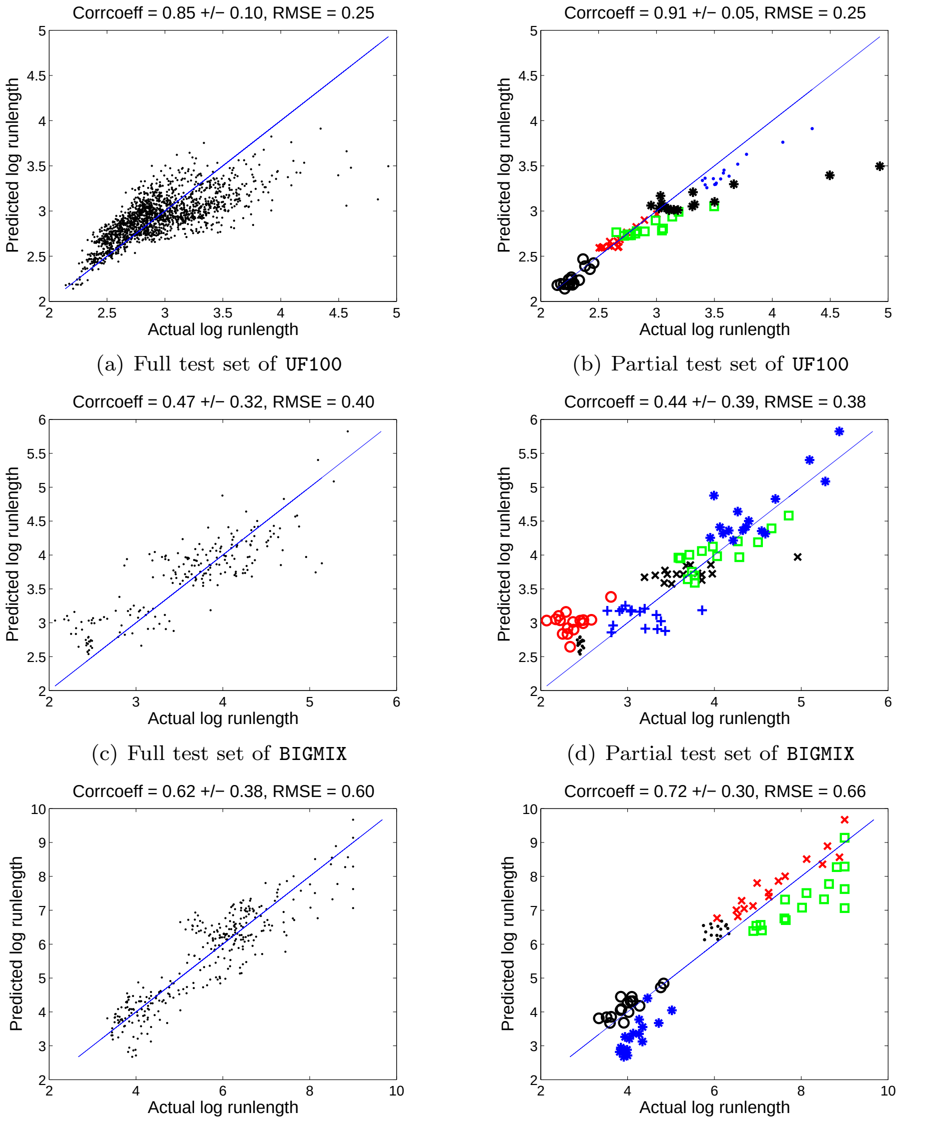
In the following experiment, we study to which extent our runtime predictions capture SAPS' dependency on its parameters on a per-instance base. We first show that we can predict log median SAPS runlength with different parameter settings on a per-instance base. In particular, the runtime predictions are such that the actual best parameter setting is almost always amongst the best predicted ones.

Experiment 6 (Predictive accuracy with varying parameters)**.** In this experiment, we study how well we can predict SAPS runlength with varying parameter setting. For each data set, we partitioned all instances into training, validation and test sets in the ratio 70-20-10. For each instance in BIGMIX and SAT04-random, we ran SAPS 10 times with each of the 16 parameter settings and recorded the median of these runs. For data set UF100, we could afford taking the median of 100 runs. Figures 4(a), 4(c), and 4(e) show the resulting predictive performance, plotting 16 data points for each instance.

In order to highlight the performance differences due to the parameter settings, we distinguish instances by different colours and symbols in Figures 4(b), 4(d), and 4(f). To prevent clutter, we only show a subset of test instances in these figures, including the extreme cases of the easiest/hardest instance with respect to actual/predicted runtime. From these figures, we see that predictive accuracy for SAPS with varying parameter settings is fairly good for the homogeneous data sets UF100 and SAT04-random and just acceptable for the small, inhomogeneous data set BIGMIX. Extremely bad parameter settings are almost always predicted to be the worst, even if the absolute runlength prediction is often off by a considerable amount. The best parameter setting is almost always predicted to be among the best and the best predicted parameter one is likewise very good in most cases. Finally, there is a strong positive correlation between the predicted runtime and actual runtime *per instance*.

In Figure 5, we show the runtime predictions for different parameter settings per instance in more detail. For each data set, we show two instances, namely the easiest and hardest instances in the test set (in terms of average log runlength across all 16 parameter settings). For each instance and each parameter setting, we show the actual and predicted log runlength. Note that the predictions mimic the trend of the actual log runlengths fairly accurately, even though they are sometimes shifted by over an order of magnitude. Also notice the varying shapes of the predictive function, both across and within instance sets. A very odd shape for the predictive function can be seen in data set BIGMIX in Figures 5(c) and 5(d). The noisy training data seems to suggest negative curvature in ρ for $\alpha = 1.4$ (the curve furthest to the right). Our domain knowledge prohibits such a prediction and in future work we may try to restrict predictive functions to match our intuitions.[13]

[13] Note that such a task is not straight-forward since we can only control our predictive function indirectly by means of its parameters. In a Bayesian setting, we may put an informative prior on



(a) Full test set of `UF100`

(b) Partial test set of `UF100`

(c) Full test set of `BIGMIX`

(d) Partial test set of `BIGMIX`

(e) Full test set of `SAT04-random`

(f) Partial test set of `SAT04-random`

Figure 4: Actual vs. predicted log runlength for SAPS with 16 different parameter configurations for each instance in the respective instance set ($\alpha \in \{1.1, 1.2, 1.3, 1.4\}$ and $\rho \in \{0.6, 0.7, 0.8, 0.9\}$). The plots on the left show all data points whereas the plots on the right focus on a few instances, indicating each one by a different colour and symbol. For `BIGMIX` and `SAT04-random`, we used the median of 10 runs, for `UF100` the median of 100 runs. Note that for the `SAT04-random` data set, runs were terminated when the runlength exceeded 10^9 steps. In the figure titles, we give the correlation coefficient between predicted log runlength and actual log runlength for the 16 parameter settings of each instance, more specifically its mean plus/minus one standard deviation across all instances.

In summary of experiment 6, we can say that the predictive accuracy for SAPS with varying parameters is acceptable and performance trends are well captured even if the absolute prediction is somewhat off. This is underlined by the strong correlation of actual and predicted log runlength per instance with varying parameter settings. We would like to stress that the imperfect absolute predictive accuracy is likely due to a fairly low amount of training data. We will strive to improve this base level performance in future work. We plan to increase both size and quality of our training data by running more experiments with more runs. This will lead to an improved predictive accuracy which will in turn enable us to pick better parameter settings yielding larger speedups.

However, the fact that the best predicted parameter setting is almost always among the actual best ones is already encouraging and we will now study whether this can be exploited to achieve good performance with an automatic parameter setting.

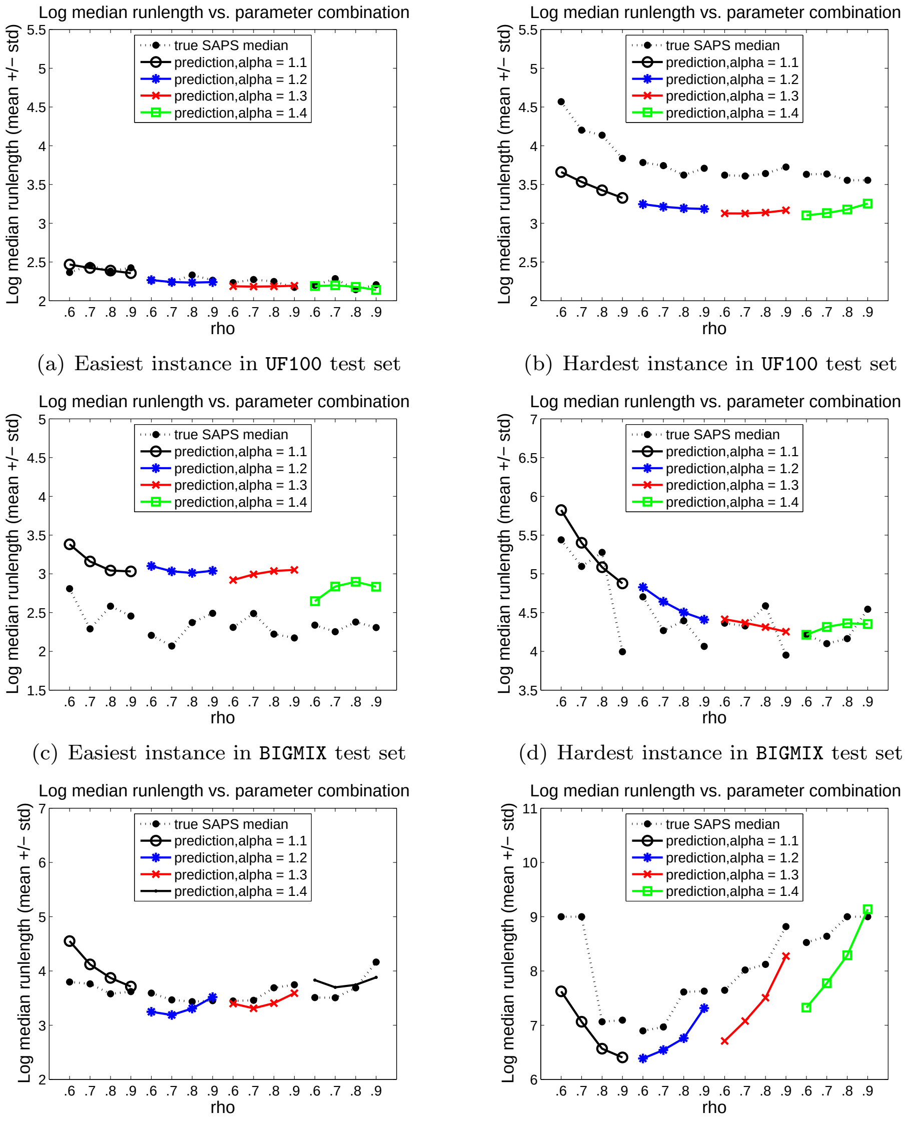
6.2 Performance of the automated parameter setting

In this section, we study whether the tolerable predictive accuracy for SAPS with varying parameter settings enables us to automatically select a good parameter setting on a per-instance base. We study the performance of an automatic parameter setting that was learned on a training and validation set that comes from the same data distribution as the test set.

Experiment 7 (Performance of the automatic parameter setting)**.** In this experiment, we use the same training, validation and test sets as in experiment 6. For each instance in the respective test set, we compare the performance of the best predicted parameter setting against a number of other parameter settings. In Figure 6, we plot the performance of our automatic parameter setting p_{auto} versus the performance of the best and the worst parameter setting for each instance (called p_{best} and p_{worst}, respectively). For instance sets `UF100` and especially `SAT04-random`, p_{auto} yields performance that is close to optimal for each instance, whereas performance for `BIGMIX` is worse. We attribute this to the small size of its training set combined with a lack of uniformity.

While these results, especially for the case of `SAT04-random`, are very encouraging, we need to put them into perspective. In Figure 3, we have already alluded to the fact that SAPS performance with the best fixed parameter setting comes within a factor of 1.66 of the optimal performance for each instance. As it turns out this is very close to the performance of our automatic parameter setting p_{auto}. Table 4 compares the performance of p_{auto} against the performance of the following parameter settings: the best and the worst parameter settings p_{best} and p_{worst} for each single instance, the SAPS default parameter setting $p_{\text{def}} = (\langle\alpha, \rho\rangle = \langle 1.3, 0.8\rangle)$, the best fixed parameter setting p^*_{train} for the merged training and validation sets and the best fixed parameter setting p^*_{test} for the test set. In accordance to what we saw in Figure 6, p_{auto} performs almost as well as p_{best}, and far better than p_{worst}. It also clearly outperforms the SAPS default parameter setting p_{def} for data set `SAT04-random`, beating it by a factor of $10^{0.44} \approx 2.75$. For this data set, p_{auto} even

the parameters for certain basis functions, but this is complicated by the fact that there are many relevant basis functions that interact in highly non-trivial ways.



(a) Easiest instance in `UF100` test set

(b) Hardest instance in `UF100` test set

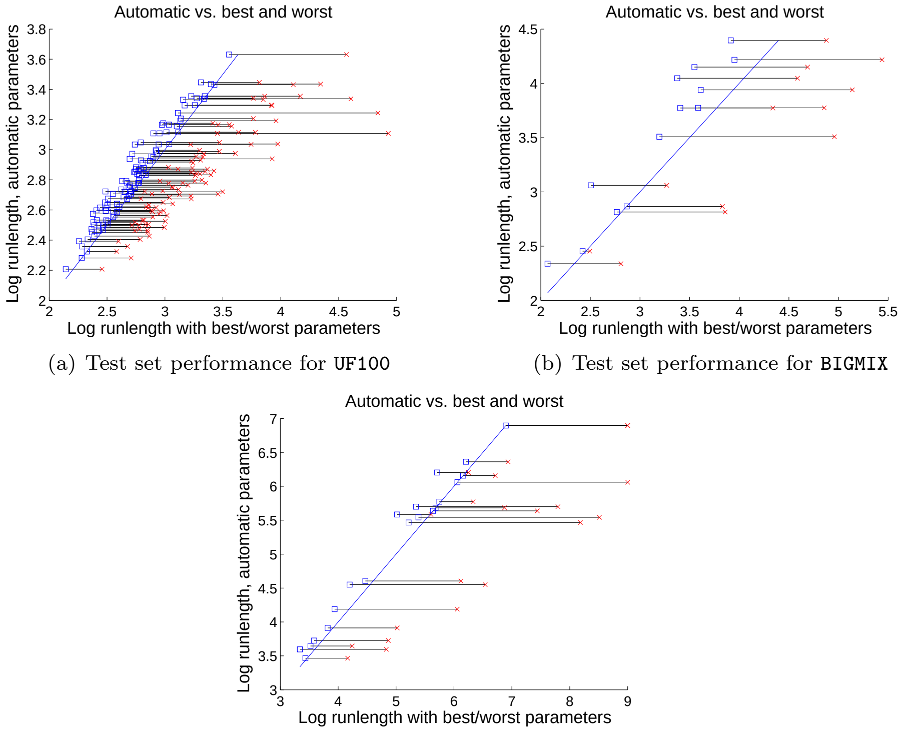
(c) Easiest instance in `BIGMIX` test set

(d) Hardest instance in `BIGMIX` test set

(e) Easiest instance in `SAT04-random` test set

(f) Hardest instance in `SAT04-random` test set

Figure 5: Actual vs. predicted log runlength for SAPS with 16 different parameter configurations for the easiest and hardest instance in the test set of each instance set (in terms of average log runlength across all 16 parameter settings). Again, we used the median of 10 SAPS runs for training and validation, but plot the median of 100 SAPS runs to reduce noise. Note the shift in y-axis, both across and within instance sets. For `SAT04-random`, actual median runlengths higher than 10^9 are plotted as 10^9.



(a) Test set performance for `UF100`

(b) Test set performance for `BIGMIX`

(c) Test set performance for `SAT04-random`

Figure 6: Actual log runlength with an automatically chosen parameter setting vs. actual log runlength with the best and worst parameter settings for each single instance. Training, validation and test sets are the same as in experiment 6 and we only report results on the test sets. Note the difference in scales for the different instance sets.

slightly outperforms the best fixed parameter setting p^*_{test} for the test set. Note that this means that in this domain our automatic choice of parameter setting outperforms *any* fixed parameter setting! For data set `UF100`, p_{auto} ties with p^*_{test} whereas for data set `BIGMIX`, it performs worse. We attribute this in part to the consistently strong performance of the best fixed parameter setting (as shown in Figure 3). In the case of `BIGMIX`, it does not come surprisingly that an optimal fixed parameter setting outperforms p_{auto} which was learned from very noisy and sparse data. We expect that with more and qualitatively better training data, predictive accuracy will improve, leading to a better performance of our automatic parameter setting. Finally, a wider range including the minimum for each instance would also improve the performance of our automatically chosen parameter setting while leaving the performance of the best fixed parameter setting unchanged.

We would like to stress the fact that for data set `SAT04-random`, our automated parameter setting outperforms the best fixed parameter setting p^*_{test}. This is a very hard problem since p^*_{test} is on average only a factor of 1.66 away from the optimal parameter setting for each instance. In our approach, small prediction errors lead to the choice of suboptimal parameter settings for some instance which leads to an overhead of $10^{0.17} \approx 1.47$ over the optimal parameter setting per instance, leaving an average improvement over p^*_{test} by a factor of $10^{0.06} \approx 1.14$.

Apart from tuning parameters on uniform data sets, we see two further practical applications of our approach. The first application is when the test set is not uniform, for example combining random and structured instances or instances from different applications. In these scenarios, there may not be a single well-performing parameter setting, such that any fixed parameter setting will compare poorly to the optimal parameter setting for each instance. In the same scenario, we still expect our automatic parameter setting to work very well, outperforming any fixed parameter setting by a larger margin than for uniform domains. We plan to study this scenario in more detail in future work. Our approach can also be applied when the test domain is not known a priori and we can only train on the instances available to use beforehand. We detail this scenario in the next section.

6.3 Performance of the automated parameter setting on domains different from the training domain

In this section, we study the performance of our automated parameter setting for the case where we train on instances from one distribution and test on instances from a different distribution. Note that this scenario is highly non-standard in machine learning since we loose any statistical guarantess, but that it may be very important in practical applications of automatic parameter setting.

When the test domain is not known a priori we cannot use a similar set of instances to determine the best fixed parameter setting. All that is available for parameter tuning is a training set of instances which may be considerably different than the training set. The most sensible option for choosing a fixed parameter setting is then to simply select the parameter setting p^*_{train} that performs best on the training set. In a machine learning approach, the best option is to train and validate on disjoint subsets of the training set and hope that the learned function will be general enough to yield reasonable predictions for different distributions as

Statistics	UF100	BIGMIX	SAT04-random	SAT04-random trained on BIGMIX
p^*_{train}	$\langle 1.4, 0.6\rangle$	$\langle 1.3, 0.7\rangle$	$\langle 1.2, 0.5\rangle$	$\langle 1.3, 0.7\rangle$
p^*_{test}	$\langle 1.4, 0.6\rangle$	$\langle 1.3, 0.7\rangle$	$\langle 1.2, 0.6\rangle$	$\langle 1.2, 0.5\rangle$
Corrcoeff per instance	0.85 ± 0.10	0.47 ± 0.32	0.62 ± 0.38	0.54 ± 0.29
Log overhead over p_{best}	0.08 ± 0.07	0.32 ± 0.22	0.17 ± 0.17	0.28 ± 0.23
Log speedup over p_{worst}	0.45 ± 0.27	0.75 ± 0.43	1.39 ± 0.90	1.25 ± 0.88
Log speedup over p_{def}	0.01 ± 0.07	-0.01 ± 0.32	0.44 ± 0.50	0.37 ± 0.54
Log speedup over p^*_{train}	-0.00 ± 0.04	-0.10 ± 0.28	0.07 ± 0.17	0.19 ± 0.40
Log speedup over p^*_{test}	-0.00 ± 0.04	-0.10 ± 0.28	0.06 ± 0.22	-0.04 ± 0.19

Table 4: This table reports on 4 experiments. For our three standard data sets, we learn an adaptive parameter setting on the respective training and validation sets and test it on the test set. The fourth experiment trains on the merged training and validation sets of BIGMIX, using the test set of BIGMIX for validation and all instances in SAT04-random as test set. For each experiment, we report the optimal fixed parameter setting p^*_{train} on the merged training and validation sets, the optimal fixed parameter setting p^*_{test} on the test set, and the correlation coefficient between the actual and predicted log runlength for all parameter settings per instance in the test set. We further compare the performance of SAPS with automatic parameter setting p_{auto} against the best and the worst parameter settings p_{best} and p_{worst} for each single instance, the SAPS default parameter setting $p_{def} = \langle \alpha, \rho \rangle = \langle 1.3, 0.8 \rangle$, the best fixed parameter setting p^*_{train} for the merged training and validation sets and the best fixed parameter setting p^*_{test} for the test set. Note that the performance differences are stated in log space. For example, "Log overhead over $p_{\text{best}} = 0.08 \pm 0.07$" means that SAPS with automatic parameter setting is on average $10^{0.08} = 1.20$ times slower than the optimal parameter setting for each instance.

well.[14]

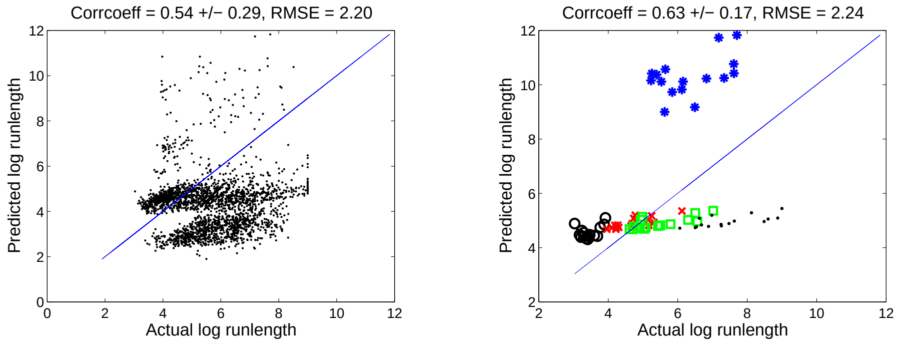
We now report on an experiment for this scenario. We trained on data set `BIGMIX` but tested on data set `SAT04-random`. Note that this experiment is motivated by the fact that the SAPS algorithm could have performed better in the SAT 2004 competition with a better setting of its parameters.[15]. All instances in `BIGMIX` were freely available online long before the competition. Thus, we could have trained on these instances in order to learn a predictor for runlength (or runtime) to be employed for an automatic choice of parameters. We now demonstrate that this approach would have improved SAPS performance both compared to its default parameters and to the best choice of parameters for the training set. Figure 7 shows that the absolute predictive accuracy is very weak in this case (this does not come surprisingly since training and test set distribution differ significantly). Note, however, that the correlation between predicted and actual runtimes for all 16 parameter settings per instance is still considerable (correlation coefficient 0.54).

Even though the absolute predictive accuracy is very weak, the best predicted parameter setting tends to perform quite well. Indeed, it is good enough to clearly outperform the SAPS default parameter setting $\langle \alpha, \rho \rangle = \langle 1.3, 0.8 \rangle$ by a factor of $10^{0.37} \approx 2.34$ and the best fixed parameter setting for the training set $\langle \alpha, \rho \rangle = \langle 1.4, 0.6 \rangle$ by a factor of $10^{0.19} \approx 1.54$. We detail this comparison in the scatter plots in Figure 8. Encouraged by these results, we plan to submit algorithms with built-in automatic tuning to future SAT competitions.

As a note of caution, however, we have to add that performance in this experiment was quite sensitive to the split of data set `BIGMIX` into training and validation set, and that with a different split, we only got a marginal speedup. This noisy effect is almost certainly due to the small size of `BIGMIX` and future experiments will include many more instances, especially for such heterogeneuous instance sets as `BIGMIX`.

[14] Obviously, in such a scenario one would choose a machine learner with excellent generalization performance and regularize it strongly to avoid any kind of overfitting. There also exist Bayesian methods to at least quantify the uncertainty in our predictions for new instances. These methods can implicitly evaluate whether or not a test instance is similar to the training instances and if it is highly dissimilar assign a high uncertainty to the runlength prediction. We will use one of these methods, Bayesian linear regression, in Section 8.

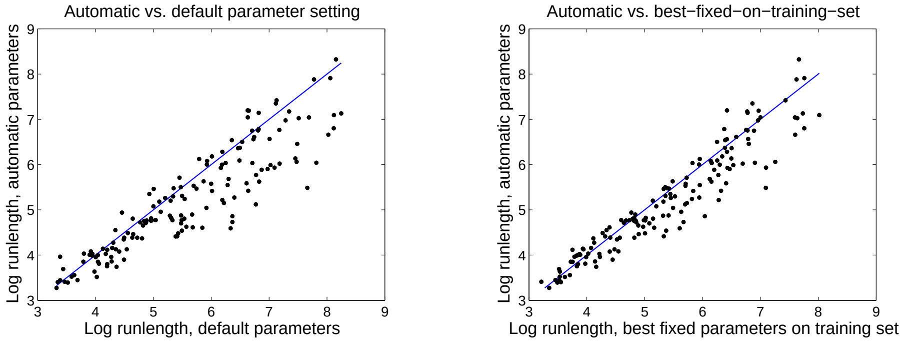
[15] SAPS placed second for solving random instances, only beaten by a variant of WalkSAT Novelty$^+$, but with the improvements made here could have placed first. However, this methodology is equivalently applicable to WalkSAT Novelty$^+$, and we plan to study this in future work.



(a) Full set `SAT04-random` as test set

(b) Partial set `SAT04-random` as test set

Figure 7: Actual vs. predicted log runlength for SAPS with 16 different parameter configurations ($\alpha \in \{1.1, 1.2, 1.3, 1.4\}$ and $\rho \in \{0.6, 0.7, 0.8, 0.9\}$) for each instance in `SAT04-random`. The plot on the left shows all data points whereas the plots on the right focusses on a few instances, indicating each one by a different colour and symbol. For training, we used the merged training and validation set of `BIGMIX`, for validation the `BIGMIX` validation set, and for test the data set `SAT04-random`. Due to the large difference between these two data sets, the poor absolute performance does not come surprisingly. For all of these sets, each data point is the median of 10 runs. Note that runs were terminated when the runlength exceeded 10^9 steps. In the figure titles, we give the correlation coefficient between predicted log runlength and actual log runlength for the 16 parameter settings of each instance, more specifically its mean plus/minus one standard deviation across all instances. Even though the absolute error is very high, this correlation is fairly high.



(a) SAPS with automatic parameter setting vs. SAPS with default parameter setting $\langle \alpha, \rho \rangle = \langle 1.3, 0.8 \rangle$ for `SAT04-random`

(b) SAPS with automatic parameter setting vs. SAPS with best fixed parameter setting $\langle \alpha, \rho \rangle = \langle 1.4, 0.6 \rangle$ on the training set `BIGMIX`

Figure 8: Actual log median runlength for SAPS with automatically chosen parameters vs. default parameters and the best parameter setting on the training set. The automatic parameter choice was trained on the merged training and validation set of `BIGMIX` and validated on the test set of `BIGMIX`.

7 Real-World Problem Settings

This section relates automated parameter setting to the reality of problem solving in industrial settings. We distinguish between two scenarios. In the first scenario, which is implicitly assumed by many previous approaches, the solver is continuously used to solve problems from a single comparably uniform problem domain. In contrast, in the second scenario the problem solver faces problem instances with significant differences. These differences may either be due to the use of the solver as a general problem solving tool across different groups, or it may be due to changes that occur in the problem modelling. We argue that most previous approaches will have problems to adapt to this second scenario.

7.1 Uniform domain space

In this scenario, a modelling m is defined in order to tackle a combinatorial problem P from a well identified domain space. A solver is then used to solve domain instances. Generic heuristics can be used to improve efficiency but specific knowledge from the domain space could allow specialized heuristics and algorithms to be used. The complete modelling process is presented in Figure 9.

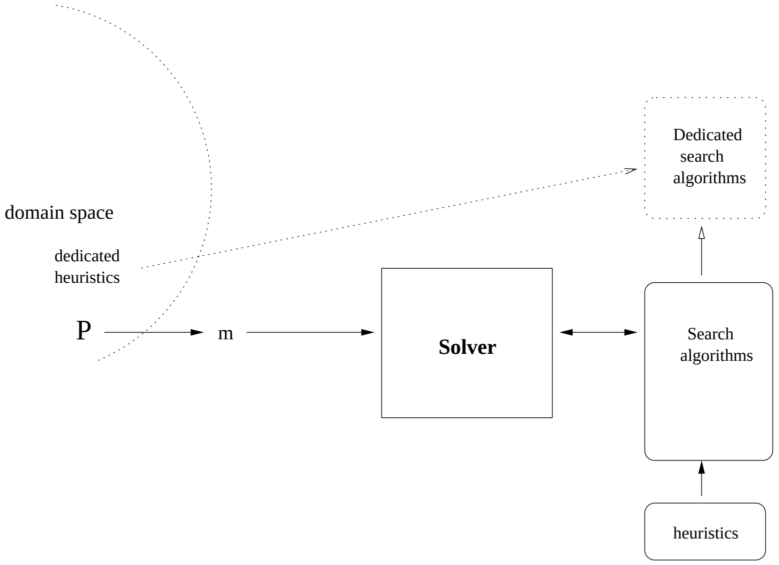


Figure 9: The modelling process

In a real-world scenario, problem instances generated by the modelling m successively appear and have to be processed by the solver. We visualize this in Figure 10 where a solver is successively used to solve incoming instances from the same modelling m.

In this setting, previous learning approaches [LL98, Sil00, LBNS02, NLBD$^+$04, GHBF05] can succeed. Indeed their extensive offline learning phase can be implemented by learning from a large set of domain instances. Since the domain space

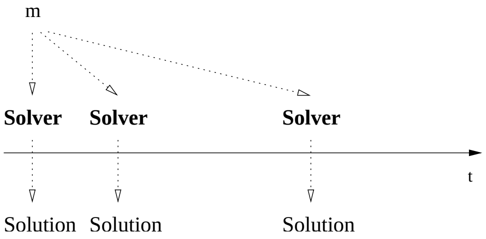


Figure 10: Uniform domain space: operational scenario

is uniform, knowledge learned from previous instances generalizes to the future and good performance prediction can be achieved.

7.2 Multiple domain spaces

In many scenarios, the solver will receive instances from different domain spaces or from different problem modellings (see Figure 11). The characteristics of problem instances modelled by m can differ significantly from those modelled by m'.

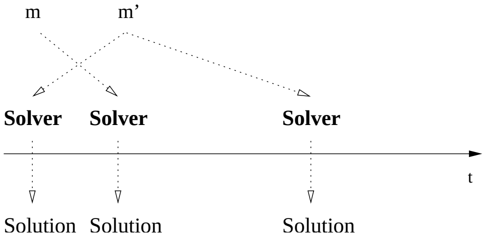


Figure 11: Multi domain spaces: operational scenario

In practice, a solver can thus receive a mix of instances from a variety of different domains. This can happen for many reasons:

- The solver is used as a "General Problem Solver" and domains are not known in advance[16]. We cannot assume a specific offline learning phase for each domain. Indeed, even if the domains were a priori known, this learning would be very expensive.

  If the domain is assumed to be fixed (e.g., planning), we still cannot assume that all instances come from the same application. Such a fixed domain with different instantiations is presented as the target application in [SM05]. In this work, the domain space is planning and instances are clustered in applications as different as web-service composition, autonomous computing, sensor networks, etc. In such a scenario, previous approaches will have to learn from a large instance set covering all three target applications.

[16] This is usual when the solver benefits from a large set of resources (Grid, clusters) and is addressed by many departments of the same company (marketing, financial services, production planning, etc).

- Even with one domain space and one final application, we cannot assume a single fixed problem modelling. Indeed, constraints could be added/removed over time in relation with changes occurring in the initial problem. When the modelling changes, we cannot assume that the knowledge learned in an initial learning phase will generalize to the new modelling even if the modifications to the modelling are slow and incremental. The work presented in [BFP$^+$05] presents a CP system which extracts its parameters from a rule engine. This allows a quick adaptation of the modelling to the environmental requirements, but an offline learning approach would clearly be infeasible in such a system.
- Last but not least, even if the problem is fixed, the capacity of programmers is not. Indeed, end-user's modelling expertise usually grows over time. As a result, initial modelling decisions are revised and new modellings are used. As in the previous case, results from an initial offline learning phase will quickly become outdated in this scenario.

As we can see, previous work based on extensive offline learning can hardly match the changing settings of real world applications. Our claim is that our proposed approach will be able to perform well in the outlined real operational settings.

7.3 Offline versus on-the-fly problem solving

What distinguishes offline from on-the-fly problem solving is just time granularity. Indeed, classical batch applications which are using a solver on a daily basis, such as, for example, production scheduling, are perceived as offline. At the same time, applications which make very close calls to a search component, such as bandwidth reservation in ATM networks, are often classified as on-the-fly. What really distinguishes the two concepts is the fact that in on-the-fly settings, a subsequent call is sometimes a refinement of the question embedded in some previous query. However, even with that in mind the frontier between offline and on-the-fly is hard to catch.

Previous batch learning approaches can be applied to both offline and on-the-fly settings, with (batch) learning repeated for example every night. However, this may quickly become infeasible when the training set grows too large, such that the complete batch learning would no longer finish in one night. Furthermore, this approach does not enable the system to use the most recent information (e.g., how it solved a very similar instance five minutes ago). These arguments clearly call for an *incremental learning* approach that starts with little knowledge and acquires additional knowledge from every instance it solves. We depict this for the uniform domain case in Figure 12, and for the multi domain case in Figure 13.

In the multiple domain case, when an instance comes from a new modelling or a new domain, performance and runtime predictions cannot be expected to be good. However, after a few instances of the new type have been seen, the system could slowly learn enough about the new domain to make increasingly accurate predictions. Practical systems would benefit from the opportunity to detect such as domain change. It could, for example, be detected if the solver makes probabilistic runtime predictions and all of a sudden the predictive uncertainty grows rapidly. In Section 9, we will show that predictive uncertainty is indeed much higher for instances that are very different from the so-far seen training instances when we

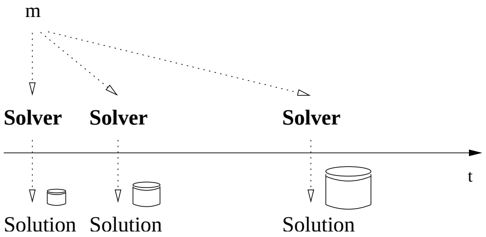


Figure 12: Uniform domain space: operational resolution and learning scenario

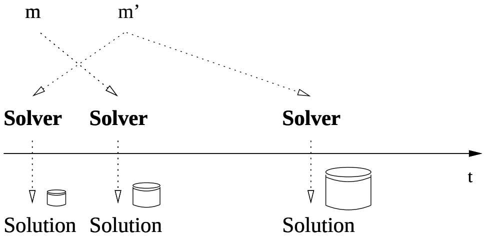


Figure 13: Multi domain spaces: operational resolution and learning scenario

use Bayesian prediction methodology. We introduce the technical details of the methodology we use in the next section.

8 On-the-fly learning and estimates of uncertainty

In this section, we introduce sequential Bayesian learning techniques which can deal with incremental data, and which also allow for a quantization of the uncertainty in the predictions we make. This approach clearly widens the applicability of runtime prediction, and in particular enables us to tackle the challenges arising in the real-world scenarios described in Section 7. We first outline the general methodology of sequential Bayesian learning and subsequently apply it to Bayesian linear regression for runtime prediction.

Generally speaking, say, we want to reason about some arbitrary quantity X in an incremental setting. In this setting, we have some prior information on X which gets refined as we acquire more and more information about X. Being Bayesian, we view X as a random variable. The prior information on X then takes the form of a *prior probability distribution* $P(X)$, and we want to update this prior in the light of some observed data $y_{1:N} = [y_1, \ldots, y_N]^T$. We will assume that $y_{1:N}$ are N independent identically distributed (i.i.d.) realizations of the random variable Y, and we will also assume that we know the conditional distribution $P(Y_n = y_n|X)$ of a single data point y_n given X. This term is also referred to as the *likelihood* of the data point. After seeing the first data point y_1, we can easily compute our *posterior* belief $P(X|Y_1 = y_1)$ in X as follows. Using Bayes' law and the fact that $P(Y_1 = y_1)$ is just a constant, this can be written as the product of the prior and the likelihood:

$$P(X|Y_1 = y_1) = \frac{P(X)P(Y_1 = y_1|X)}{P(Y_1 = y_1)} \propto P(X)P(Y_1 = y_1|X).$$

An equivalent observation holds for the second data point, now using the posterior distribution after the first data point as the prior for the second data point:

$$P(X|Y_{1:2} = y_{1:2}) \propto P(X|Y_1 = y_1)P(Y_2 = y_2|X) = P(X)\prod_{n=1}^{2} P(Y_n = y_n|X).$$

Sequentially carrying out this "updating" step for each data point y_n in turn yields the expression

$$P(X|Y_{1:N} = y_{1:N}) \propto P(X)\prod_{n=1}^{N} P(Y_n = y_n|X),$$

which is exactly the same expression one big batch update of X with all the data $Y_{1:N} = y_{1:N}$ would yield. Thus, sequential Bayesian updating does not loose any information due to its incrementality. It becomes both very powerful and elegant when the prior probability distribution and the likelihood function take on similar forms, such that the posterior (prior times likelihood) has the same functional form as the prior.[17] If this functional form has a compact representation with some free parameters, then Bayesian updating comes down to simply updating these parameters with every data point.[18]

Now, let us apply sequential Bayesian updating to the case of linear regression for runtime prediction. In order to keep the notation uncluttered, we only cover the case of predicting the runtime of a single algorithm with fixed parameters – however, the approach generalizes in a straight-forward fashion to multiple algorithms and/or multiple parameter settings.

In section 3, we learned a linear function that predicted a single runtime $\hat{r}_n = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_n)$ for each problem instance s_n. In contrast, we will now predict a probability density $P(r_n|\boldsymbol{x}_n, \boldsymbol{w})$ for the runtime.

As in linear and ridge regression, Bayesian linear regression adapts the parameters $\boldsymbol{w}$ of a linear function. Linear and ridge regression found the best point estimate of the parameter vector $\boldsymbol{w}$. In contrast, Bayesian linear regression employs a *probability distribution* for $\boldsymbol{w}$. We employ a Gaussian prior $P(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \Sigma_0)$ and update this prior in the light of sequentially arriving data. For example, our first data point consists of the pair $\langle \boldsymbol{\phi}(\boldsymbol{x}_1), r_1\rangle$ of basis functions $\boldsymbol{\phi}(\boldsymbol{x}_1)$ and runtime r_1. Assuming that our observation noise is Gaussian distributed with zero mean and variance σ^2_{obs}, this yields a likelihood function for parameter vector $\boldsymbol{w}$:

$$lik(\boldsymbol{w}; \boldsymbol{x}_1, r_1) = P(r_1|\boldsymbol{x}_1, \boldsymbol{w}) = \mathcal{N}(r_1; \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_1), \sigma^2_{obs}).$$

Since the product of two Gaussians is still a Gaussian, the posterior distribution $P(\boldsymbol{w}|\boldsymbol{x}_1, r_1)$ of the parameters $\boldsymbol{w}$ will also be a Gaussian $\mathcal{N}(\boldsymbol{\mu}_1, \Sigma_1)$. Further, since a multivariate Gaussian $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$ is completely specified by its mean vector $\boldsymbol{\mu}$ and

[17] In this case, the prior is said to be *conjugate* for the likelihood function.

[18] This happens, e.g., for members of the popular *exponential family* of distributions.

covariance matrix Σ, we only need to sequentially update these quantities. The necessary update equations to get the Gaussian parameter distribution $\mathcal{N}(\boldsymbol{\mu}_N, \Sigma_N)$ after having seen N data points are as follows (see, e.g., [Bis06] for a derivation):

$$\boldsymbol{\mu}_N = \Sigma_N \Sigma_0^{-1} \boldsymbol{\mu}_0 + \sigma_{obs}^{-2} \Sigma_N \Phi^T \boldsymbol{r}, \tag{5}$$

$$\Sigma_N^{-1} = \Sigma_0^{-1} + \sigma_{obs}^{-2} \Phi^T \Phi. \tag{6}$$

For $(D+1)$-dimensional data, the complexity of a single update with K data points is dominated by inverting the $(D+1)$-dimensional matrix Σ_0 (complexity $O(D^3)$), and the product $\Phi^T \Phi$ (complexity $O(D^2 K)$). If every data point is added on its own, N matrix inversions yield a total computational complexity of $O(ND^3)$ which can be reduced to $O(ND^2)$ by using a Kalman filter to update the weight vector, thus only requiring a matrix-vector mutliplication with complexity $O(D^2)$ per step (see the chapter on Kalman filtering and smoothing in [Jor06] for details). Thus, the asymptotic complexity of sequential and batch learning is identical.

For a zero mean Gaussian prior $\mathcal{N}(\boldsymbol{w}; \boldsymbol{0}, \Sigma_0)$ on the parameters $\boldsymbol{w}$, it is easily shown that the posterior mean is $\boldsymbol{\mu}_N = (\sigma_{obs}^2 \Sigma_0^{-1} + \Phi^T \Phi)^{-1} \Phi^T \boldsymbol{r}$. An uninformative prior would have infinite variance such that $\Sigma_0^{-1} = 0$ which leads to a posterior mean $\boldsymbol{\mu}_N = (\Phi^T \Phi)^{-1} \Phi^T \boldsymbol{r}$ that exactly matches the normal equations from the least squares solution in Equation (2). A slightly more informed prior with $\Sigma_0 = \alpha I$ (where I is the $(D+1)$-dimensional identity matrix) would lead to $\boldsymbol{\mu}_N = ((\sigma_{obs}^2/\alpha) I + \Phi^T \Phi)^{-1} \Phi^T \boldsymbol{r}$ which matches the solution for ridge regression from Equation (4) with $\lambda = \sigma_{obs}^2/\alpha$. We have thus shown that Bayesian linear regression with a prior with zero mean and diagonal variance yields a Gaussian distribution over the parameters $\boldsymbol{w}$ that is centered on the ridge solution $\boldsymbol{w}_{ridge}$.

While a probability distribution on the parameters $\boldsymbol{w}$ can be very useful, we are ultimately interested in a probability distribution for our runtime predictions. In the non-Bayesian case, where we had a point estimate for the parameters $\boldsymbol{w}$, a runtime prediction was computed by evaluating $r_{N+1} = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_{N+1})$. The Bayesian alternative is to integrate out the parameters, yielding a probability distribution over runtime (we drop the index $N+1$ for simplicity):

$$P(r|\boldsymbol{x}) = \int P(r|\boldsymbol{w}_N, \boldsymbol{x}) P(\boldsymbol{w}_N) d\boldsymbol{w}_N, \tag{7}$$

where $\boldsymbol{w}_N$ denotes the posterior parameters after having seen N data cases. $P(\boldsymbol{w}_N)$ is just the posterior parameter distribution $\mathcal{N}(\boldsymbol{\mu}_N, \Sigma_N)$, with $\boldsymbol{\mu}_N$ and Σ_N defined by Equations (5) and (6), respectively. Assuming Gaussian observation noise with variance σ_{obs}^2, the runtime prediction $P(r|\boldsymbol{w}, \boldsymbol{x})$ for a fixed parameter $\boldsymbol{w}$ equals $\mathcal{N}(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}), \sigma_{obs}^2)$. Since the convolution of two Gaussians is again a Gaussian, Equation (7) is solvable in closed form and yields the predictive distribution

$$P(r|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}_N^T \boldsymbol{\phi}(\boldsymbol{x}), \sigma_{obs}^2 + \boldsymbol{\phi}(\boldsymbol{x})^T \Sigma_N \boldsymbol{\phi}(\boldsymbol{x})), \tag{8}$$

We showed above that in the case of a Gaussian prior with zero mean $\boldsymbol{\mu}_0 = \boldsymbol{0}$ and diagonal covariance matrix $\Sigma_0 = \alpha I$, $\boldsymbol{\mu}_N$ is just the solution of ridge regression. Thus, the mean of the predictive distribution $P(r|\boldsymbol{x})$ is just the runtime prediction obtained by ridge regression, whereas the covariance depends both on the observation noise σ_{obs}^2 and the strength α of the prior. Just like for the ridge regularizer λ in the non-Bayesian case, the performance of Bayesian linear regression depends quite strongly on the strength of the prior which requires a careful setting, for example by cross validation.

The Bayesian treatment provides an alternative solution to the problem of setting α which we sketch here (we did not implement this and it is not of importance for understanding the current report, but it *is* important for our future work). The standard Bayesian solution (see, e.g., [Bis06]) is to maximize the *marginal likelihood* of the data

$$P(\boldsymbol{r}|\alpha, \sigma_{obs}^2) = \int P(\boldsymbol{r}|\boldsymbol{w}_N, \sigma_{obs}^2) P(\boldsymbol{w}_N|\alpha) d\boldsymbol{w}_N$$

with respect to α and σ_{obs}^2. This procedure is known as *empirical Bayes*, *type 2 maximum likelihood*, or the *evidence framework*. Even though it finds a point estimate of (hyper)parameters, this approach is not very prone to overfitting since it marginalizes over the first level of parameters $\boldsymbol{w}$. (Remember that this first level of parameters is maximized in maximum likelihood and also in the non-Bayesian case.) Type 2 maximum likelihood thus lifts the maximization one level higher, achieving a much more stable solution that does not require cross validation anymore. We will implement type 2 maximum likelihood in future work.

Finally, a truly Bayesian treatment would put (uninformative) hyperpriors on both hyperparameters α and σ_{obs}^2 and then integrate them out to yield a Student-t distribution $P(\boldsymbol{w}_N)$ for the posterior parameters. Unfortunately, Equation (7) is then not solvable in closed form anymore. Although approximations can be employed, type 2 maximum likelihood seems to yield better results in practice [Bis06].

In summary of this section, note that Bayesian linear regression (when using a prior with mean zero and diagonal covariance matrix) leads to a predictive distribution for runtime that is centered on the runtime prediction of ridge regression, but also provides an estimate of the uncertainty for this prediction.

8.1 Bayesian linear regression: a demonstration

In the previous Section, we showed that in contrast to least squares or ridge regression, Bayesian linear regression not only yields predictions of runtime, but also its uncertainty in these predictions. In this section, we show in a toy example that these estimates of uncertainty can be very informative.

For visualization purposes, data in this toy example only has a single feature x. We thus want to learn a function $f : \Re \rightarrow \Re$. For this function, we have six training examples that are depicted by the crosses in Figure 14. The true function is $f(x) = x^4$ and the training data is corrupted by white Gaussian noise $\mathcal{N}(0, 1)$. We only employ the basis functions $\boldsymbol{\phi}(x)^T = [1 \; x \; x^2]$, such that our predictive mean $\boldsymbol{\mu}^T \boldsymbol{\phi}(\boldsymbol{x}) = \mu_0 + \mu_1 x + \mu_2 x^2$ will be a two-dimensional polynom in x. Obviously, by including higher order terms, we could fit the function perfectly (given enough data points), but in this example, we *want* to have an error in order to demonstrate our estimates of uncertainty. This also mimics a practical situation in which we can't assume that the true function is included in our hypothesis class).

Note in Figure 14 that the predictive mean is fairly accurate in regions where we have training data. This is to be expected since we have a lot of information about the function in thoses areas. The further we move away from the data the less information we have, especially on the two boundaries, where we have no more

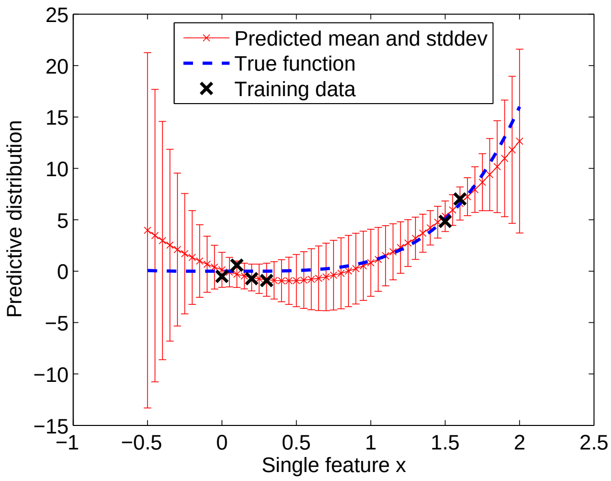


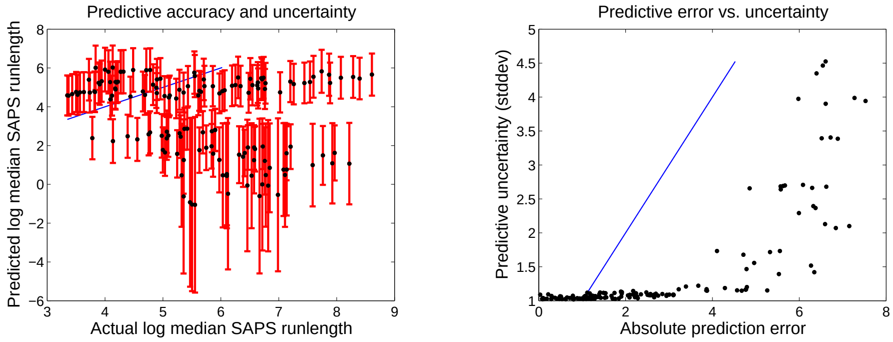
Figure 14: Demonstration of predictive distribution (predicted mean and standard deviation) in Bayesian linear regression for a toy example. Since the predictive distribution is a Gaussian (see Formula 8), this completely specifies the predictive distribution. Note that the variance in the regions with training data points is very small but grows the further away one gets from the training data.

data up to minus/plus infinity. Now note that the predictive variance captures this information very well. In regions with data the predictive variance is very small, but it grows larger far away from the data. Thus, even though we cannot do better in the regions where we have no data, at least we know that we cannot trust our prediction. This has obvious implications for practical scenarios. One very simplistic use would be to use the prediction when the predictive variance is low and to discard it when the variance is too high. We can also imagine straightforward combinations with other methods, such as simple heuristics.

Furthermore, predictive variance can be used to decide where to pick the next training data point. In Figure 14, a data point near $x = 1$ would be very informative and reduce the variance for a large region, whereas another data point around $x = 0$ would not carry a lot of information. The question of data selection relates to the field of *active learning*, an active research area methods of which we plan to use in this domain in the future.

9 Bayesian linear regression in practice

The theoretical treatment in the last section tells us that Bayesian linear regression will lead to the exact same mean prediction as ridge regression. On top of this, Bayesian linear regression also yields uncertainty estimates. In this section, we show that a similar effect as in our demonstration of Bayesian linear regression in Section 8.1 also occurs for real data. Regions in space for which there exist no training data will be associated with high predictive uncertainty. Further, we



(a) Uncertainty of predictions when trained on `BIGMIX` and tested on `SAT04-random`

(b) Uncertainty vs. absolute prediction error when trained on `BIGMIX` and tested on `SAT04-random`

Figure 15: Predictions and their uncertainty when trained on `BIGMIX` and tested on `SAT04-random`. We only use the SAPS default parameter setting and medians of 1000 runs for `BIGMIX`and of 100 runs for `SAT04-random`. For the uncertainty estimates, note that we assumed $\sigma^2_{obs} = 1$, such that the minimal predictive variance is 1 and that values close to this are taken on for the points with small absolute prediction error.

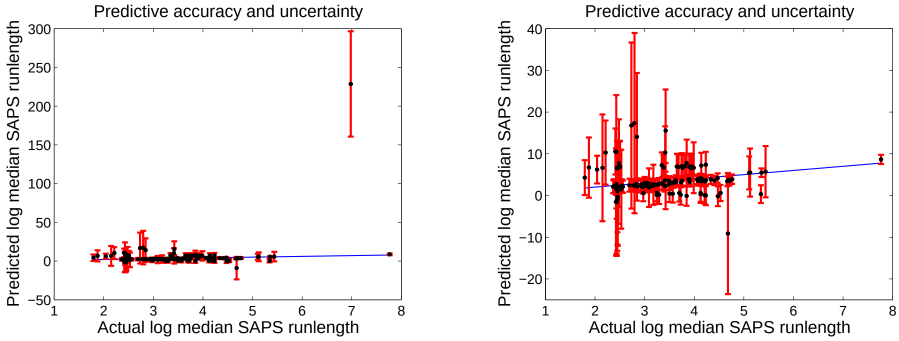
perform a first experiment to study the performance of Bayesian linear regression when incrementally trained on instances from different domains.

9.1 Uncertainty estimates for real data

In Section 6.3, we have already seen experimental evidence that predictive accuracy is very low when test and training set do not come from the same distribution. Here, we study whether we can at least hope to identify instances for which our predictions will be erroneous.

First, we study predictive performance when trained on data set `BIGMIX` and tested on data set `SAT04-random`. For both data sets, we use the default parameter setting only, but the methodology is equally applicable in the case of multiple parameter settings. For `BIGMIX`, we use the median of 1000 runs, for `SAT04-random` of 100 runs. As before, we use the approach sketched in Section 3.2 in order to select up to 40 linear basis functions. For feature selection, the training set is split into training and validation set, but afterwards, the complete training set is used to learn a predictor. For Bayesian linear regression, we assume constant measurement noise of $\sigma^2_{obs} = 1$ and a zero mean prior with covariance $\Sigma_0 = \alpha I$ with $\alpha = 10$. This corresponds to ridge regression with $\lambda = 10^{-1}$ and showed robust performance in all our experiments. In practice, however, one would obviously like to estimate both α and σ_{obs} from the data, and we plan to implement this in future work.

Figure 15(a) presents the result of training on data set `BIGMIX` and testing on `SAT04-random`. Like in Figure 7, predictive performance is very weak, because training and test instances come from different distributions. Bayesian linear regression solves this problem in part by at least associating higher predictive uncertainty to



(a) Uncertainty of predictions when trained on `SAT04-random`and tested on `BIGMIX`

(b) Zoomed in version of (a)

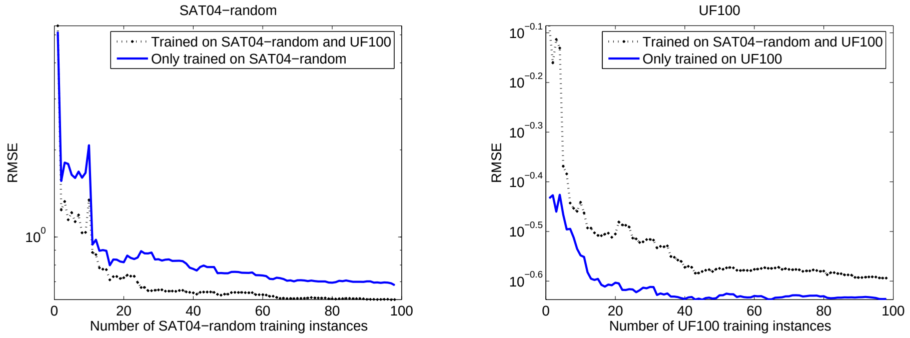
Figure 16: Predictions and their uncertainty when trained on `SAT04-random`and tested on `BIGMIX`. We only use the SAPS default parameter setting and medians of 1000 runs for `BIGMIX`and of 100 runs for `SAT04-random`. The figure on the right is the same as the one on the left, but zoomed to the bulk of data points.

the instances it shows very poor performance on. Note that since we assume measurement noise $\sigma^2_{obs} = 1$, the predictive uncertainty can never fall below one. Figure 15(b) shows that it is indeed very close to one for all instances with reasonable prediction and grows with prediction error.

In Figure 16, we study how Bayesian linear regression deals with a scenario in which a fairly uniform distribution is used for training and a more diverse distribution is used as test set. In particular, we used `SAT04-random` for training and tested on `BIGMIX`. Intuitively, we expect good predictive performance and small uncertainty for those instances in `BIGMIX` that are similar to the instances in `SAT04-random`, and poor accuracy with high uncertainty for other instances. We are very pleased to perfectly observe this pattern in Figure 16.

9.2 Incremental learning for multiple domains

In this section, we study the predictive performance of Bayesian linear regression when incrementally trained with instances from different distributions. In particular, we incrementally feed in 200 data points, alternating between the data sets `UF100` and `SAT04-random`. As test set, we use both hold out test sets from `UF100` and `SAT04-random`. We compute root mean squared error (RMSE) of the learned regression function after every data point on each of these test sets and compare it to the RMSE obtained when only training on the available data points from the respective distribution. Figure 17 shows the result of this experiment. Here, we observe that the quality of all models incrementally improves with the amount of training data. For the models learned only on data from a single domain, this relates to the real-world operational scenario depicted in Figure 12, for the case of learning from multiple domains to the scenario in Figure 13. Training on a mix of `UF100` and `SAT04-random` leads to somewhat weaker performance for `UF100` than only training on `UF100`, but in the case of `SAT04-random` we see an improvement



(a) Performance on `SAT04-random` when training on `UF100` and `SAT04-random`

(b) Performance on `UF100` when training on `UF100` and `SAT04-random`

Figure 17: Performance when incrementally training on instance distributions `SAT04-random` and `UF100` at the same time. The test set performance for `SAT04-random` is slightly better than when only training on `SAT04-random`, whereas the performance for `UF100` gets somewhat worse. Note that this data is fairly noisy since this experiment only reflects one single split into training and test set and one particular ordering of the data points.

due to the additional training data. However, we expect this experiment to be quite noisy and plan to study the same effect with more data in future work. For now, we merely note that predictive performance was not affected much by partly training on instances from another distribution.

This has implications for practical applications since it suggests that our approach can be used in a scenario where a solver is continuously presented with instances from different domains. However, we expect that a hierarchical Bayesian approach [TJBB04, Jor06] can exploit similarities between different domains even better, and we plan to study this in future work.

Another possible approach for learning in scenarios with domain changes would be to control the strength of the posterior weight distribution $\mathcal{N}(\boldsymbol{\mu}_N, \Sigma_N)$ to prevent the learner from becoming too confident over time. This is because the standard scenario with some (possibly unknown but) constant distribution over domains is a static model, and in Bayesian linear regression for static models the variance decreases monotonically with more data. Domain changes, on the other hand, call for a dynamic model, in which the variance can grow due to the dynamics of the system. One straight-forward implementation of such a dynamic model would simply multiply the variance Σ_N by a factor larger than one when a domain change is detected.

In this section, we have seen that Bayesian linear regression can be used beneficially in real-world scenarios. In particular, it can deal with incremental learning in uniform and multiple domains (cf. Figures 12 and 13) and provides predictive uncertainties which can be used to detect domain or modelling changes.

10 Conclusion and Future Work

This report extends research in the area of runtime prediction for algorithms in several directions. We reviewed previous work in the field by Leyton-Brown et al. [LBNS02, NLBD$^+$04] and showed that it can be used to predict the median runtime of the stochastic local search algorithm SAPS, the predictive accuracy being comparable to previous results for tree search algorithms. We demonstrated how different parameter configurations can be taken into account in runtime prediction and that this can be used for automatically tuning the parameters of an algorithm on a per-instance base. We showed experimental results confirming that an automatically tuned version of SAPS outperforms SAPS with its default parameter setting by a factor of more than two on the random instances from the SAT04 competition (in which SAPS placed second behind a Novelty$^+$ variant). We also showed that in some cases the automatically tuned version of SAPS is faster than SAPS with its best fixed parameter setting.

Our framework for automatically tuning algorithms on a per-instance base applies to many algorithms and many domains. Its requirements for an algorithm are currently that all the parameters to tune be continuous or ordinal (but not categorical with more than two values), a requirement we would like to drop in future work by using Gaussian processes [Wil97, Mac98, See04, Ras, RW06]. All that our framework requires to be applicable to a domain of interest is the existence of a set of computationally inexpensive features that are predictive of instance hardness. This set of features can be engineered once and for all by a domain expert, aided by automatic feature selection techniques that combine features and choose the most useful ones. The development of new methods for automatically constructing and detecting useful features is very important, but we see this research area as largely orthogonal to our own work.

We hope to further improve predictive accuracy in the future, which will immediately lead to improved performance of algorithms with automatically chosen parameter configurations. We also wish to fully exploit the capabilities of our current approach by employing a search in continuous search space for the parameter configuration that is predicted to be fastest. So far, our experiments are limited to a rather toy experiment with 16 possible parameter configurations – this unnecessary discretization is not a limitation of our approach but of our experiments, and we would like to remove this shortcoming in future work.

Another contribution of this report is the introduction of Bayesian regression methods to the field of runtime prediction. We demonstrated that the predictive uncertainty Bayesian approaches yield can be very informative in practical applications, such as learning and prediction in a scenario with multiple domains. When learning from one domain and testing on another domain, *any* supervised Machine learning approach will show poor performance since all statistical guarantees are lost. However, we demonstrated that our Bayesian approach – while also suffering from poor predictive performance – can at least quantify the uncertainty of its predictions. In short, when launched on a new instance that is very different from all training instances, the Bayesian approach will *know* that it will not perform well. For the problem of learning in multiple domains, we expect that hierarchical Bayesian models [TJBB04, Jor06] can better exploit similarities between domains and shield off adversary effects from highly dissimilar domains. We plan to study this in future work.

We deliberately kept our machine learning approach simple to start with. As a first step, we used Bayesian linear regression, but we intend to move on to more powerful approaches, such as Gaussian processes, in the near future. Gaussian processes usually take cubic time in the number of training points [RW06], but there exist efficient approximations, such as assumed density filters using a fixed set of basis functions [ZR95], online sparse Gaussian processes [CO02], and Gaussian processes using pseudo-inputs [SG06]. Fast methods, such as KD trees can also be applied to the problem [SNS06].

Runtime prediction for algorithms with various parameter configurations lends itself nicely to active learning [SWWW89, SN96, Mac92, CGJ96]. We plan to apply methods from this field to choose the most informative parameters for each instance to train on. In an incremental setting, we would like to study the tradeoff of exploitation (choosing a parameter configuration that will solve a new instance quickly) and exploration (choosing other parameter configurations from which we learn more about poorly known parameter regimes). There has already been some work on trading off exploration and exploitation in a regression setting [Thr95]. [CS05] showed how this problem should be addressed in an optimization framework.

Finally, we are also interested in algorithms that can adapt their parameters during the search. From our point of view, the most promising and most principled line of research for this is based on reinforcement learning. However, general reinforcement learning strategies, such as described in [SB98] and applied to search algorithms in [LL01], need a compact state representation. It should be studied whether supervised or even completely unsupervised machine learning algorithms could automatically learn to distinguish a small number of states to be used in the standard reinforcement learning framework.

11 Acknowledgements

We would like to express our gratitude to a number of people that helped us in the process of this work. Our Matlab implementation for runtime prediction is based on code written by Kevin Leyton-Brown. He and Holger Hoos also supported our work with much useful feedback. Eugene Nudelman thankworthily provided the basic feature computation code also used in satzilla, and Dave Tompkins's UBCSAT provided a convenient way of computing new features and run experiments.

The idea for applying Bayesian machine learning techniques to our problems formed in part during exchange with the machine learning group at Microsoft Research Cambridge, in particular with Thomas Minka, Martin Szummer, and the summer interns Ashish Kapoor and Percy Liang. Finally, Lucas Bordeaux provided us with another useful perspective on our work. He and Sathi Subbarayan also gave us feedback on an earlier draft of this report.

References

[ADL06] Belarmino Adenso-Daz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1), 2006. To appear.

[APSS05] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern sls. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, 2005.

[BB05] Roberto Battiti and Mauro Brunato. Reactive search: machine learning for memory-based heuristics. Technical Report DIT-05-058, Università Degli Studi Di Trento, Department of information and communication technology, Trento, Italy, September 2005.

[BFP$^+$05] Thomas Bousonville, Filippo Focacci, Claude Le Pape, Wim Nuijten, Frederic Paulin, Jean-Francois Puget, Anna Robert, and Alireza Sadeghin. Integration of rules and optimization in plant powerops. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-05)*, pages 1–15. Springer Verlag, 2005.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* 2006. To appear.

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

[BSPV02] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002.

[BTW95] James E. Borrett, Edward P. K. Tsang, and Natasha R. Walsh. Adaptive constraint satisfaction: The quickest first principle. Technical Report CSM-256, Dept. of Computer Science, University of Essex, Colchester, UK, November 1995.

[CB04] Tom Carchrae and J. Christopher Beck. Low knowledge algorithm control. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, 2004.

[CB05] Tom Carchrae and J. Christopher Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387, 2005.

[CGJ96] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145, 1996.

[CO02] Lehel Csato and Manfred Opper. Sparse online Gaussian processes. *Neural Computation*, 14:641–668, 2002.

[CS05] Vincent A. Cicirello and Stephen F. Smith. The max k-armed bandit. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, 2005.

[EF01] Susan L. Eppstein and Eugene C. Freuder. Collaborative learning for constraint solving. In *Principles and Practice of Constraint Programming (CP'01)*, 2001.

[EFW$^+$02] Susan L. Epstein, Eugene C. Freuder, Richard J. Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming (CP'02)*, pages 525 – 540. Springer Verlag, 2002.

[GHBF05] Cormac Gebruers, Brahim Hnich, Derek Bridge, and Eugene Freuder. Using cbr to select solution strategies in constraint programming. In *Proceedings of the 6th International Conference on Case Based Reasoning (ICCBR'05)*, pages 222–236, 2005.

[Hoo02] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 655–660. AAAI Press / The MIT Press, Menlo Park, CA, USA, 2002.

[HRG$^+$01] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry Kautz, Bart Selman, and David Maxwell Chickering. Branch and bound algorithm selection by performance prediction. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI'01)*, 2001.

[HS04] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search - Foundations & Applications.* Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.

[HTH02] Frank Hutter, Dave A.D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, Berlin, Germany, 2002.

[Hut04] Frank Hutter. Stochastic local search for solving the most probable explanation problem in Bayesian networks. Master's thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, September 2004.

[Jor06] Michael I. Jordan. *An Introduction to Probabilistic Graphical Models.* 2006. In preparation.

[KJ95] Ron Kohavi and George H. John. Automatic parameter selection by minimizing estimated error. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML-95)*, 1995.

[Knu75] Donald Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[LBNS02] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Principles and Practice of Constraint Programming (CP'02)*, 2002.

[LL98] Lionel Lobjois and Michel Lemaître. Branch and bound algorithm selection by performance prediction. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, 1998.

[LL00] Michail G. Lagoudakis and Michael L. Littman. Reinforcement learning for algorithm selection. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*, 2000.

[LL01] Michail G. Lagoudakis and Michael L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. In *Electronic Notes in Discrete Mathematics (ENDM)*. Elsevier Science Publishers, 2001.

[Mac92] David J. C. MacKay. Information-based objective functions for active data selection. *Neural Computation*, 4(4):589–603, 1992.

[Mac98] David J. C. MacKay. Introduction to Gaussian processes. In Christopher M. Bishop, editor, *Neural Networks and Machine Learning*, NATO ASI Series, pages 133–166. Kluwer Academic Press, 1998.

[Min93] Steven Minton. An analytic learning system for specializing heuristics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, 1993.

[Min96] Steven Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1):1–40, 1996.

[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, 1997.

[NLBD$^+$04] Eugene Nudelman, Kevin Leyton-Brown, Alex Devkar, Yoav Shoham, and Holger H. Hoos. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Principles and Practice of Constraint Programming (CP'04)*, 2004.

[PK01] Donald J. Patterson and Henry Kautz. Auto-walksat: a self-tuning implementation of walksat. In *Electronic Notes in Discrete Mathematics (ENDM), 9*, 2001. Presented at the LICS 2001 Workshop on Theory and Applications of Satisfiability Testing, June 14-15, 2001, Boston University, MA.

[Ras] Carl Edward Rasmussen. Gaussian processes in machine learning. Technical report, Max Planck Institute for Biological Cybernetics, 72076 Tübingen, Germany.

[Ric76] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[RW06] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006. ISBN 0-262-18253-X.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[See04] Matthias Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems*, 14(2):69–106, 2004.

[SG06] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18 (NIPS-05)*, Cambridge, MA, 2006. MIT Press.

[Sil00] Jonathan Sillito. Improvements to and estimating the cost of backtracking algorithms for constraint satisfaction problems. Master's thesis, University of Alberta, Edmonton, Canada, June 2000.

[SLE04] Tiziana Ligorio Susan L. Epstein. Fast and frugal reasoning enhances a solver for really hard problems. 2004.

[SM05] Biplav Srivastava and Anupam Mediratta. Domain-dependent parameter selection of search-based algorithms compatible with user performance criteria. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, 2005.

[SN96] Kah Kay Sung and Partha Niyogi. Active learning for function approximation. In *Advances in Neural Information Processing Systems 8 (NIPS-95)*, 1996.

[SNS06] Yirong Shen, Andrew Ng, and Matthias Seeger. Fast gaussian process regression using kd-trees. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18 (NIPS-05)*. MIT Press, Cambridge, MA, 2006.

[SWWW89] Jerome Sacks, William J. Welch, Toby J. Welch, and Henry P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4(4):409–423, November 1989.

[TH04] Dave A. D. Tompkins and Holger H. Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for SAT & MAX-SAT. In *SAT2004 — Highlights of Satisfiability Research in the Year 2004*, 2004.

[Thr95] Sebastian Thrun. Exploration in active learning. In Arbib, M.A. (ed.): The handbook of brain theory and neural networks, pages 381 – 384. mit press, 1995., 1995.

[TJBB04] Yee W. Teh, Michael I. Jordan, Matthew J. Beal, and David M. Blei. Hierarchical Dirichlet processes. Technical Report 653, Department of Statistics, University of California at Berkeley, 2004.

[Wil97] Christopher K. I. Williams. Prediction with Gaussian processes: From linear regression to linear prediction and beyond. Technical Report NCRG/97/012, Neural Computing Research Group, Dept. of Computer Science & Applied Mathematics, Aston University, Birmingham, UK, October 1997.

[ZR95] Huaiyu Zhu and Richard Rohwer. Sparse online Gaussian processes. to appear in *Neural Computation and Applications*. Neural Computing Research Group Department of Computer Science and Applied Mathematics, Aston University, Birmingham B4 7ET, UK, 1995.