# Performance Modelling and Automated Algorithm Design for NP-hard Problems

by

Lin Xu

BSc. Chemistry, Nanjing University, 1996

MSc. Chemistry, Nanjing University, 1999

MSc. Computer Science, University of Nebraska-Lincoln, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Doctor of Philosophy**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

November 2014

# Abstract

In practical applications, some important classes of problems are NP-complete. Although no worst-case polynomial time algorithm exists for solving them, state-of-the-art algorithms can solve very large problem instances quickly, and algorithm performance varies significantly across instances. In addition, such algorithms are rather complex and have largely resisted theoretical average-case analysis. Empirical studies are often the only practical means for understanding algorithms' behavior and for comparing their performance.

My thesis focuses on two types of research questions. On the science side, the thesis seeks a in better understanding of relations among problem instances, algorithm performance, and algorithm design. I propose many instance features/characteristics based on instance formulation, instance graph representations, as well as progress statistics from running some solvers. With such informative features, I show that solvers' runtime can be predicted by predictive performance models with high accuracy. Perhaps more surprisingly, I demonstrate that the solution of NP-complete decision problems (e.g., whether a given propositional satisfiability problem instance is satisfiable) can also be predicted with high accuracy.

On the engineering side, I propose three new automated techniques for achieving state-of-the-art performance in solving NP-complete problems. In particular, I construct portfolio-based algorithm selectors that outperform any single solver on heterogeneous benchmarks. By adopting automated algorithm configuration, our highly parameterized local search solver, `SATenstein-LS`, achieves state-of-the-art performance across many different types of SAT benchmarks. Finally, I show that portfolio-based algorithm selection and automated algorithm configuration could be combined into an automated portfolio construction procedure. It

requires significant less domain knowledge, and achieved similar or better performance than portfolio-based selectors based on known high-performance candidate solvers.

The experimental results on many solvers and benchmarks demonstrate that the proposed prediction methods achieve high predictive accuracy for predicting algorithm performance as well as predicting solutions, while our automatically constructed solvers are state of the art for solving the propositional satisfiability problem (SAT) and the mixed integer programming problem (MIP). Overall, my research results in more than 8 publications including the 2010 IJCAI/JAIR best paper award. The portfolio-based algorithm selector, `SATzilla`, won 17 medals in the international SAT solver competitions from 2007 to 2012.

# Preface

I am indebted to many outstanding co-authors for their contributions to the work described in this thesis. My two supervisors, Kevin Leyton-Brown and Holger Hoos, involved in all my works, and contributed in designing, discussion, and writing.

The development of SATzilla had a major impact on my PhD thesis. In collaboration with Frank Hutter, Holger Hoos, and Kevin Leyton-Brown, I designed and built all versions of SATzilla, which won 17 medals in the international SAT solver competitions from 2007 to 2012. More details of this work was described in Chapter 7 (published in [210, 211, 216]) and Chapter 8 (published in [215]).

Collaborating with Holger Hoos and Kevin Leyton-Brown, I then designed the first version of Hydra, automatic configuration of algorithms for portfolio-based selection, based on the work of SATzilla (Chapter 10.2, published in [212]). Later, with Frank Hutter joining the force, we developed an improved the version of Hydra and applied it on the domain of MIP (Chapter 10.3, published in [213]).

Both SATzilla and Hydra were based on our own performance predictive models. The runtime prediction was described in Chapter 5 (published in [208]). The solution prediction was described in Chapter 4 (published in [214]). I also collaborated with Frank Hutter to compare different prediction techniques in Chapter 6 (published in [101]).

The work of automatically building high-performance algorithms from components (Chapter 9, published in [115]) is based on the project of SATenstein, which initiated by Ashiqur KhudaBukhsh for his M.S. thesis. I contributed in supervising, designing, implementation, and writing.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgments

I gratefully acknowledge my two symmetrical co-supervisors, Holger Hoos and Kevin Leyton-Brown. They not only accepted me into the Ph.D. program eight years ago, but also made sure that I stayed on the right track. They encouraged me to develop principled ideas and fine-tuned them with me to ensure their good performance in practice. Their passion for science, creativity and high standards deeply influenced me in many ways. In finalizing this thesis, I benefited from their detailed and insightful feedback. Many thanks also to Alan Hu, the third member of my supervisory committee, for making sure I work on interesting but feasible questions and toward finishing my Ph. D.

While co-authors are acknowledged in a separate section of the front-matter, I would like to thank Frank Hutter in particular, for the fruitful collaboration and many insightful discussions. I would like to thank many other members of the Laboratory for Computational Intelligence (LCI) and the Bioinformatics, Empirical and Theoretical Algorithmics (BETA) Laboratory at UBC computer science, for many interesting chats about our respective work and friendship that made the joy of my graduate studies.

I would like to thank my family here, my wife, Ping Xiang; my son, Daniel Binbin Xu and my daughter, Grace Linglin Xu for their continuous support and for bringing sunshine into my life when all else failed. I also like to thank my mom and my two sisters back in China for supporting my choice.

# Chapter 1

# Introduction

In many practical applications, algorithm designers confront computationally hard problems. Examples are graph coloring (see, e.g., Garey and Johnson, 1979), planning and scheduling (see, e.g., Kautz and Selman, 1999), Boolean satisfiability (SAT) (see, e.g., Cook, 1971), traveling salesperson (TSP) (see, e.g., Applegate et al., 2006), software/hardware verification (see, e.g., Biere et al., 1999), protein folding (see, e.g., Fraenkel, 1993), and gene sequencing (see, e.g., Pop et al., 2002). In complexity theory, such problems belong to the complexity class of NP-complete problems ($\mathsf{NP}-\mathsf{C}$ or NPC). These are the most difficult problems in NP. If one could find a deterministic, polynomial-time solution to any NP-complete problem, then one would be able to provide a polynomial-time solution to every other problem in NP. It is widely believed that no worst-case polynomial time algorithm exists for solving NP-complete problems. The Clay Mathematics Institute has offered a one million US dollar prize for the first correct proof or disproof of whether NP is equivalent to the complexity class P, where all problems would be solved on a deterministic sequential machine in polynomial time [102].

For NP-complete problems, even the best currently known algorithms have worst-case runtimes that increase exponentially with instance size. Luckily, while these problems may be hard to solve on worst-case inputs, it is often feasible to solve large problem instances that arise in practice. However, state-of-the-art algorithms often exhibit exponential runtime variation across instances from realistic distributions, even when problem size is held constant, and conversely the same

instance can take exponentially different amounts of time to solve depending on the algorithm used [13]. There is little theoretical understanding of what causes this variation, and thus it is nontrivial to determine how long a given algorithm will take to solve a given problem instance without incurring the potentially large cost of running the algorithm. This phenomenon suggests that worst case analysis is not sufficient for studying an algorithm's behavior on practical applications. Instead, empirical studies are often the only practical means for assessing and comparing an algorithm's performance. Researchers and practitioners seek to locate features/characteristics of instances that explain when instances will be hard for a particular algorithm, choosing the most promising heuristics for designing high-performance algorithms, and finding the most efficient algorithm for an unseen instance drawn from a given instance distribution. Answers to these questions can help one to better understand and solve NP-complete problems.

My PhD work focuses on studying NP-complete problems based on empirical data and machine learning techniques, in addition to proposing automated methods for improving the effectiveness of solving problem instances from real applications. My work has four major components. For a better understanding of the nature of NP-complete problems, the relations between instance features and algorithm performance were studied for both NP-complete decision and optimization problems using supervised machine learning techniques. Furthermore, the relations between instance features and an instance's satisfiability status for decision problems were studied. Based on the successes of these studies, we set out to improve the state of the art in solving NP-complete problems. The thesis proposes three different approaches. The first is a portfolio-based algorithm selector that combines the strengths of multiple candidate solvers. Here, predictive models are used as the basis for an algorithm portfolio that selects the most promising candidate solver for an unseen instance automatically. Inspired by the successes of automated algorithm configuration that is able to find very good parameter settings for highly parameterized algorithms, the second approach suggests a new algorithm design philosophy. Unlike the traditional approach for building heuristic algorithms, the algorithm designer should include as many alternate approaches to solving the same subproblem as seem promising instead of fixing most of the design choices at development time. The optimal instantiation of heuristic algorithms

for a given instance benchmark should be automatically produced by automated algorithm configuration tools. The third approach targets domains where only one highly parameterized algorithm is competitive and combines portfolio-based algorithm selection and automated algorithm configuration together in a novel manner. By changing the performance measure in algorithm configuration, this approach automatically discovers algorithm configurations that possess the greatest potential for improving the current algorithm portfolio.

## 1.1 Science Side Research

Previous empirical studies on NP-complete problems have revealed many intriguing results. For example, problem instances with certain properties can be much harder than others for many algorithms. Mitchell et al. (1992) showed how random 3-SAT instances with clauses-to-variables ratios around 4.3 are usually harder than other random 3-SAT instances of the same size. More recent work studied the use of machine learning methods to make instance-specific predictions about solver runtimes. Leyton-Brown et al. (2002, 2009) introduced the use of such models for predicting the runtimes of solvers for solving NP-complete problems, and Nudelman et al. (2004) showed that using this approach, surprisingly accurate runtime predictions can be obtained for uniform random 3-SAT. Nudelman et al. also noticed that training models on only SAT or UNSAT instances allowed much simpler, albeit very dissimilar, models to achieve high accuracy. Since unconditional models, without considering SAT/UNSAT status, are able to predict runtimes accurately, despite the qualitative differences between the SAT and UNSAT regimes, the models must implicitly predict satisfiability status.

Motivated by this result, we investigated the feasibility of predicting the satisfiability of a previously unseen SAT instance by considering a variety of both structured and unstructured SAT instances. The empirical results proved rather promising, the classification accuracies were always better than 68% rather than 50% given by random guess. A detailed case study of uniform random 3-SAT at the phase transition revealed that the classification accuracies remained roughly constant and far above random guessing even using a single decision tree with only two simple features. Furthermore, we investigated the benefit of having a reasonably

accurate (but imperfect) classifier on runtime prediction. We improved runtime prediction by constructing hierarchical hardness models using a mixture-of-experts approach with fixed ("clamped") experts, that is, with conditional models trained on satisfiable instances and unsatisfiable instances separately. The classifier's confidence correlated with prediction accuracy, giving useful per-instance evidence on the quality of the runtime prediction. Of course, there are many other regression techniques that could be used for runtime/performance prediction. We performed a thorough study on different machine learning techniques on many NP-complete problems such as the Boolean satisfiability (SAT), the mixed integer programming (MIP), and the traveling salesperson (TSP).

## 1.2 Engineering Side Research

The wide applications of NP-complete problems facilitate the development of high-performance algorithms, while significant research and engineering efforts have led to sophisticated algorithms. In one prominent and ongoing example, the SAT community holds an annual SAT Competition/Race/Challenge (http://www.satcompetition.org/). This competition intends to provide an objective assessment of SAT algorithms, and thus to track the state of the art in SAT solving, to assess and promote new solvers, and to identify new challenging benchmarks. Solvers are judged based on their empirical performance with both speed and robustness taken into account. One observation from the competitions is that algorithm performance highly depends on the type of instances. One algorithm could be much better than others on solving one class of instances, but dramatically worse on instances from other classes (see, e.g., Le Berre et al., 2012). One possible explanation is that many practical problem instances possess some special structure. Solvers can achieve much better performance if they can exploit such structural.

One manner in which evaluations such as the SAT competition are useful is that they allow practitioners to determine which algorithm performs best for instances relevant to their problem domain. However, choosing a single algorithm on the basis of competition ranks is not always a good approach. Such a "winner-take-all" approach typically results in the neglect of many algorithms that are not competitive on average but that nevertheless offer very good performance on particular

instances. Thus, practitioners with hard problems to solve confront a potentially difficult "algorithm selection problem" [170]: which algorithm(s) should be run in order to minimize an performance objective, such as expected runtime? The *ideal* solution to the algorithm selection problem, conversely, would be to consult an oracle that knows the amount of time that each algorithm will take to solve a given problem instance, and then to select the algorithm with the best performance. Unfortunately, computationally cheap, perfect oracles of this nature are not available for SAT or any other NP-complete problem. Inspired by the success of runtime prediction, Nudelman et al. (2004) proposed an automated algorithm selection approach based on approximate performance predictors, which can be seen as a heuristic approximation to a perfect oracle. Initial trial of such an approach demonstrated promising results. In the 2003 SAT Competition, the first version of `SATzilla` [155] placed 2$^{nd}$ in two categories and 3$^{rd}$ in another.

Note that due to the nature of NP-completeness, one could not expect such approximation to be perfect without solving the instances. Therefore, we introduced several new techniques to improve the robustness of `SATzilla`, such as pre-solving, backup solver and feature cost prediction. The competition results demonstrate that my portfolio-based algorithm selectors are capable of achieving state-of-the-art performance. They won many medals in the 2007 and 2009 SAT Competitions in conjunction with the 2012 SAT Challenge (with a new selection technique based on cost-sensitive classification).

We also showed that the general framework of `SATzilla` was compatible with other performance predictors, and performed very well on other problem domains, such as MIP. Given that portfolio-based algorithm selectors often achieve state-of-the-art performance, the community could benefit from rethinking how to value individual solvers. Developing a solver that helps to improve state-of-the-art performance should be more valuable than designing a slightly on-average better solver. We developed techniques for analyzing the extent to which the performance of the state-of-the-art (*SOTA*) portfolio depends on each of their component solvers.

High-performance heuristic algorithms are able to solve very large problem instances from practical applications. However, designing them is a time-consuming task even for domain experts. Traditionally, heuristic algorithms are designed in an iterative, manual process in which most design choices are fixed at development

time, usually based on preliminary experimentation, leaving only a small number of parameters exposed to the user. Although such an approach has proven to work effectively in the past, this approach requires a significant amount of effort on the part of the domain experts. Recently, a new line of research has attempted to automate parts of the algorithm design process with cheap computing power, and achieved many successes (see, e.g., Hoos, 2008). Inspired by such work, our team proposed a new approach to heuristic algorithm design in which the designer fixes as few design choices as possible, instead exposing all promising design choices as parameters. This approach removes the burden from the algorithm designer of making early design decisions without knowing how different algorithm components will interact on problem distributions of interest. Instead, now the designer is encouraged to consider many alternative designs from known solvers in addition to novel mechanisms. Of course, such flexible, highly parameterized algorithms must be instantiated appropriately to achieve good performance on a given instance set. With the availability of advanced automated parameter configurators and cheap computational resources, finding a good parameter configuration from a huge parameter space becomes practical (see, e.g., [22, 33, 94]).

Although this general idea is not specifically tailored to a particular domain, in this work we applied it to the challenge of constructing stochastic local search (SLS) algorithms for the propositional satisfiability problem (SAT). SLS-based solvers have exhibited consistently dominant performance for several families of SAT instances; they also play an important role in state-of-the-art portfolio-based automated algorithm selection methods for SAT [210]. Our team implemented a highly parameterized SLS algorithm by drawing mechanisms from two dozen existing high-performance SLS SAT solvers and also incorporating many novel strategies and termed this `SATenstein-LS`. Similar to the "perfect human being" created by Victor Frankenstein using scavenged human body parts in the classic novel *Frankenstein*, here one scavenges components from existing high-performance algorithms for a given problem and combines them to build new high-performance algorithms. Unlike Frankenstein's creation, our algorithm is built using an automated construction process that enables one to optimize performance with minimal human effort. The design space contains a total of $2.01 \times 10^{14}$ possible instantiations, and includes most existing, state-of-the-art SLS SAT solvers

that have been proposed in the literature. With the aid of automated algorithm configuration tools, we demonstrate experimentally that our new, automatically-constructed solvers dramatically outperform the best SLS-based SAT solvers currently available on six well-known SAT instance distributions, ranging from hard random 3-SAT instances to SAT-encoded factoring and software verification problems. This makes it interesting to understand the similarities and differences between our new configurations and existing SLS algorithms. We propose an automatic, quantitative approach for visualizing the degree of similarity between a set of algorithms. Using this approach, we investigated the similarities among our `SATenstein-LS` solvers and SLS-based incumbents. This visualization demonstrates that most of our new solvers are very different from existing solvers.

Although portfolio-based algorithm selection and automated algorithm configuration have demonstrated many positive results in practice [22, 33, 94, 210], they each have some shortcomings. The former approach requires relatively significant domain knowledge, including in particular, a set of relatively uncorrelated candidate solvers. The latter approach requires no domain knowledge beyond a parameterized algorithm framework, and no human effort to target a new domain; however, it produces only a single algorithm, which is designed to achieve a high performance overall, but which may perform badly on many individual instances. This drawback is particularly serious when the instance distribution is heterogeneous. Once a state-of-the-art portfolio exists for a domain, such as `SATzilla` for various SAT distributions, the critical question to the algorithm developer is: how should new research aim to improve upon it? One approach is to build new stand-alone algorithms either by hand or by automatic configuration, with the goal of replacing the portfolio. This approach has the weakness that it reinvents the wheel: the new algorithm must perform well on all the instances for which the portfolio is already effective, and must also make additional progress.

Alternatively, one may attempt to build a new algorithm to *complement* the portfolio, which has been dubbed "boosting as a metaphor for algorithm design" [128]. The boosting algorithm in machine learning builds an ensemble of classifiers by focusing on problems that are handled poorly by the existing ensemble. The proposal is to approach algorithm design analogously, focusing on instances on which the existing portfolio performs poorly. In particular, the suggestion is

to use sampling (with replacement) to generate a new benchmark distribution that will be harder for an existing portfolio, and for new algorithms to attempt to minimize average runtime on this benchmark. Indeed, such a method was shown to be particularly effective for inducing new, hard distributions. While we agree with the core idea of aiming explicitly to build algorithms that will complement a portfolio, we have come to disagree with its concrete realization as described most thoroughly by Leyton-Brown et al. (2009), realizing that average performance on a new benchmark distribution is not always an adequate proxy for the extent to which a new algorithm would complement a portfolio. A region of the original distribution that is exceedingly hard for all candidate algorithms can dominate the new distribution, leading to stagnation.

Based on this observation, we introduced `Hydra`, a new method for automatically designing algorithms to complement a portfolio. This name was inspired by the Lernaean Hydra, a mythological, multi-headed beast that grew new heads for those cut off during its struggle with the Greek hero Hercules. `Hydra`, given only a highly parameterized algorithm and a set of instance features, automatically generates a set of configurations that form an effective portfolio. It thus does not require any domain knowledge in the form of existing algorithms. `Hydra` is an *anytime procedure*: it begins by identifying a single configuration with the best overall performance, and then iteratively adds algorithms to the portfolio. `Hydra` is also able to drop previously added algorithms when they are no longer helpful. `Hydra` offers the greatest potential benefit in domains where only one highly parameterized algorithm is competitive (e.g., certain distributions of mixed-integer programming problems), and the least potential benefit in domains where a wide variety of strong, uncorrelated solvers already exist. We performed case studies on both SAT and MIP, where `Hydra` consistently achieved significant improvements over the best existing individual algorithms designed both by experts and automatic configuration methods. More importantly, `Hydra` always at least roughly matched—and indeed often exceeded—the performance of the best portfolio of such algorithms.

## 1.3 Overview of Contributions

Overall, my research resulted in major advances in understanding a variety of NP-complete decision and optimization problems, as well as in pushing forward the state-of-the-art for solving them. My major contributions are summarized as follows.

- Features: We extended the feature set proposed by Nudelman et al. (2004) for characterizing the propositional satisfiability problem (SAT). Chapter 3 also introduces new features for other NP-complete problems (TSP and MIP). Those features were proven to be informative and have been widely used by other research groups [111].

- Predictive models: We demonstrated that simple rules can predict the solubility of uniform random 3-SAT at the phase with surprisingly high accuracy [Chapter 4]. Extensive empirical results suggest that classification accuracy does not decrease with instance size. Chapter 5 relates how to improve runtime prediction by combing classifiers with conditional hardness models into a hierarchical hardness model using a mixture-of-experts approach. Chapter 6 describes a thorough comparison of different existing and new model building techniques for SAT, MIP, and TSP. We demonstrated that random forests yield substantially better runtime predictions than previous approaches.

- Portfolio-based algorithm selection: We made significant advances in building state-of-the-art portfolio-based algorithm selectors. With many new techniques introduced in Chapter 7, `SATzilla` won the 2007 and 2009 SAT Competition, and the 2012 SAT Challenge. Due to the huge success of `SATzilla`, the paper by Xu et al. (2008) won the 2010 IJCAI-JAIR best paper prize. In addition to state-of-the-art performance, `SATzilla` is useful for evaluating solver contributions. By omitting a solver from the portfolio, We measured the contribution of this solver by computing `SATzilla`'s performance difference with and without it. Chapter 8 shows that solvers that exploited novel strategies were more valuable than those with the best overall performance. We also demonstrate that cost-sensitive classification-based

algorithm selector achieved the best performance. In fact, SATzilla2012 won the 2012 SAT Challenge by using cost-sensitive decision forests as the algorithm selector.

- Automatically building high-performance algorithms from components: We proposed a new approach to heuristic algorithm design in which the designers fix as few design choices as possible at development time, instead exposing a huge number of design choices in the form of parameters. Chapter 9 demonstrates a case study on constructing stochastic local search (SLS) algorithms for SAT. By taking components from 25 local search algorithms, we built a highly parameterized local search algorithm, SATenstein-LS, which can be instantiated as $2.01 \times 10^{14}$ different solvers. The empirical results show that the automatically constructed SATenstein-LS outperforms existing state-of-the-art solvers with both manually and automatically tuned configurations. In addition, we proposed a new representation for algorithm parameter settings, concept DAGs, and defined a novel similarity metric based on the transformation cost. We have shown that the visualization based on such similarity measure provides useful insights into algorithm design.

- Automatically configuring algorithms for portfolio-based selection: By combining the strengths of automated algorithm selection and automated algorithm configuration, we proposed a novel technique, Hydra, for automatically discovering a set of solvers iteratively with complementary strengths. The case study on SAT benchmarks (Chapter 10.2) showed that Hydra with a single solver, SATenstein-LS, significantly outperforms state-of-the-art SLS algorithms. Hydra reaches and often exceeds the performance of portfolios that use many strong local search solvers as candidate solvers. By adapting the cost-sensitive classification models and modifying method for selecting candidate configurations, we demonstrated that MIP-Hydra converges faster, and achieves strong performance for MIP (Chapter 10.3).

# Chapter 2

# Related Work

Over the last decades, considerable research efforts have led to significant progress in understanding and solving NP-complete problems. In this chapter, we review some of the work that is most relevant to this thesis. The remainder of this chapter is structured as follows. Section 2.1 introduces recent advances in empirical hardness models. Section 2.2 summarizes related work on algorithm portfolios and the algorithm selection problem. Section 2.3 discusses some automated techniques for constructing high-performance algorithms, which closely relate to my `SATenstein` work. Section 2.4 overviews the techniques for automated algorithm configuration. They play an important role in building `SATenstein` and `Hydra`. In the end, we discuss other approaches for automatically configuring algorithms for portfolio-based selection, and compare them with my `Hydra` approach.

## 2.1 Empirical Hardness Models

Most of the heuristic algorithms for solving NP-complete problems are highly complex, and thus have largely resisted theoretical average-case analysis. Instead, empirical studies are often the only practical means for assessing and comparing their performance. One recent approach for understanding the empirical hardness of computational hard problems was proposed by Leyton-Brown et al. (2002). It used linear basis-function regression to build models that predict the time required

for an algorithm to solve a given problem instance [127]. These so-called empirical hardness models can be used to evaluate the factors responsible for an algorithm's performance, or to introduce challenging instances for a given algorithm [128]. They can also be leveraged to select among several different algorithms for solving a given problem instance [128, 129, 209] and can be applied in automated algorithm configuration and tuning [92].

On a high level, empirical hardness models represent functional relations between instance characteristics and algorithm performance (e.g., CPU time). Given a set of training data (pairs of instance characteristics and algorithm performance), an empirical hardness model is trained to fit the training data using regression techniques. Later, for a new, unseen problem instance (test data point), a performance prediction can be made by evaluating the empirical hardness model on the characteristics of the test instance. The instance characteristics are very important for building good models for predicting algorithm's performance. Good instance characteristics should correlate well with algorithm performance and be cheap to compute. Algorithm performance is measured by a function that maps from algorithm output to a real value (e.g., algorithm's runtime [156], performance score [209], or solution quality found within a certain budget).

Beyond the previous work conducted in our group [128, 129, 209], there exist a few other approaches for predicting algorithm runtime. Similar models were applied by Brewer (1995) and Huang et al. (2010), although they considered only algorithms with low-order polynomial runtimes. The most closely related work is by Smith-Miles and van Hemert (2011), who employed neural network models to predict the runtime of local search algorithms for solving the traveling salesperson problem. A different approach for predicting the performance of tree search algorithms rests on predictions of the search tree size [116, 118, 138]. The literature on search space analysis has investigated measures that correlate with algorithm runtime. Prominent examples include fitness distance correlation [110], landscape ruggedness [205], and autocorrelation [83]. The typical approach is either to visually inspect the relationship between a measure and runtime (e.g., in a scatter plot), or to compute descriptive statistics, such as the Spearman correlation coefficient between the two.

Empirical hardness models have proven effective in predicting runtime for

many algorithms on a number of interesting instance distributions. In a study on combinatorial auction winner determination, a prominent NP-hard optimization problem, empirical hardness models were used to predict CPLEX's runtimes on randomly-generated problem instances using 30 characteristics [127]. Nudelman et al. (2004) used empirical hardness models to predict several tree-search algorithms' runtimes on uniform-random 3-SAT instances. One interesting observation from this work is that if instances were restricted to be either only satisfiable or only unsatisfiable, very different models were needed to make accurate runtime predictions. Furthermore, models for each type of instance were simpler and more accurate than models that must handle both types. Empirical hardness models have been applied to the study of local search algorithms as well. Based on the work of Nudelman et al. (2004), Hutter et al. (2006) used empirical hardness models to predict runtime distributions of randomized, incomplete algorithms. They also have been used in model-based algorithm configuration procedures (such as SMAC [99]) to identify promising combinations of algorithm components to evaluate.

## 2.2 Algorithm Portfolios and the Algorithm Selection Problem

With recent advances in algorithm development, many previously challenging problem instances can be quickly solved by at least some algorithms. However, one algorithm often only performs well on some small classes of instances. Hence, one possible approach for solving NP-complete problems effectively is to use multiple existing algorithms and find out the best way to allocate computational resources to each individual algorithm.

One way of using multiple existing algorithms is to build algorithm portfolios. The term "algorithm portfolio" was introduced by Huberman et al. (1997) to describe the strategy of running $k$ algorithms in parallel, potentially with each algorithm $i$ getting a share of computational resources $s_i$ ($i = 1, ..., k$). Gomes and Selman (2001) built a portfolio of stochastic algorithms for quasi-group completion and logistics scheduling problems. Low-knowledge algorithm control by Carchrae and Beck (2005) employed a portfolio of anytime algorithms, prioritizing

each algorithm according to its performance so far. Dynamic algorithm portfolios by Gagliolo and Schmidhuber (2006) also ran several algorithms at once, where an algorithm's priority depends on its predicted runtime conditioned on the fact that it has not yet found a solution. In a recent approach, black-box techniques were used for learning how to interleave the execution of multiple heuristics to improve average-case performance based on the development of solution quality [190].

Besides algorithm portfolios, there is another line of research that takes advantage of multiple algorithms. Given a computationally hard problem instance and multiple algorithms with relatively uncorrelated performance, it is natural to define an "algorithm selection problem" [170]: which algorithm(s) should be used to minimize some performance objective, such as classification error (for solving a classification problem) or expected runtime (e.g., for solving SAT)? Much early work on solving the algorithm selection problem focused on selecting learning algorithms for solving classification problems [1, 137, 159]. Instead of using the term "algorithm selection", they used the term "meta-learning". For example, Aha (1992) used rule-based learning algorithms to decide which classification algorithm should be used based on a number of characteristics of the test data sets. Later, this learning approach had been applied to many other problem domains. Arinze et al. (1997) demonstrated a knowledged-based system that selected among three forecasting methods with six features for solving a time-series forecasting problem. Lobjois and Lemaître (1998) studied the problem of selecting between branch-and-bound algorithms based on an estimation of search tree size due to Knuth (1975). Gebruers et al. (2005) employed case-based reasoning to select a solution strategy for instances of a constraint programming problem. Various authors have proposed classification-based methods for algorithm selection [55, 65, 66, 84]. Note that one problem with such approaches is that they typically use an error metric that penalizes all misclassifications equally, regardless of their real cost. However, using a sub-optimal algorithm is acceptable in solving an algorithm selection problem if the difference between its performance and that of the best algorithm is small. The studies of Leyton-Brown et al. (2003) and Nudelman et al. (2004) were most closely related to my own work of `SATzilla`. (Nudelman et al. (2004) indeed coined the name `SATzilla`.) Specifically, they built empirical hardness models to predict the runtime of given algorithms using regression techniques. By

modeling a portfolio of algorithms and choosing the algorithm predicted to have the lowest runtime, empirical hardness models can serve as the basis for building an automatic system that solves the algorithm selection problem. In fact, such a system can also be viewed as a type of classification that takes the real cost of misclassification into account.

Algorithm selection is closely related to algorithm portfolios. They work for the same reason—they exploit lack of correlation in the best-case performance of several algorithms in order to obtain improved performance in the average case. In fact, algorithm selection can be viewed as a special type of algorithm portfolios such that the algorithm with the best performance has 100% share of computational resources. To more clearly describe algorithm portfolios in a broad sense, we introduced some new terminology [210]. An $(a,b)$-of-$n$ portfolio is defined as a procedure for selecting among a set of $n$ algorithms with the property that if no algorithm terminates early, at least $a$ and no more than $b$ algorithms will be executed. We consider a portfolio to have terminated early if it solves the problem before one of the solvers has a chance to run, or if one of the solvers crashes. For brevity, we also use the terms $a$-of-$n$ portfolio to refer to an $(a,a)$-of-$n$ portfolio, and $n$-portfolio for an $n$-of-$n$ portfolio. It is also useful to distinguish how solvers are run after being selected. Portfolios can be parallel, sequential, or partly sequential (some combination of the two). Thus traditional algorithm portfolios can be described as parallel $n$-portfolios. In contrast, pure algorithm selection procedures are 1-of-$n$ portfolios.

Some approaches fall between these two extremes, making decisions about which algorithms to use on the fly instead of committing in advance to a fixed number of candidates. Lagoudakis and Littman (2001) employed reinforcement learning to solve an algorithm selection problem at each decision point of a DPLL solver for SAT in order to select a branching rule. Similarly, Samulowitz and Memisevic (2007) employed classification to switch between different heuristics for QBF solving during the search. These approaches can be viewed as $(1,n)$-of-$n$ portfolios.

In the recent SAT Competitions/Challenge, portfolio-based solvers achieved many successes. Our own portfolio-based algorithm selector, `SATzilla`, won a total of 17 medals in the 2007 and 2009 SAT Competitions and the 2012 SAT

Challenge. A simple parallel portfolio, `ppfolio` [171] with 5 candidate solvers performed very well in the 2011 SAT Competition and the 2012 SAT Challenge. `3S` [112] achieved remarkable performance by using nearest neighbor classification. It also used a powerful fixed solver schedule as the pre-solving step.

## 2.3 Automated Construction of Algorithms

Designing high-performance heuristic algorithms for solving NP-complete problems is often a time-consuming task. The traditional approach requires significant efforts from domain experts to select design choices, and pick default parameters based on preliminary experimentation. However, the demand for high-performance solvers for difficult combinatorial problems in practical applications has increased sharply. With ever-increasing availability of cheap computing power, a new line of research has automated parts of the algorithm design process (see also Hoos, 2008) and achieved many successes [31, 50, 51, 54, 64, 145, 157, 158, 207, 210].

Here we discuss three closely related lines of previous work in more detail. First, Minton (1993) used meta-level theories to produce distribution-specific versions of generic heuristics, and then found the most useful combination of these heuristics by evaluating their performance on a small set of test instances. He focused on producing distribution-specific versions of candidate heuristics, and only considered at most 100 possible heuristics. The performance of the resulting algorithms was comparable with that of algorithms designed by a skilled programmer, but not an algorithm expert. Second, Gratch and Dejong (1992) presented a system that starts with a STRIPS-like planner, and augments it by incrementally adding search control rules. Finally and most relatedly, Fukunaga (2002) proposed a genetic programming approach that has a goal similar to the one we pursued in our work on SATenstein 9: the automated construction of local search heuristics for SAT. Fukunaga considered a potentially unbounded design space, based only on GSAT-based and WalkSAT-based SLS algorithms up to the year 2000. His candidate variable selection mechanisms were evaluated on Random 3-SAT instances and graph coloring instances with at most 250 variables. While Fukunaga's approach could in principle be used to obtain high-performance solvers for specific types of SAT instances, to our knowledge this potential has never been realized;

the best automatically constructed solvers obtained by Fukunaga only achieved a performance level similar to that of the best WalkSAT variants available in 2000 on moderately-sized SAT instances. In contrast, as mentioned in Chapter 1, our new `SATenstein-LS` solvers performs substantially better than current state-of-the-art SLS-based SAT solvers on a broad range of challenging, modern SAT instances. We consider a huge but bounded combinatorial space of algorithms based on components taken from 25 of the best SLS algorithms for SAT currently available, and we use an off-the-shelf, general-purpose algorithm configuration procedure to search this space. Our target distribution contains instances with up to 4 978 variables.

## 2.4 Automated Algorithm Configuration Tools

Recently, considerable attention has been paid to the problem of automated algorithm configuration [3, 12, 50, 64, 93, 96]. A variety of black-box, automated configuration procedures have been proposed. They take as input a highly parameterized algorithm, a set of benchmark instances, and a performance metric, and then optimize the algorithm's empirical performance automatically. These approaches can be categorized into two major families: model-based approaches that learn a response surface over the parameter space, and model-free approaches that do not. Most of the early approaches were only able to handle relatively small numbers of numerical (often continuous) parameters. Some relatively recent approaches permit both larger numbers of parameters and/or categorical domains, in particular Composer [63], F-Race [21–23], and ParamILS [94, 96, 97].

Automated algorithm configuration procedures have been applied to optimize a variety of parametric algorithms. Gratch and Chien (1996) successfully applied their Composer system to optimize an algorithm for scheduling communication between a collection of antennas and spacecraft in deep space. F-Race and its extensions have been used to optimize numerous algorithms, including iterated local search for the quadratic assignment problem, ant colony optimization for the traveling salesperson problem, and the best-performing algorithm submitted to the 2003 timetabling competition [23]. Our group successfully used various versions of ParamILS to configure algorithms for a wide variety of problem domains [94,

96, 97].

## 2.5 Automatically Configuring Algorithms for Portfolio-Based Selection

In domains where only one highly parameterized algorithm is competitive (e.g., certain distributions of mixed-integer programming problems), how should we build a strong portfolio-based algorithm selector for a given (potentially heterogeneous) distribution? Applying automated algorithm configuration tools can improve the overall performance. However, the resulting single configuration may perform poorly on some subset of instance. Meanwhile, due to the absent of multiple strong and uncorrelated candidate solvers, algorithm selection approaches cannot give the edge over a single configuration. One possible solution is combining the above two techniques and performing instance-specific selection from an automatically generated set of algorithm configurations.

Beyond our work on `Hydra`, there exist a few other approaches for solving this problem. Stochastic offline programming (SOP) [140] assumes that each of these algorithms has a particular structure, iteratively sampling from a distribution over heuristics and using the sampled heuristic for one search step. It clusters the instances based on features and then configures one algorithm for each cluster. A custom optimization method is used for building its set of algorithms.

Later, the same research group improved this approach with Instance-Specific Algorithm Configuration (ISAC) [111]. It first divides instance sets into clusters based on instance features using the *G*-means clustering algorithm, then applies an algorithm configurator to find a good configuration for each cluster. At runtime, `ISAC` computes the distance in feature space to each cluster centroid and selects the configuration for the closest cluster.

We note two theoretical problems with this approach. First, `ISAC`'s clustering is solely based on distance in feature space, ignoring the importance of each feature to runtime. Thus, `ISAC`'s performance can change dramatically if additional features are added (even if they are uninformative). Second, no amount of training time allows `ISAC` to recover from a misleading initial clustering or an algorithm configuration run that yields poor results. Nevertheless, `ISAC` substan-

tially outperformed solvers with default configurations and configurations obtained by automated algorithm configuration tools on a set covering problem, MIP, and SAT [111].

# Chapter 3

# Domains of Interest

Computationally hard combinatorial problems are ubiquitous in AI. This thesis focuses on some fundamental problems in computer science that have wide real-world application. In particular, we applied machine learning techniques to study the empirical hardness of the propositional satisfiability problem (SAT), the mixed integer programming problem, and the traveling salesperson problem (TSP). We also developed many meta-algorithmic techniques that improved the state of the art for solving SAT and MIP. This chapter first gives overviews on problem domain, and prominent solver for each domain, then introduces the sets of features for characterizing problem instances in conjunction with benchmarks used for case studies. [1]

## 3.1 Propositional Satisfiability (SAT)

The propositional satisfiability problem (SAT) asks, for a given propositional formula $F$, whether there exists a complete assignment of truth values to the variables of $F$ under which $F$ evaluates to true [83]. $F$ is considered satisfiable if there exists at least one such assignment, otherwise the formula is labeled unsatisfiable. A SAT instance is usually represented in conjunctive normal form (conjunction of disjunctions), where each disjunction has one or more literals, each of which is either a variables or the negation of variables. These disjunctions are called clauses. Thus

[1] This chapter is based on the joint work with Ashiqur KhudaBukhsh, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown [101, 115, 209, 210].

the goal for a SAT solver is to find a variable assignment that satisfies all clauses or to prove that no such assignment exists. For example, a solution of formula $(A \vee B \vee C) \wedge (\neg B \vee \neg C)$ is $A = true, B = false, C = false$. SAT is one of the most fundamental problems in computer science. Indeed, there are entire conferences and journals devoted to the study of this problem. Another important reason for interest in SAT is that instances of other NP-complete problems will be encoded into SAT and solved by SAT solvers. This approach has been shown effective for solving several real-world applications, including planning [113, 114], scheduling [39], graph coloring [202], bounded model checking [20], and formal verification [189].

Over the past decades, considerable research and engineering efforts have been invested into designing and optimizing algorithms for SAT solving. Today's high-performance SAT solvers include tree-search algorithms [41, 44, 46, 73, 121, 139], local search algorithms [80, 91, 105, 131, 178, 179], and resolution-based preprocessors [9–11, 40, 42, 192]. Here, we will give a brief introduction of the most popular SAT solving methods: tree search and local search.

### 3.1.1 Tree Search for SAT

A tree-search algorithm attempts to locate solutions to a problem instance in a systematic manner. It guarantees that eventually a solution is found if there exists one, or the algorithm will report that no solution exists. In other words, tree-search is complete. Most modern tree-search algorithms for SAT are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [41]. This procedure explores a binary search tree in which each node corresponds to assigning a truth value to one variable (that value is then fixed for all subtrees beneath that node). Since the search space size increases exponentially with the number of variables, simple backtrack search becomes rapidly infeasible even for relatively small problem instances. Fortunately, in many cases, it is possible to prune large parts of the search tree that do not contain any solution. One of the key techniques used in SAT solving for reducing the size of the search tree is unit propagation. When SAT instances are represented in conjunctive normal form, any clause containing a literal with a "true" assignment can be deleted, and all literals with "false" assignments can be eliminated from the clauses. Clauses with only one literal are termed unit clauses.

The literal in a unit clause must be assigned a "true" value. Furthermore, assigning a value to a variable in a unit clause may lead to more unit clauses. Therefore, unit propagation is a procedure that propagates the consequences of particular unit clause's literal assignment down the search tree and prunes parts of the search tree that do not contain any solution.

Recent advances in tree search algorithms include clause learning [139], preprocessing [45], backbone detection [44], and belief propagation [85]. These techniques are used for intelligent backtracking, simplifying the original formula, determining the best variable for branching, and finding the most promising assignments, respectively.

### 3.1.2 Local Search for SAT

Another common approach for solving hard combinatorial problems is local search. A local search algorithm starts at some location in the space of candidate solutions and subsequently moves from the present location to a neighboring location. There are many ways to define a neighborhood relation. For SAT, the neighbors of a candidate solution (a complete assignment) are usually the candidates only differing from the current one by a single variable assignment. Typically, every candidate solution has more than one neighbor; the choice of which one to move to is based on information mainly related to the candidates in the neighborhood of the current one, such as the number of unsatisfiable clauses for each neighbor. In contrast to tree search algorithms, typical local search algorithms are incomplete: there is no guarantee that an existing solution will eventually be found within limit amount of time, nor can unsatisfiability ever be proven.

The basic local search framework for SAT solving [83] is as follows. Given a propositional formula $F$ with $n$ variables, first randomly pick a complete variable assignment that corresponds to a point in the solution space. Then check whether the current assignment satisfies $F$. If so, terminate and report the current assignment as a solution. Otherwise, modify the current assignment (i.e., visit a neighboring location) by selecting a variable based on some predefined scoring function and changing its value from "true" to "false" or vice versa. This procedure repeats until a solution is found or a maximal number of steps have been performed.

Many local search algorithms can get stuck in a small part of the solution space (a situation called search stagnation), and they are unable or unlikely to escape from this condition without some special mechanisms. In order to avoid search stagnation, modern local search algorithms are typically randomized, leading to stochastic local search (SLS) [83]. For example, `WalkSAT` avoids search stagnation by using a random walk strategy that randomly changes the value of a variable in an unsatisfied clause [179]. Currently, much research in local search focuses on finding good tradeoffs of intensification (more intensely searching a promising small part of the solution space) and diversification (exploring other regions of the solution space).

Existing SLS-based SAT solvers can be grouped into four broad categories: GSAT-based algorithms [178], WalkSAT-based algorithms [179], dynamic local search algorithms [91, 195], and $G^2$WSAT variants [131]. `SATenstein-LS`, the highly parameterized algorithm framework described in Chapter 9, takes components from solvers from each of these categories; therefore, we describe the major features in detail for each of these categories in the following subsections.

**Category 1: GSAT-based Algorithms**

`GSAT` [178] was one of the earliest SLS SAT solvers. At each step, `GSAT` computes the score of each variable using a scoring function, then flips the variable (changes the value from $true$ to $false$ or from $false$ to $true$) with the best score. The score of a variable depends on two quantities, *MakeCount* and *BreakCount*. The *MakeCount* of a variable with respect to an assignment is the number of previously-unsatisfied clauses that will be satisfied if the variable is flipped. Similarly, the *BreakCount* of a variable with respect to an assignment is the number of previously-satisfied clauses that will be unsatisfied if the variable is flipped. The scoring function of `GSAT` is *MakeCount - BreakCount*.

Variants of `GSAT` introduced many techniques that were later used by other SLS solvers. For example, `GWSAT` [177] performs a conflict-directed random walk step with probability *wp*, otherwise it performs a regular `GSAT` step. Conflict-directed random walk is an example of a search diversification strategy that was later used by many SLS solvers. `GSAT` randomly picks a variable if multiple vari-

ables have the same score. HSAT [57] introduces a new tie-breaking scheme in which ties are broken in favor of the least-recently-flipped variable. In subsequent SLS solvers, breaking ties randomly and breaking in the favor of the least-recently-flipped variable were prominent tie-breaking schemes. GSAT now has only historical importance, as there is a substantial performance gap between GSAT and recent state-of-the-art SLS solvers.

**Category 2: WalkSAT-based Algorithms**

The major difference between WalkSAT algorithms and GSAT algorithms is the neighborhood each considers. For a WalkSAT algorithm, the neighborhood consists of the variables appearing in all currently unsatisfied clauses rather than the full set of variables. At each search step, a WalkSAT algorithm first picks an unsatisfied clause (e.g., uniformly at random), and then flips a variable from that clause depending on some heuristic. WalkSAT/SKC [179] was one of the earliest WalkSAT algorithms, and has a scoring function that only depends on *BreakCount*.

Novelty [142] and its several variants are among the most prominent WalkSAT algorithms. Novelty picks a random unsatisfied clause and computes the variables with highest and second-highest scores with the same scoring function as GSAT. Ties are broken in favor of the least-recently-flipped variable. If the variable with the highest score is not the most recently flipped variable within the clause, then it is deterministically selected for flipping. Otherwise, it is selected with probability (1 - $p$), where $p$ is a parameter termed the *noise setting* (with probability $p$, the second-best variable is selected). The idea of considering flip history is exploited in various ways in different SLS solvers, such as the age of a variable (e.g., in Novelty), flip counts (e.g., in VW [166]). To prevent stagnation (getting stuck in local minima), Novelty is often augmented with a probabilistic conflict-directed random walk [79]. Recent Novelty variants (e.g., adaptNovelty$^+$ [80]) also use a reactive mechanism that adaptively changes the *noise* parameter. This reactive mechanism is extended to many SLS solvers [135] and often yields improved performance.

### Category 3: Dynamic Local Search Algorithms

The most prominent feature of dynamic local search (DLS) algorithms is the use of "clause penalties" or "clause weights". At each step, the penalty of an unsatisfied clause is increased (this increase can be additive [195] or multiplicative [91]). In this manner, information that pertains to the *difficulty* of solving a given clause is recorded in its associated clause penalty. In order to prevent an unbounded increase in weights and to emphasize the most recent information about the difficulty of a given clause, occasional *smoothing* steps are performed to reduce them. The scoring function is the sum of the clause penalties of all unsatisfied clauses. For prominent DLS solvers, such as `SAPS`, `RSAPS` [91], and `PAWS` [195], the neighborhood consists of variables that appear in at least one unsatisfied clause.

### Category 4: `G`$^2$`WSAT` Variants

G$^2$WSAT [131] can be viewed as a combination of the GSAT and WalkSAT architectures. Similar to GSAT, G$^2$WSAT has a deterministic greedy component that examines a large number of variables belonging to a *promising list* data structure that contains *promising decreasing variables* (defined below). If the list has at least one promising decreasing variable, G$^2$WSAT deterministically selects the variable with the best score for flipping. Ties are broken in favor of the least-recently-flipped variable. If the list is empty, G$^2$WSAT executes its stochastic component, a `Novelty` variant that belongs to the WalkSAT architecture.

The definition of a *promising decreasing variable* is somewhat technical. A variable $x$ is said to be *decreasing* with respect to an assignment $A$ if $\text{score}_A(x) > 0$. A *promising decreasing variable* is defined as follows:

1. For the initial random assignment $A$, all *decreasing* variables with respect to $A$ are *promising*.
2. Let $x$ and $y$ be two different variables where $x$ is not *decreasing* with respect to $A$. If, after $y$ is flipped, $x$ becomes *decreasing* with respect to the new assignment $A'$, then $x$ is a *promising decreasing variable* with respect to $A'$.
3. As long as a *promising decreasing variable* is *decreasing*, it remains *promising* with respect to subsequent assignments in local search.

Apart from G$^2$WSAT [131], all G$^2$WSAT variants use the reactive mechanism found in adaptNovelty$^+$ [79]. gNovelty$_+$ [161], the winner of 2007 SAT Competition in the random satisfiable category, also uses clause penalties and smoothing found in dynamic local search algorithms [195].

#### UBCSAT

UBCSAT [198] is an SLS solver implementation and experimentation environment for SAT. It has already been used to implement many existing high-performance SLS algorithms from the literature (e.g., SAPS [91], adaptG$^2$WSAT+ [135]). These implementations generally match or exceed the efficiency of implementations by the original authors. UBCSAT implementations have therefore been widely used as reference implementations (see, e.g., [115, 166]) for many well-known local search algorithms. In addition, it also provides a rich interface that includes numerous statistical and reporting features facilitating empirical analysis of SLS algorithms.

Many existing SLS algorithms for SAT share common components and data structures. The general design of UBCSAT allows for the reuse and extension of such common components and mechanisms. This rendered UBCSAT a suitable environment for the implementation of highly-parameterized local search algorithms, such as our SATenstein-LS solver described in Chapter 9.

### 3.1.3 SAT Features

For the propositional satisfiability (SAT) problem, we used 138 features listed in Figure 3.1. Since a preprocessing step can significantly reduce the size of the CNF formula (particularly for industrial type instances), we chose to apply the preprocessing procedure SatElite [45] on all instances first, and then to compute instance features on the preprocessed instances. The first 90 features, except Features 22–26 and 32–36, were introduced by Nudelman et al. [156]. They can be categorized as *problem size* features (1–7), *graph-based* features (8–36), *balance* features (37–49), *proximity to horn formula* features (50–55), *DPLL probing* features (56–62), *LP-based* features (63–68), and *local search probing* features (69–90).

We incrementally introduced additional features in our work on SATzilla [210,

**Problem Size Features:**

1–2. **Number of variables and clauses in original formula**: denoted `v` and `c`, respectively

3–4. **Number of variables and clauses after simplification with `SatElite`**: denoted `v'` and `c'`, respectively

5–6. **Reduction of variables and clauses by simplification**: (`v-v'`)/`v'` and (`c-c'`)/`c'`

7. **Ratio of variables to clauses**: `v'`/`c'`

**Variable-Clause Graph Features:**

8–12. **Variable node degree statistics:** mean, variation coefficient, min, max, and entropy

13–17. **Clause node degree statistics:** mean, variation coefficient, min, max, and entropy

**Variable Graph Features:**

18–21. **Node degree statistics:** mean, variation coefficient, min, and max

22–26. **Diameter:** mean, variation coefficient, min, max, and entropy

**Clause Graph Features:**

27–31. **Node degree statistics:** mean, variation coefficient, min, max, and entropy

32–36. **Clustering Coefficient:** mean, variation coefficient, min, max, and entropy

**Balance Features:**

37–41. **Ratio of positive to negative literals in each clause:** mean, variation coefficient, min, max, and entropy

42–46. **Ratio of positive to negative occurrences of each variable:** mean, variation coefficient, min, max, and entropy

47–49. **Fraction of unary, binary, and ternary clauses**

**Proximity to Horn Formula:**

50. **Fraction of Horn clauses**

51–55. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max, and entropy

**DPLL Probing Features:**

56–60. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256

61–62. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes

**LP-Based Features:**

63–66. **Integer slack vector:** mean, variation coefficient, min, and max

67. **Ratio of integer vars in LP solution**

68. **Objective value of LP solution**

**Local Search Probing Features, based on 2 seconds of running each of SAPS and GSAT:**

69–78. **Number of steps to the best local minimum in a run:** mean, median, variation coefficient, 10th and 90th percentiles

79–82. **Average improvement to best in a run:** mean and coefficient of variation of improvement per step to best solution

83–86. **Fraction of improvement due to first local minimum:** mean and variation coefficient

87–90. **Best solution:** mean and variation coefficient

**Clause Learning Features (based on 2 seconds of running Zchaff_rand):**

91–99. **Number of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

100–108. **Length of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

**Survey Propagation Features**

109–117. **Confidence of survey propagation:** For each variable, compute the higher of $P(true)/P(false)$ or $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

118–126. **Unconstrained variables:** For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

**Timing Features**

127–138. **CPU time required for feature computation:** one feature for each of 12 computational subtasks

**Figure 3.1:** 12 groups of SAT features

211] since 2007. Our new *diameter* features 22–26 are based on the variable graph [71]. For each node $i$ in that graph, we compute the longest of the shortest path between $i$ and any other node. As with most of the features that follow, we then compute various statistics over this vector (e.g., , mean, max). Our new *clustering coefficient* features 32–36 measure the local cliqueness of the clause graph. For each node in the clause graph, let $p$ denote the number of edges present between the node and its neighbours, and let $m$ denote the maximum possible number of such edges; we compute $p/m$ for each node.

Our new *clause learning* features (91–108) are based on statistics gathered in 2-second runs of `Zchaff_rand` [139]. We measure the number of learned clauses (features 91–99) and the length of the learned clauses (features 100–108) after every 1000 search steps.

Our new *survey propagation* features (109–126) are based on estimates of variable bias in a SAT formula obtained using probabilistic inference [86]. We used `VARSAT`'s implementation to estimate the probabilities that each variable is true in every satisfying assignment, false in every satisfying assignment, or unconstrained. Features 109–117 measure the confidence of survey propagation (that is, $\max(P_{\text{true}}(i)/P_{\text{false}}(i), P_{\text{false}}(i)/P_{\text{true}}(i))$ for each variable $i$) and features 118–126 are based on the $P_{\text{unconstrained}}$ vector.

Finally, our new *timing features* (127–138) measure the time taken by 12 different blocks of feature computation code: instance preprocessing by `SatElite`, problem size (1–6), variable-clause graph (clause node) and balance features (7, 13–17, 37–41, 47–49); variable-clause graph (variable node), variable graph and proximity to Horn formula features (8–12, 18–21, 42–46, 50–55); diameter-based features (22–26); clause graph features (27–36); unit propagation features (56–60); search space size estimation (61–62); LP-based features (63–68); local search probing features (69–90) with SAPS and GSAT; clause learning features (91–108); and survey propagation features (109–126).

The cost of computing these features depends on the size of the SAT instances. Normally, feature computation takes less than 200 CPU seconds on a Intel Xeon 3.2GHz CPU, but it may take over 1000 seconds for very large instances (with millions of variables) from industrial applications.

### 3.1.4 SAT Benchmarks

Many interesting SAT benchmarks are used for studying empirical hardness and constructing high-performance portfolio solvers. They come from three major sources: SAT Competitions and SAT Races, instance generators, and other sources.

*SAT Competitions and SAT Races* These benchmarks comprise instances from 2002-2011 SAT competitions in addition to 2006-2010 SAT Races. For each competition, the instances were divided into three categories: Industrial/application (`INDU`/`APP`), Handmade/Crafted (`HAND`/`CRAFTED`), and Random (`RAND`). SAT races only considered instances from industrial/application. These are very heterogeneous benchmarks with the number of instances limited by the actual data used in the competitions/races.

*Instance generators* Many instance generators have been used for generating random and structured instances. Detailed information about instance generators and their parameters are listed as follows.

- **`rand3-fix/R3SAT`:** uniform-random 3-SAT at the solubility phase transition ($c = 4.258 \cdot v + 58.26 \cdot v^{-2/3}$) [32, 180]. The satisfiable/unsatsfiable ratio is approximately 50/50.
- **`rand3-var`:** uniform-random 3-SAT with clauses-to-variables ratio randomly selected from 3.26 to 5.26.
- **`QCP`:** SAT-encoded quasi-group completion problem: the task of determining whether the missing entries of a partial Latin square can be filled in to obtain a complete Latin square. We generated instances around the solubility phase transition using the parameters given by Gomes and Selman (1997) (order $O \in [10, ..., 30]$; holes $H = h \times O^{1.55}$, $h \in [1.2, ..., 2.2]$).
- **`SW-GCP`:** SAT-encoded graph-coloring on small-world graphs [58] with ring lattice size $S \in [100, ..., 400]$, nearest neighbors connected 10, rewiring probability $2^{-7}$, chromatic number 6.
- **`HGEN`**: satisfiable only, random instances generated using HGEN2 [76].

- **FAC**: SAT-encoded factoring problems based on prime numbers $\in [3000, ..., 4000]$ [200].

- **CBMC(SE)** : SAT-encoded software verification instances based on a binary search algorithm [34] with array size $s \in [1, ..., 2000]$ and loop-unwinding values $n \in [4, 5, 6]$. To reduce the size of the original instances, we preprocessed these instances with `SatElite` [45].

*Other sources* Some benchmarks were downloaded from industrial users with real applications.

- **IBM** This distribution of SAT-encoded bounded model checking instances comprises 765 instances generated by Zarpas (2005); these instances were downloaded from the IBM Formal Verification Benchmarks Library.

- **SWV**: This distribution of SAT-encoded software verification instances comprises 604 instances generated with the CALYSTO static checker [7], used for the verification of five programs: the spam filter Dspam, the SAT solver HyperSAT, the Wine Windows OS emulator, the gzip archiver, and a component of xinetd (a secure version of inetd).

## 3.2 Mixed Integer Programming (MIP)

Mixed integer programming (MIP) is a general approach for representing constrained optimization problems with integer-valued and continuous variables. Because MIP serves as a unifying framework for NP-complete optimization problems and combines the expressive power of integrality constraints with the efficiency of continuous optimization, it is widely used both in academia and industry. MIP used to be studied mainly in operations research, but has recently become an important tool in AI, with applications ranging from auction theory [125] to computational sustainability [62]. Furthermore, several recent advances in MIP solving have been achieved with AI techniques [59, 97].

### 3.2.1 IBM ILOG CPLEX

One important advantage of the MIP representation is that broadly applicable solvers can be developed in a problem-independent manner. IBM ILOG's `CPLEX` solver is particularly well known for achieving strong practical performance; it is used by over 1 300 corporations (including one-third of the Global 500) and researchers at more than 1 000 universities [103]. CPLEX principally uses a branch and cut algorithm that essentially solves a series of relaxed LP subproblems. In order to find the optimal solution more effectively, additional cuts and sophisticated branching strategies are employed at these subproblems. CPLEX also uses heuristics that help finding initial good solutions, while it includes a sophisticated mixed integer preprocessing system. There are some other commercial/non-commercial solvers for MIP, such as XPRESS, Gurobi, and SCIP. Each of them offers its own advantages. For example, the XPRESS MIP Optimizer uses a sophisticated branch and bound algorithm to solve MIP problems and is well known for its ability to quickly find high quality solutions [147].

State-of-the-art MIP solvers typically expose many parameters to end users; for example, `CPLEX` 12.1 comes with a 221-page parameter reference manual describing 135 parameters.

### 3.2.2 MIP Features

Figure 3.2 summarizes 121 features for mixed integer programs (i.e., MIP instances). These include 101 features based on existing work [90, 111, 130], 15 new *probing* features, and 5 new *timing* features. Features 1–101 are primarily based on features for the combinatorial winner determination problem from our group's past work [130], generalized to MIP and previously only described in a Ph.D. thesis [90]. These features can be categorized as *problem type & size* features (1–25), *variable-constraint graph* features (26–49), *linear constraint matrix* features (50–73), *objective function* features (74–91), and *LP-based* features (92–95). We also integrated ideas from the feature set used by Kadioglu et al., 2010 (*right-hand side* features (96–101) and the computation of separate statistics for continuous variables, non-continuous variables, and their union). We extended existing features by adding richer statistics where applicable: medians, variation coefficients (vc),

**Problem Type (trivial):**

1. **Problem type**: LP, MILP, FIXEDMILP, QP, MIQP, FIXEDMIQP, MIQP, QCP, or MIQCP, as attributed by CPLEX

**Problem Size Features (trivial):**

2–3. **Number of variables and constraints**: denoted $n$ and $m$, respectively

4. **Number of non-zero entries in the linear constraint matrix, $\boldsymbol{A}$**

5–6. **Quadratic variables and constraints**: number of variables with quadratic constraints and number of quadratic constraints

7. **Number of non-zero entries in the quadratic constraint matrix, $\boldsymbol{Q}$**

8–12. **Number of variables of type**: Boolean, integer, continuous, semi-continuous, semi-integer

13–17. **Fraction of variables of type** (summing to 1): Boolean, integer, continuous, semi-continuous, semi-integer

18-19. **Number and fraction of non-continuous variables** (counting Boolean, integer, semi-continuous, and semi-integer variables)

20-21. **Number and fraction of unbounded non-continuous variables**: fraction of non-continuous variables that has infinite lower or upper bound

22-25. **Support size**: mean, median, vc, q90/10 for vector composed of the following values for bounded variables: domain size for binary/integer, 2 for semi-continuous, 1+domain size for semi-integer variables.

**Variable-Constraint Graph Features (cheap):** each feature is replicated three times, for $X \in \{C, NC, V\}$

26–37. **Variable node degree statistics**: characteristics of vector $(\sum_{c_j \in C} \mathbb{1}(A_{i,j} \neq 0))_{x_i \in X}$: mean, median, vc, q90/10

38–49. **Constraint node degree statistics**: characteristics of vector $(\sum_{x_i \in X} \mathbb{1}(A_{i,j} \neq 0))_{c_j \in C}$: mean, median, vc, q90/10

**Linear Constraint Matrix Features (cheap):** each feature is replicated three times, for $X \in \{C, NC, V\}$

50–55. **Variable coefficient statistics**: characteristics of vector $(\sum_{c_j \in C} A_{i,j})_{x_i \in X}$: mean, vc

56–61. **Constraint coefficient statistics**: characteristics of vector $(\sum_{x_i \in X} A_{i,j})_{c_j \in C}$: mean, vc

62–67. **Distribution of normalized constraint matrix entries, $\boldsymbol{A}_{i,j}/b_i$**: mean and vc (only of elements where $b_i \neq 0$)

68–73. **Variation coefficient of normalized absolute non-zero entries per row** (the normalization is by dividing by sum of the row's absolute values): mean, vc

**Objective Function Features (cheap):** each feature is replicated three times, for $X \in \{C, NC, V\}$

74–79. **Absolute objective function coefficients** $\{|c_i|\}_{i=1}^n$: mean and stddev

80–85. **Normalized absolute objective function coefficients** $\{|c_i|/n_i\}_{i=1}^n$, where $n_i$ denotes the number of non-zero entries in column $i$ of $\boldsymbol{A}$: mean and stddev

86–91. **squareroot-normalized absolute objective function coefficients** $\{|c_i|/\sqrt{n_i}\}_{i=1}^n$: mean and stddev

**LP-Based Features (expensive):**

92–94. **Integer slack vector:** mean, max, $L_2$ norm

95. **Objective function value of LP solution**

**Right-hand Side Features (trivial):**

96-97. **Right-hand side for $\leq$ constraints**: mean and stddev

98-99. **Right-hand side for $=$ constraints**: mean and stddev

100-101. **Right-hand side for $\geq$ constraints**: mean and stddev

**Presolving Features (moderate):**

102-103. **CPU times**: presolving and relaxation CPU time

104-107. **Presolving result features:** # of constraints, variables, non-zero entries in the constraint matrix, and clique table inequalities after presolving.

**Probing Cut Usage Features (moderate):**

108-112. **Number of specific cuts**: clique cuts, Gomory fractional cuts, mixed integer rounding cuts, implied bound cuts, flow cuts

**Probing Result features (moderate):**

113-116. **Performance progress**: MIP gap achieved, # new incumbent found by primal heuristics, # of feasible solutions found, # of solutions or incumbents found

**Timing Features**

117–121. **CPU time required for feature computation:** one feature for each of 5 groups of features (see text for details)

**Figure 3.2:** MIP instance features; for the variable-constraint graph, linear constraint matrix, and objective function features, each feature is computed with respect to three subsets of variables: continuous, $C$, non-continuous, $NC$, and all, $V$.

and percentile ratios (q90/q10) of features, which are based on vectors of values.

We introduce two new sets of features. Firstly, our new *MIP probing features* 102–116 are based on 5-second runs of CPLEX with default settings. They are obtained via the CPLEX API and include 6 *presolving* features based on the output of CPLEX's presolving phase (102–107); 5 *probing cut usage* features describing the different cuts CPLEX used during probing (108–112); and 4 *probing result* features summarizing probing runs (113–116).

Secondly, our new *timing features* 117–121 capture the CPU time required for computing five different groups of features: variable-constraint graph, linear constraint matrix, and objective features for three subsets of variables ("continuous", "non-continuous", and "all", 26–91); LP-based features (92–95); and CPLEX probing features (102–116). The cost of computing the remaining features (1–25, 96–101) is small (linear in the number of variables or constraints).

### 3.2.3 MIP Benchmarks

Most of the MIP benchmarks were collected from other research groups (except REG and RCW). In order to test the robustness of our predictive models and MIP solvers, we also considered some heterogenous benchmarks by combining several homogeneous benchmarks together.

*BIGMIX* This benchmark is a highly heterogenous mix of 1 510 publicly available Mixed Integer Linear Programming (MILP) instances. The instances in this set have an average of 8 610 variables and 4 250 constraints. Some of the instances are very large with up to 550 539 variables and 550 339 constraints.

*CORLAT* This benchmark comprises 2 000 MILP instances based on real data used for the construction of a wildlife corridor for grizzly bears in the Northern Rockies region (the instances were described by Gomes et al. (2008) and made available to us by Bistra Dilkina). All instances have 466 variables with 486 constraints on average.

*RCW* This benchmark comprises 1 980 MILP-encoded instances from computational sustainability for modeling the spread of the endangered red-cockaded woodpecker, conditional on decisions about certain parcels of land to be protected. We generated 1 980 instances (20 random instances for each combination of 9 maps and 11 budgets), using the generator from Ahmadizadeh et al. (2010) with the same parameter setting, but with a smaller sample size of 5.

*REG* This benchmark comprises 2 000 MILP-encoded instances based on the winner determination problem in combinatorial auctions. We used the `regions` generator from the Combinatorial Auction Test Suite [126], with the number of bids selected uniformly at random between 750 and 1250, and a fixed bids/goods ratio of 3.91 (following [130]).

*CL∪REG* This set is a mixture of two homogeneous subsets, `CORLAT` and `REG`. We randomly selected 1 000 `CORLAT` and 1 000 `REG` instances.

*CL∪REG∪RCW* This benchmark set is the union of `CL∪REG` and 990 randomly selected `RCW` instances.

*ISAC(new)* This set is a subset of the MIP benchmark set used by Kadioglu et al. (2010); we could not use the entire set, since the authors informed us that they irretrievably lost their test set. There are 276 instances in total.

*MIX* This is a very heterogenous benchmark that combines the sets studied in Hutter et al. (2010). It includes all instances from `MASS` (100 instances), `MIK` (120 instances), `CLS` (100 instances), and a subset of `CL` (120 instances) and `REG200` (120 instances). (see, e.g., [97] for the description of each underlying set.) There are 560 instances in total.

*MIPLIBless* `MIPLIB` is one of the most widely used benchmark for studying and evaluating MIP solvers. `MIPLIBless` consists of all 44 instances that can be solved by `CPLEX` 12.1 default within 1 800 CPU seconds on our reference

machines with Intel Xeon 3.2GHz CPUs.

*NELAND* This benchmark set comprises 640 MIP instances from Northeast Land Management [141] and is divided into 32 subsets while each subset contains 20 instances.

## 3.3 Traveling Salesperson (TSP)

The traveling salesperson problem (TSP) is one of the most widely studied combinatorial optimization problems. Given an edge-weighted directed graph $G$ with vertices $V = \{v_1, ..., v_n\}$, the goal is to find a Hamiltonian cycle (tour) in $G$ with a minimal path weight. For simplicity, a TSP instance is often defined in such a manner that the underlying graph is complete with very large edge weights for edges between disconnected nodes. Hence, a Hamiltonian cycle in $G$ corresponds exactly to a cyclic permutation of the vertices in $V$. There are many different types of TSP instances depending on restrictions on their weight functions. The best studied type is the Euclidean TSP, where the edge weight function $w$ is a Euclidean distance metric.

Over the past five decades, much work on TSP has been a driving force for many important research areas, such as stochastic local search [4, 107], branch-and-cut methods [5], and Ant Colony Optimization algorithms [191].

### 3.3.1 TSP Solvers

Most state-of-the-art complete algorithms for TSP are based on branch-and-cut methods. In brief, the basic process of branch-and-cut methods is to formulate a TSP as an integer programming problem (IP), and repeatedly solve linear programming (LP) relaxations of it. First, a cutting-plane method is used to solve an LP relaxation of a TSP that allows variables to take arbitrary values between 0 and 1. If the optimal LP solution is also an IP solution, the algorithm terminates and reports the IP solution; otherwise, a new restriction (cut) is added to the LP relaxation that cuts off non-integer solutions but does not cut off any integer solution. The new LP relaxation (with additional restrictions) is solved to optimality again. This procedure is repeated until no good "cut" can be found. At this stage, the cur-

rent problem needs to be branched into two sub-problems: an edge is selected and forced to be part of all solutions for one sub-problem and not to be part of any solution for another sub-problem. For each sub-problem, the cutting-plane method is used again. Thus, branch-and-cut methods iterate between a cutting-plane step and a branching step until an integer solution is found. The state-of-the-art complete algorithm for TSP, `Concorde`, can solve very large TSP instances [35].

Much work on incomplete algorithms for solving TSP has been focused on tour-construction heuristics and iterative tour-improvement algorithms. Tour-construction heuristics include the Nearest Neighbor Heuristic, the Insert Heuristic, and the Greedy Heuristic [169]. As an example, the Nearest Neighbor Heuristic constructs a tour starting from a randomly-chosen vertex $u_1$ and then iteratively adds one unvisited vertex $u_{k+1}$ to the current partial tour $(u_1, \cdots, u_k)$ such that $(u_k, u_{k+1})$ has minimal weight. After all vertices have been visited, a complete tour is obtained by connecting the end vertex of the partial tour $u_n$ to the starting vertex $u_1$. In practice, the tours obtained by tour-construction heuristics are usually very good for median size TSP with a few thousand nodes (11-16% to the optimal solutions) [83].

Most successful tour-improvement algorithms are based on $k$-exchange iterative improvement methods. Two candidate solutions, $s$ and $s'$, are called direct $k$-exchange neighbors if and only if $s'$ can be obtained from $s$ by deleting $k$ edges and reconnecting the resulting $k$ tour fragments into a complete tour with $k$ edges (the new edges may be the same as the deleted ones). The common choices of $k$ are 2 or 3, and much larger $k$ are rarely used because of the complexity in coding and the high computational cost in each step ($O(n^k)$). Experimental results suggest that with larger $k$ (4 or 5), an algorithm's solution quality can only be improved slightly [183]. Some additional techniques are available for making $k$-exchange more efficient, such as using *fixed radius search*, *candidate lists*, and *don't look bits*. One of the most well-known tour-improvement algorithms for TSP is the Lin-Kernighan (`LK`) algorithm [136]. This rather complicated algorithm is the foundation of many state-of-the-art incomplete TSP algorithms such as Iterated Lin-Kernighan [107], Chained Lin-Kernighan [4], and Iterated Helsgaun [70]. Optimal or very close to optimal solutions can be obtained within hours by using high-performance tour-improvement algorithms for TSPs with tens of thousands of nodes [108].

**Problem Size Features:**

1. **Number of nodes**: denoted n

**Cost Matrix Features:**

2–4. **Cost statistics:** mean, variation coefficient, skew

**Minimum Spanning Tree Features:**

5–8. **Cost statistics:** sum, mean, variation coefficient, skew

9–11. **Node degree statistics:** mean, variation coefficient, skew

**Cluster Distance Features:**

12–14. **Cluster distance:** mean, variation coefficient, skew

**Local Search Probing Features:**

15–17. **Tour cost from construction heuristic:** mean, variation coefficient, skew

18–20. **Local minimum tour length :** mean, variation coefficient, skew

21–23. **Improvement per step:** mean, variation coefficient, skew

24–26. **Steps to local minimum**: mean, variation coefficient, skew

27–29. **Distance between local minima**: mean, variation coefficient, skew

30–32. **Probability of edges in local minima**: mean, variation coefficient, skew

**Branch and Cut Probing Features:**

33–35. **Improvement per cut:** mean, variation coefficient, skew

36. **Ratio of upper bound and lower bound**

37–43. **Solution after probing:** Percentage of integer values and non-integer values in the final solution after probing. For non-integer values, we compute statics across nodes: min,max, 25%,50%, 75% quantiles

**Ruggedness of Search Landscape:**

44. **Autocorrelation coefficient**

**Timing Features**

45–50. **CPU time required for feature computation:** one feature for each of 6 computational subtasks

**Node Distribution Features (after instance normalization)**

51. **Cost matrix standard deviation:** standard deviation of cost matrix after instance being normalized into the rectangle [(0,0), (400, 400)].

52–55. **Fraction of distinct distances**: precision to 1, 2, 3, 4 decimal places.

56–57. **Centroid:** the (x, y) coordinates of the instance centroid.

58. **Radius:** the mean distances from each node to the centroid.

59. **Area:** the rectangular area within which the nodes lie.

60–61. **nNNd**: the standard deviation and coefficient variation of the normalized nearest neighbour distance.

62–64. **Cluster:** #clusters / n , #outliers / n, variation of #nodels in clusters.

**Figure 3.3:** 9 groups of TSP features

### 3.3.2 TSP Features

Figure 3.3 summarizes 64 features for the travelling salesperson problem (TSP). Features 1–50 are new, while Features 51–64 were introduced by Smith-Miles et al. [185]. Features 51–64 capture the spatial distribution of nodes (features 51–61) and clustering of nodes (features 62–64); we used the authors' code (available at http://www.vanhemert.co.uk/files/TSP-feature-extract-20120212.tar.gz) to compute these features.

Our 50 new TSP features are as follows.[2] The *problem size* feature (1) is the number of nodes in the given TSP. The *cost matrix* features (2–4) are statistics of the cost between two nodes. Our *minimum spanning tree* features (5–11) are based on constructing a minimum spanning tree over all nodes in the TSP: features 5–8 are the statistics of the edge costs in the tree and features 9–11 are based on its node degrees. Our *cluster distance* features (12–14) are based on the cluster distance between every pair of nodes, which is the minimum bottleneck cost of any path between them; here, the bottleneck cost of a path is defined as the largest cost along the path. Our *local search probing* features (15–32) are based on 20 short runs (1000 steps each) of LK [136], using the implementation available from [37]. Specifically, features 15–17 are based on the tour length obtained by LK; features 18–20, 21–23, and 24–26 are based on the tour length of local minima, the tour quality improvement per search step, and the number of search steps to reach a local minimum, respectively; features 27–29 measure the Hamming distance between two local minima; and features 30–32 describe the probability of edges appearing in any local minimum encountered during probing. Our *branch and cut probing* features (33–43) are based on 2-second runs of `Concorde`. Specifically, features 33–35 measure improvement in the lower bound per cut; feature 36 is the ratio of upper and lower bound at the end of the probing run; and features 37–43 analyze the final LP solution. Feature 44 is the autocorrelation coefficient: a measure of the ruggedness of the search landscape, based on an uninformed random walk (see, e.g., , [83]). Finally, our *timing features* 45–50 measure the CPU time required for computing feature groups 2–7 (the cost of computing the number of nodes can be ignored).

### 3.3.3 TSP Benchmarks

We used three TSP benchmarks that come from random TSP generators and TSPlib. The detailed information about these benchmarks is as follows.

[2] In independent work, Mersmann et al. [144] have introduced feature sets similar to some of those described here.

*PORTGEN* This benchmark comprises 4 993 uniform random EUC-2D TSP instances generated by the random TSP generator portgen [106]. The number of nodes was randomly selected from 100 to 1 600 and the generated TSP instances have $849 \pm 429$ nodes.

*PORTCGEN* This benchmark comprises 5 001 random clustered EUC-2D TSP instances generated by the random TSP generator, portcgen [106]. The number of nodes was randomly selected from 100 to 1 600 and the number of clusters was set to 1% of the number of nodes. The generated TSP instances have $852 \pm 432$ nodes.

*TSPLIB* This benchmark contains a subset of the prominent TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/) repository. We only included the 63 instances for which both our own feature computation code and the code by Smith-Miles and van Hemert (2011) completed successfully within 3,600 CPU seconds on our reference machines.

# Chapter 4

# Solution Prediction for SAT

Recent work has studied the use of regression methods to make instance-specific predictions on solver runtimes. Nudelman et al. (2004) showed that using this approach, surprisingly accurate runtime predictions can be obtained for uniform random 3-SAT. They also noticed that training models on only SAT or UNSAT instances allowed much simpler, but very different, models to achieve high accuracies. (In Chapter 5, we demonstrate that such simpler models can be used for constructing hierarchical hardness models for better runtime prediction.) Since unconditional models are able to predict runtimes accurately, despite the qualitative differences between the SAT and UNSAT regimes, we believe that the models must implicitly predict satisfiability status. This chapter tests this hypothesis on one of most difficult SAT benchmarks, the uniform random 3-SAT at the phase transition, and shows how to build classification models that achieve accuracies of approximately 70% with a small set of features. Two arguments demonstrate that this is not a small-size effect. First, the models' predictive accuracy remains roughly constant, and is far better than that of random guessing (50%) across the entire range of problem sizes. Second, a classifier trained on our easiest ($v = 100$) instances again achieves very accurate predictions across the whole range of instance sizes. A detailed investigation shows that two features sufficed to achieve good performance for all instance sizes: one based on variation in the slack vectors of an LP relaxation of the problem, and one based on the ratio of positive to negative literals in the formula. Finally, we trained a three-leaf decision tree based on these

two features only on the smallest instances, which achieved prediction accuracies across the entire range of instance sizes close to those of our most complex model. [1]

## 4.1 Uniform Random 3-SAT and Phase Transition

A prominent family of SAT instances is uniform random 3-SAT. Instances from this class are easy to generate and often hard to solve, they have often been used as a test bed in the design and evaluation of heuristic algorithms (see, e.g., Le Berre et al., 2012). One interesting phenomena related to uniform random 3-SAT is the so-called solubility phase transition: the probability that a random 3-SAT instance is satisfiable exhibits sharp threshold behavior when the control parameter $\alpha = c/v$ passes a critical value [32, 146]. The width of the window in which this solubility phase transition takes place becomes narrower as instance size grows.

Most interestingly, a wide range of state-of-the-art SAT solvers exhibit dramatically longer runtimes for instances in this critical region. For intuition, note that instances are under-constrained when $\alpha$ is small (where few constraints exist, and therefore many solutions), and over-constrained when $\alpha$ is large (where many constraints exist, making it relatively easy to derive a contradiction). The so-called phase transition point occurs between these extremes, when the probability of generating a satisfiable instance is 0.5. Crawford and Auton (1996) confirmed these findings in an extensive empirical study and proposed a more accurate formula for identifying the phase transition point. Kirkpatrick and Selman (1994) used finite-size scaling, a method from statistical physics, to characterize size-dependent effects near the transition point, with the width of this transition narrowing as the number of variables increases. Yokoo (1997) studied the behavior of simple local search algorithms on uniform random 3-SAT instances, observing a peak in the hardness of solving satisfiable instances at the phase transition point. He attributed this hardness peak to a relatively larger number of local minima present in critically constrained instances, as compared to over-constrained satisfiable instances.

There is a useful analogy between uniform random 3-SAT problems and what physicists call "disordered materials": conflicting interactions in the latter are sim-

[1] This chapter is based on the joint work with Holger Hoos, and Kevin Leyton-Brown [214].

ilar to the randomly negated variables in the former. Exploiting this connection, uniform random 3-SAT has been studied using methods from statistical physics. Monasson and Zecchina [148, 149] applied replica methods to determine the characteristics of uniform random 3-SAT and showed that at the phase transition, the ground state entropy is finite. They concluded that the transition itself is due to the abrupt appearance of logical contradictions in all solutions and not to a progressive decrease in the number of models.

## 4.2 Experimental Setup

We considered uniform random 3-SAT instances generated at the solubility phase transition with $v$ ranging from 100 to 600 variables in steps of 25. For each value of $v$, we generated 1000 instances with different random seeds using the same instance generator as used in SAT competitions since 2002 [182]. In total, we generated 21 instance sets jointly comprising 21 000 3-SAT instances. For $v = 100$, 25 000 additional instances are generated (referred as $v100(large)$).

We solved all of these instances using `kcnfs07` [44] within 36 000 CPU seconds per instance, with the exception of 2 instances for $v = 575$ and 117 for $v = 600$. For these instances, additional 5 runs of `adaptg2wsat09++` [134] were performed with a cutoff time of 36 000 CPU seconds, which failed to solve them. As this cutoff is more than 100 times larger than the runtime of the hardest instances solved by `adaptg2wsat09++`, and the largest increase in running time needed for any size $v > 475$ to solve an additional instance we ever observed was lower than a factor of 6.5, we believe that these instances are unsatisfiable, and so treated them as such for the remainder of our study. (Readers who feel uncomfortable with this approach should feel free to disregard our results for $v = 575$ and $v = 600$; none of our qualitative conclusions is affected.)

Figure 4.1 (left) shows the median runtime of `kcnfs07` on both satisfiable and unsatisfiable instances across our 21 instance sets. Median `kcnfs07` runtime increased exponentially with the number of variables, growing by a factor of about 2.3 with every increase of 25 variables beyond $v = 200$ (smaller instances were solved more quickly than the smallest CPU time one could measure). To verify that the generating instances are indeed at the phase transition point, we examined

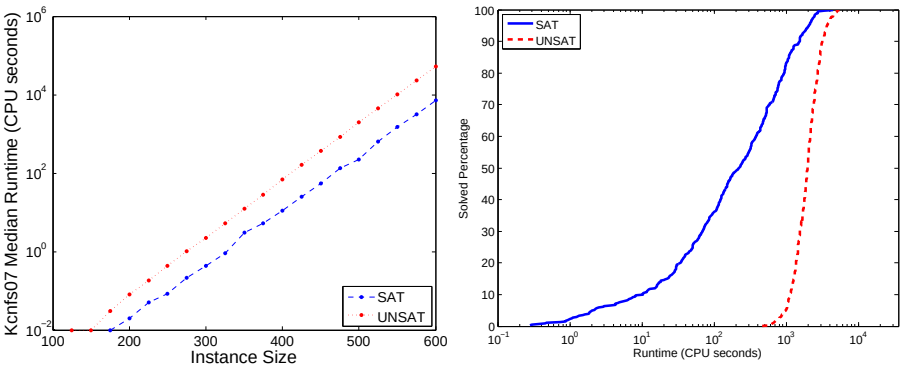



**Figure 4.1:** *Left: Median runtime of* `kcnfs07` *for each instance set. The solutions of some instances in* $v = 575$ *and* $v = 600$ *were estimated by running* `adaptg2wsat09++` *for 36 000 CPU seconds. Right: CDF of* `kcnfs07`*'s runtime for* $v = 500$.

the fraction of satisfiable and unsatisfiable instances for each set; the majority contained between 49 and 51% satisfiable instances (with mean 50.2% and standard deviation 1.6%), and there was no indication that deviations from the 50% mark correlated with instance size. The $v100(large)$ set contained 49.5% satisfiable instances. As illustrated in Figure 4.1 (right) for $v = 500$, unsatisfiable instances tended to be harder to solve, and to give rise to less runtime variation; satisfiable instances were easier on average but with larger runtime variation. Intuitively, to prove unsatisfiability, a complete solver such as `kncfs07` needs to reason about the entire space of candidate assignments, while satisfiability may be proven by exhibiting a single model. Depending on the number of solutions of a given instance, which is known to vary at the phase transition point, the search cost of finding the first one can vary significantly.

This work used a state-of-the-art classifier, decision forests, as they can produce good predictions with robust uncertainty estimates and direct visualization from tree structures. Decision forests are constructed as collections of $T$ decision trees [196]; in this work, a rather large number of trees ($T = 99$) is used for high classification accuracy. Following Breiman (2001), given $n$ training data points with $k$ features each, for each tree we drew a bootstrap sample of $n$ training data points sampled uniformly at random with repetitions; during tree construction, we sampled a random subset of $\log_2(k)+1$ features at each internal node to be consid-

ered for splitting the data at that node. Predictions were based on majority voting across all $T$ trees. In our case, the class labels were SAT and UNSAT. Since we used 99 trees, an instance $i$ was classified as SAT if more than 49 trees predicted that it was satisfiable. We measured the decision forest's confidence as the fraction of trees that predicted $i$ to be satisfiable; by choosing $T$ as an odd number, we avoided the possibility of ties.

We used the feature set as listed in Figure 3.1. The feature computation time depends on the size of the instance under consideration (e.g., $\approx 50$ CPU seconds on average for all features of a single instance with $v = 550$, of which $\approx 41$ CPU seconds were spent on computing the 6 LP-based features). Some easy instances are solved during the computation of certain features (in particular, local search probing features). Thus, even though these features have been found quite useful for other problems/taskes [156], we excluded them from this study and resolved to use 61 features, of which 7 are related to problem size, 29 to graph-based representations of the CNF formula, 13 to balance properties, 6 to proximity to a Horn formula, and 6 to LP relaxations.

For each instance size $v$, we first partitioned the respective instance set into two subsets based on satisfiability status, and then randomly split each subset 60:40 into training and test sets. Finally, we combined the training sets for SAT and UNSAT into the final training set, and the SAT and UNSAT test sets into the final test set. We trained our decision forests on the training sets only, and used only the test sets to measure model accuracy. In order to reduce variance in these accuracy measurements we repeated this whole process 25 times (with different random training/test splits); the results reported in this paper are medians across these 25 runs.

All runtime and feature data were collected on a computer cluster with 840 nodes, each equipped with two 3.06 GHz Intel Xeon 32-bit processors and 2GB of RAM per processor. The decision forest classifier was implemented in Matlab, version R2010a, which we also used for data analysis.

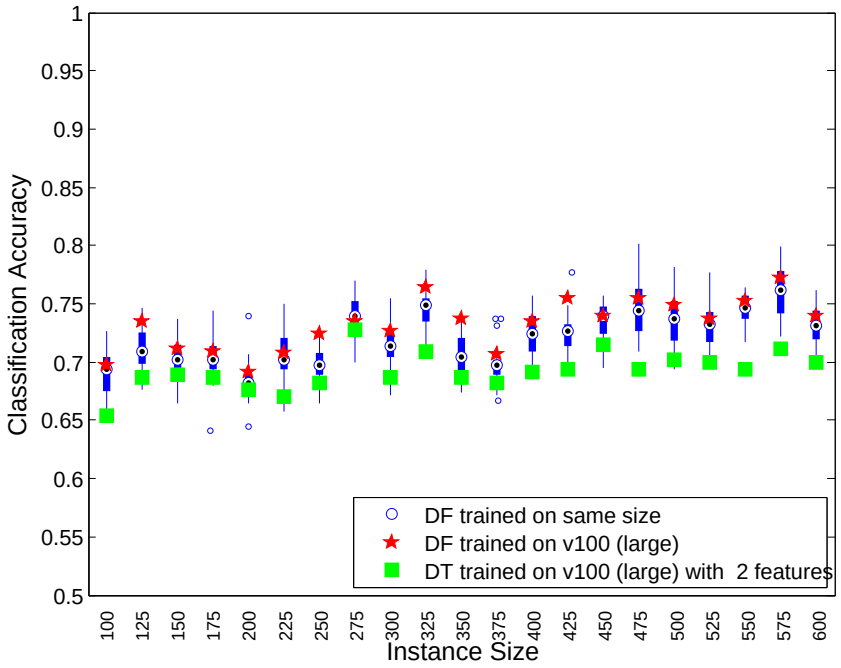



**Figure 4.2:** *Classification accuracies achieved on our 21 primary instance sets. The blue box plots are based on 25 replicates of decision forest models, trained and evaluated on different random splits of training and test data. The median prediction accuracies of using the decision forest trained on $v100(large)$ are shown as red stars. The median prediction accuracies of using a single decision tree trained on $v100(large)$ based on two features are shown as green squares.*

## 4.3 Experimental Results

At the solubility phase transition, uniform random 3-SAT instances are equally likely to be satisfiable or unsatisfiable. Thus, random guessing can achieve predictive accuracy of only 50%. The first goal was to investigate the extent to which our models were able to make more accurate predictions. As shown in Figure 4.2 (first data series) and Table 4.1, the models did achieve accuracies of between about 70% and 75%. There is no significant difference in the frequency of the two possible predictive errors (predicting SAT as UNSAT and vice versa).

Figure 4.3 shows two sample distributions ($v = 200$ and $v = 500$) of the classifier's confidence. The plots for other instance sets were qualitatively similar. Recall that the confidence of the classifier is measured by the fraction of 'SAT' predictions among the 99 trees. Therefore, the classifier had full confidence if all 99 predic-

| Variables | Median Accuracy | False Positives | False Negatives |
|---|---|---|---|
| 100 | 0.694 | 0.138 | 0.168 |
| 125 | 0.709 | 0.125 | 0.166 |
| 150 | 0.702 | 0.148 | 0.150 |
| 175 | 0.702 | 0.155 | 0.144 |
| 200 | 0.682 | 0.153 | 0.164 |
| 225 | 0.703 | 0.148 | 0.153 |
| 250 | 0.697 | 0.158 | 0.148 |
| 275 | 0.740 | 0.140 | 0.120 |
| 300 | 0.714 | 0.143 | 0.143 |
| 325 | 0.749 | 0.122 | 0.130 |
| 350 | 0.704 | 0.151 | 0.143 |
| 375 | 0.697 | 0.148 | 0.155 |
| 400 | 0.724 | 0.143 | 0.135 |
| 425 | 0.727 | 0.138 | 0.135 |
| 450 | 0.740 | 0.128 | 0.132 |
| 475 | 0.744 | 0.118 | 0.138 |
| 500 | 0.737 | 0.130 | 0.133 |
| 525 | 0.733 | 0.143 | 0.125 |
| 550 | 0.747 | 0.120 | 0.133 |
| 575 | 0.762 | 0.113 | 0.125 |
| 600 | 0.732 | 0.129 | 0.139 |

**Table 4.1:** *The performance of decision forests with 61 features on our 21 primary instance sets. We report median classification accuracy over 25 replicates with different random splits of training and test data as well as the fraction of false positive and false negative predictions.*

tions were consistent, and had small confidence if the numbers of 'SAT' predictions and 'UNSAT' predictions were about the same. As shown in Figure 4.3, the classifier had low levels of confidence more often than high levels of confidence. The representative examples in the left and right panes of Figure 4.3 illustrate that our decision forests had more difficulty with small instances, in the sense that they were uncertain about a larger fraction of such instances.

As one would hope, the confidence was positively correlated with classification accuracy. This can be seen by comparing the height of the bars for correct and incorrect predictions at each predicted probability of SAT. When the predicted probability of SAT was close to 0 or 1, the classifier was almost always correct, and

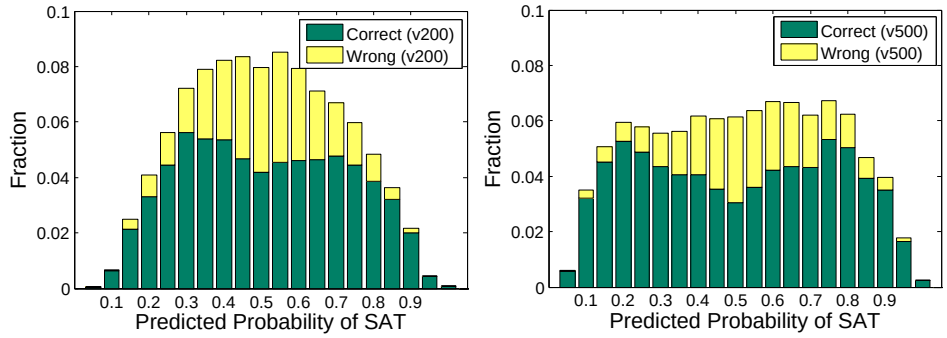



**Figure 4.3:** *Classifier confidence* vs *fraction of instances. Left: $v = 200$; Right: $v = 500$.*

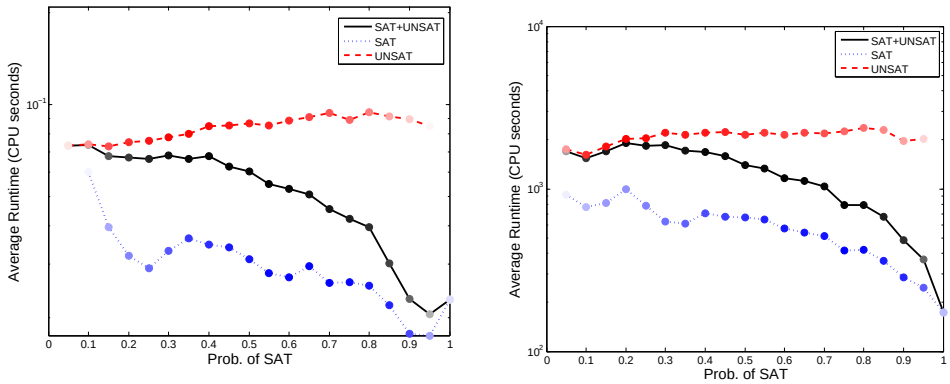



**Figure 4.4:** *Classifier confidence* vs *instance hardness. Each marker ($[x,y]$) shows the average runtime of* `kcnf07` *over a bin of instances with classifier confidence (predicted probability of SAT) between $x-0.05$ and $x$. Each marker's intensity corresponds to the amount of data inside the bin. Left: $v = 200$; Right: $v = 500$.*

when the predicted probability of SAT was close to 0.5, accuracy dropped towards 0.5 (i.e., random guessing).

The decision forest's confidence was also correlated with `kcnfs07`'s runtime. As shown in Figure 4.4, instances tended to be easier to solve when the predicted probabilities of SAT were close to either 0 or 1. Recall that variation in runtime was more pronounced on satisfiable instances, as previously illustrated in Figure 4.1 (right).

We now examine the hypothesis that our models' prediction accuracy decreases as problem size grows. Two pieces of evidence suggest that this hypothesis should be rejected. The first evidence is the result of a pairwise comparison of the clas-

sification accuracies obtained from the full decision forest models trained for each instance size. For each pair of data sets with instance sizes $i$ and $j$ ($i > j$), Figure 4.5 shows a blue dot when classification accuracy on size $i$ was significantly higher than on size $j$ and a yellow dot when classification accuracy on size $i$ was significantly lower than on size $j$, according to a Mann-Whitney U test. Among the 210 paired comparisons with significance level 0.05, there are 133 blue dots (63.3%), and 21 yellow dots (10.0%). Thus, there is little evidence that prediction accuracy decreases as instance size grows; indeed, the data shown in Figure 4.5 appears to be more consistent with the hypothesis that prediction accuracy *increases* with instance size.

The second piece of evidence against the hypothesis of lower accuracies for bigger problems is that models trained only on the smallest problems achieved high levels of predictive accuracy across the whole range of problem sizes. The red stars in Figure 4.2 (the second data series) indicate the performance of the decision forest trained on $v100(large)$ evaluated on problems of other sizes. This single model performed about as well—indeed, in many cases better—than the models specialized to different problem sizes. Although we do not report the results here, we also trained decision forests on each of the other 20 instance sets; in each case, the models generalized across the entire range of problem sizes in qualitatively the same manner.

The next task is to identify the smallest set of features that could be used to build accurate models. Hopefully, such feature set will be useful to other researchers seeking a theoretical explanation of the phenomenon identified in this chapter.

One may imagine that most predictive features are different for large instances and for small instances. Therefore, the 21 instance sets are divided into two groups, *small* (10 instance sets, $v = 100$ to 325) and *large* (10 instance sets, $v = 375$ to 600). We do not use the 350-variable set in this analysis in order to keep the two groups balanced. For every subset of the 61 features with cardinality 1, 2 and 3, we measure the accuracy of the model. For both *small* and *large*, the best 1- and 2-feature sets were subsets of the best 3-feature subset. Next, we use a forward selection procedure to build subsets of up to 10 features, as exhaustive enumeration was infeasible above 3 features. Specifically, starting with the best 3-feature set,

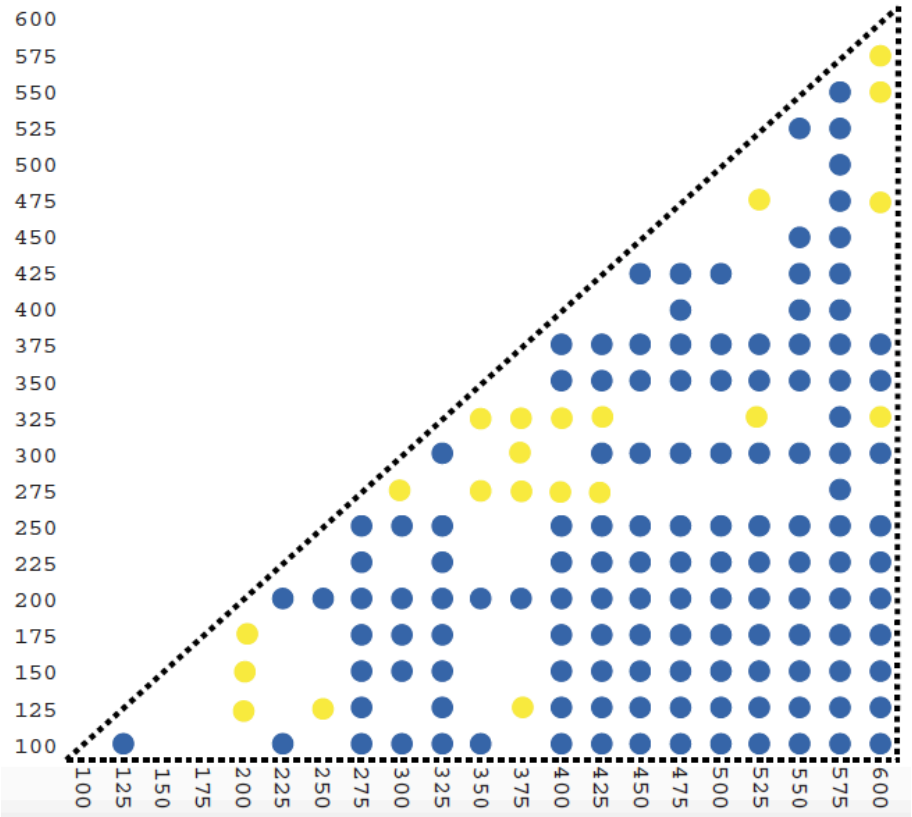



**Figure 4.5:** *Statistical significance of pairwise differences in classification accuracy for our 21 primary instance sets. Yellow: accuracy on the smaller instance size is significantly higher than on the larger size. **Blue**: accuracy on the smaller instance size is significantly lower than on the larger size. No dot: the difference is insignificant. Significance level: $p = 0.05$.*

and for every feature not in the set, we compute the mean of median classification accuracy across all of the instance sets in the group. Then, we add the feature with the best such accuracy to the feature subset. These steps are repeated until the subset contains 10 features.

Table 4.2 describes the sets of obtained features, as well as the improvement in classification accuracy achieved at each step. In both cases of *small* and *large*, the classifier was able to achieve good predictive accuracy with a small number of features; for each number of features, the classification accuracy on large instances was better than on small instances. The two most informative features were identi-

| Group | Features ordered by FW | Classification Accuracy | Stepwise Improvement |
|---|---|---|---|
| small | **LPSLACK_coeff_variation** | **0.614** | – |
| | **POSNEG_ratio_var_mean** | **0.670** | **0.056** |
| | LP_OBJ | 0.681 | 0.011 |
| | VG_mean | 0.688 | 0.007 |
| | LPSLACK_max | 0.690 | 0.002 |
| | VG_max | 0.692 | 0.002 |
| | VCG_var_max | 0.694 | 0.002 |
| | HORNY_var_coeff_variation | 0.694 | 0.000 |
| | LPSLACK_mean | 0.695 | 0.001 |
| | LP_int_ratio | 0.697 | 0.002 |
| large | **LPSLACK_coeff_variation** | **0.646** | – |
| | **POSNEG_ratio_var_mean** | **0.696** | **0.050** |
| | LPSLACK_mean | 0.706 | 0.010 |
| | LP_int_ratio | 0.714 | 0.008 |
| | VCG_clause_max | 0.720 | 0.006 |
| | CG_mean | 0.721 | 0.001 |
| | TRINARYp | 0.725 | 0.004 |
| | HORNY_var_coeff_variation | 0.727 | 0.002 |
| | DIAMETER_entropy | 0.728 | 0.001 |
| | POSNEG_ratio_clause_entropy | 0.728 | 0.000 |

**Table 4.2:** *The mean of median classification accuracy with up to 10 features selected by forward selection. The stepwise improvement for a feature $f_i$ at forward selection step $k$ is the improvement when we add $f_i$ to the existing $k-1$ features. Each median classification accuracy is based on the results of 25 runs of classification with different random split of training and test data.*

cal for both groups, and adding additional features beyond this point offered little marginal benefit.

It is worth understanding the meaning of these features. LPSLACK_coeff_variation is based on solving a linear programming relaxation of an integer program representation of SAT instances. For each variable $i$ with LP solution $S_i$, $\texttt{LPSLACK}_i$ is defined as $\min\{1-S_i, S_i\}$: $S_i$'s proximity to integrality. LPSLACK_coeff_variation is then the coefficient of variation of the vector LPSLACK. POSNEG_ratio_var_mean is the average ratio of positive and negative occurrences of each variable. For each variable $i$ with $P_i$ positive occurrences and $N_i$ negative occurrences, $\texttt{POSNEG\_ratio\_var}_i$ is defined as $2 \cdot |0.5 - P_i/(P_i+N_i)|$. POSNEG_ratio_var_mean is then the average over elements of the vector POSNEG_ratio_var.

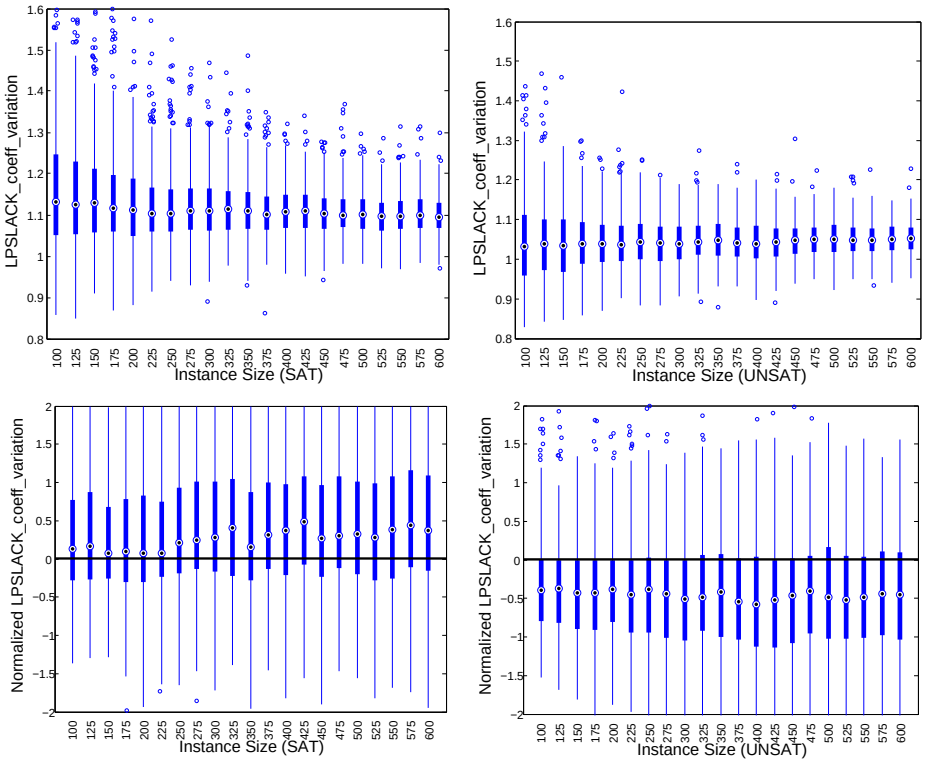



**Figure 4.6:** *Distribution of* `LPSLACK_coeff_variation` *over instances in each of our 21 sets. Left: SAT; Right: UNSAT. Top: original value; Bottom: value after normalization. The line at* $y = 0.0047$ *indicates the decision threshold used in the tree from Figure 4.8.*

There are three main findings: (1) models achieved high accuracies; (2) models trained on small instances were effective for large instances; (3) a model consisting of only two features was nearly as accurate as the full model. We now show that all of these findings also held simultaneously where we were able to achieve high accuracies using a two-feature model trained only on small instances. Specifically, we construct a single decision tree (rather than a random forest) using only the `LPSLACK_coeff_variation` and `POSNEG_ratio_var_mean` features, and trained using only the very easiest instances, $v100(large)$. We further simplify this model by setting the parameter `minparent` of the tree building procedure to 10 000. The `minparent` parameter defines the smallest number of observations that impure

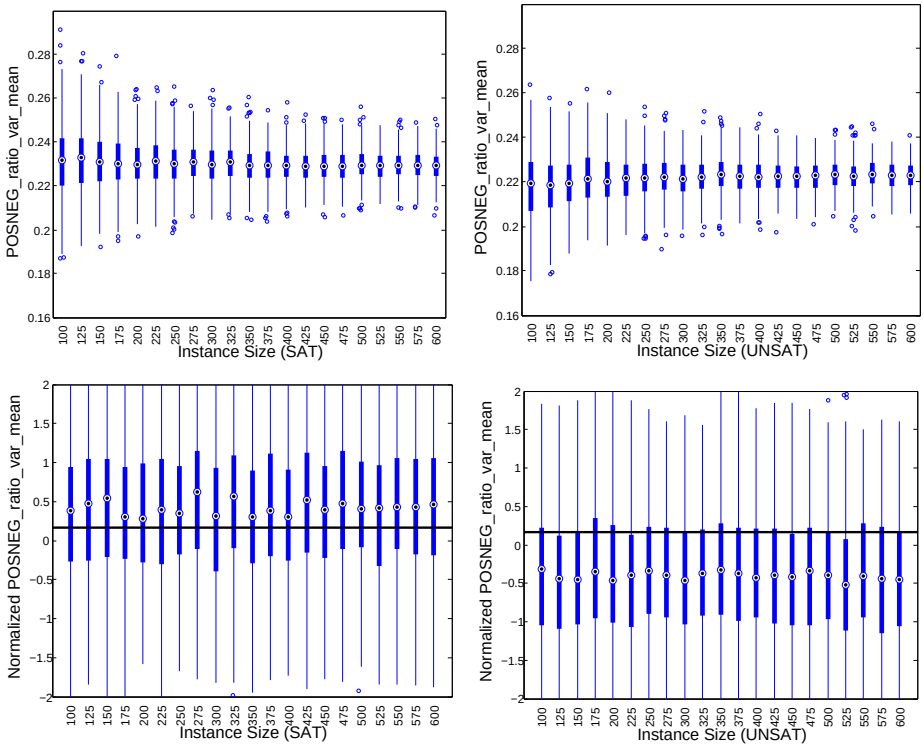



**Figure 4.7:** *Distribution of* `POSNEG_ratio_var_mean` *over instances in each of our 21 instance sets. Left: SAT; Right: UNSAT. Top: original value; Bottom: value after normalization. The line at* $y = 0.1650$ *indicates the decision threshold used in the tree from Figure 4.8.*

nodes may contain before they are allowed to further split; setting it to such a large value forced the decision tree to be extremely simple. This tree's performance is plotted using green squares (third data series) in Figure 4.2. Overall, it achieved remarkably good prediction accuracies of always more than 65% on all instance sets.

Figure 4.8 shows the decision tree. First, it classifies instances as satisfiable if `LPSLACK_coeff_variation` takes a large value: that is, if `LPSLACK` exhibits large variance across the variables in the given formulae (region `A`). When `LPSLACK_coeff_variation` takes a small value, the model considers the balance between positive and negative literals in the formula (`POSNEG_ratio_var_mean`). If the literals' signs are relatively balanced, the model predicts unsatisfiability (re-

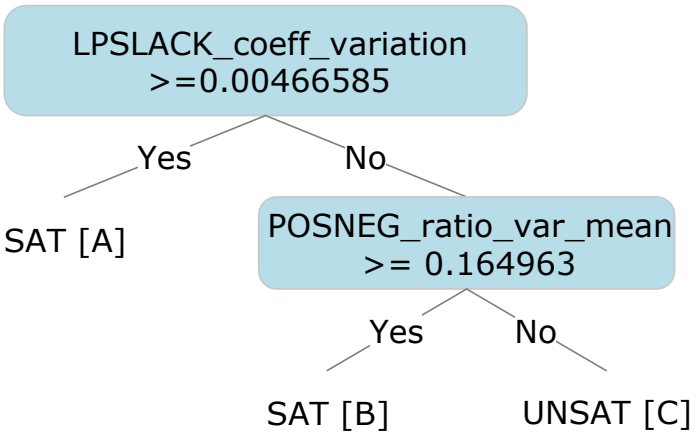



**Figure 4.8:** *The decision tree trained on* $v100(large)$ *with only the features* `LPSLACK_coeff_variation` *and* `POSNEG_ratio_var_mean`*, and with* (`minparent`) *set to 10 000.*

gion `B`). Otherwise, it predicts satisfiability (region `C`). To gain further understanding about the effectiveness of this model, we partitioned each of our 21 data sets into the three regions and observed the fraction of each partition that was correctly labeled by the tree. These fractions were between 60 and 70% (region `A`), between 70 and 80% (region `C`), and about 50% (region `B`).

Finally, Figures 4.6 and 4.7 show the distribution of the `LPSLACK_coeff_variation` and `POSNEG_ratio_var_mean` features over each of our 21 instance sets, before and after normalization, and considering satisfiable and unsatisfiable instances separately. Note that both features' pre-normalization variations decreased with instance size, while their median values remained relatively constant. After normalization, both features' distributions remained very similar as instance size increased. The decision thresholds used by our simple decision tree are plotted as solid horizontal lines in these figures. Both thresholds were located near the 25th or 75th percentiles of the respective distributions, regardless of instance size.

## 4.4 Conclusion

Uniform random 3-SAT instances from the solubility phase transition are challenging to solve considering their size. Nevertheless, this work has shown that the sat-

isfiability of such instances can be predicted efficiently and with surprisingly high accuracy. The experimental results demonstrated that high prediction accuracies (19.4% – 26.2% better than random guessing) can be achieved across a wide range of instance sizes, and that there is little support for the hypothesis that this accuracy decreases as instance sizes grow. The predictive confidence of our classifiers correlates with the prediction accuracy obtained and with the runtime of a state-of-the-art complete SAT solver. A classifier trained on small, very easy instances also performed well on large, extremely challenging instances. Furthermore, the features most important to models trained on different problem sizes were substantially the same. Finally, using only two features, `LPSLACK_coeff_variation` and `POSNEG_ratio_var_mean`, we could build a trivial, three-leaf decision tree that achieved classification accuracies only slightly below those for the most complex decision forest classifier. Examining the operation of this model, we observe the surprisingly simple rules that instances with large variation in `LPSLACK` (distance of LP solutions to integer values) across variables are likely to be satisfiable, while instances with small variation of `LPSLACK` and roughly the same number of positive and negative literals are likely to be unsatisfiable. We hope that these rules will lead to novel heuristics for SAT solvers targeting random instances, and will serve as a starting point for new theoretical analysis of uniform random 3-SAT at the phase transition.

# Chapter 5

# Runtime Prediction with Hierarchical Hardness Models

The previous chapter shows that the satisfiability of SAT instances can be predicted with high accuracy for uniform random 3-SAT at the phase transition. This chapter demonstrates that such satisfiability prediction can be used for constructing hierarchical hardness models for better runtime prediction. First, we confirm the observation of Nudelman et al. (2004) that better models can be learned with only satisfiable instances (SAT) or only unsatisfiable instances (UNSAT). Then, we show that predicting satisfiability is possible in general by considering a variety of both structured and unstructured SAT instances. More importantly, satisfiability prediction is useful for combining SAT models and UNSAT models into hierarchical hardness models using a mixture-of-experts approach. Such hierarchical models improve overall runtime prediction accuracy. Classification confidence correlates with runtime prediction accuracy, giving useful per-instance evidence about the quality of the runtime prediction. [1]

## 5.1 Empirical Hardness Models

For a given problem instance, empirical hardness models predict the runtime of an algorithm based on polytime-computable instance features using regression tech-

[1] This chapter is based on the joint work with Holger Hoos, and Kevin Leyton-Brown [208].

niques. One of the most popular methods is linear basis-function ridge regression, which has previously been demonstrated to be very successful in studies of uniform random SAT and combinational auctions [127, 156].

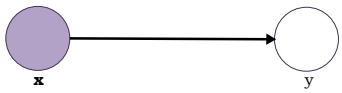
### 5.1.1 Overview of Linear Basis-function Ridge Regression

In order to predict the runtime of an algorithm A on a distribution D of problem instances, one first collects the runtimes of algorithm A on $n$ instances drawn from D, $\boldsymbol{y}$, and computes $p$-dimensional feature vectors, $\boldsymbol{X}$. Let $\boldsymbol{I_p}$ be the $p \times p$ identity matrix, and let $\varepsilon$ be a small constant. Then, compute the weight vector

$$\boldsymbol{w} = (\boldsymbol{X}^\intercal \boldsymbol{X} + \varepsilon \boldsymbol{I_p})^{-1} \boldsymbol{X}^\top \boldsymbol{y},$$

where $\boldsymbol{X}^\intercal$ denotes the transpose of matrix $\boldsymbol{X}$. The effect of $\varepsilon > 0$ is to regularize the model by penalizing large coefficients $\boldsymbol{w}$ and to improve numerical stability. For the latter reason, we also use forward selection to eliminate highly correlated features. Since algorithm runtime can often be better approximated by a polynomial function than by a linear one, a quadratic basis function expansion was used by Leyton-Brown et al. (2002), augmenting each model input $\boldsymbol{X}_i = [x_{i,1}, \ldots, x_{i,p}]^\intercal$ with pairwise product inputs $x_{i,j} \cdot x_{i,q}$ for $j = 1, \ldots, p$ and $q = j, \ldots, p$. Then, another pass of forward feature selection is performed for selecting a subset of extended features $\boldsymbol{\Phi}$. The final empirical hardness model is learned with $\boldsymbol{\Phi}$ and $\boldsymbol{y}$. Given a new unseen instance with extended feature vector $\boldsymbol{\Phi}_{n+1}$, ridge regression predicts $f_{\boldsymbol{w}}(\boldsymbol{\Phi}_{n+1}) = \boldsymbol{w}^\intercal \boldsymbol{\Phi}_{n+1}$.

Empirical hardness models have a probabilistic interpretation. The features $\boldsymbol{\Phi}$ and the empirical algorithm runtime $y$, when seen as random variables, are related as in the following graphical model:




In this model, the feature vector $\boldsymbol{\Phi}$ (or $\boldsymbol{X}$), and the probability distribution over runtime $y$ is conditionally dependent on $\boldsymbol{\Phi}$. Since one trains a linear model using least squares fitting, we have implicitly chosen to represent $P(y|\boldsymbol{\Phi})$ as a Gaussian with mean $\boldsymbol{w}^\top \boldsymbol{\Phi}$ and some fixed variance $\beta$. The prediction of an empirical hardness

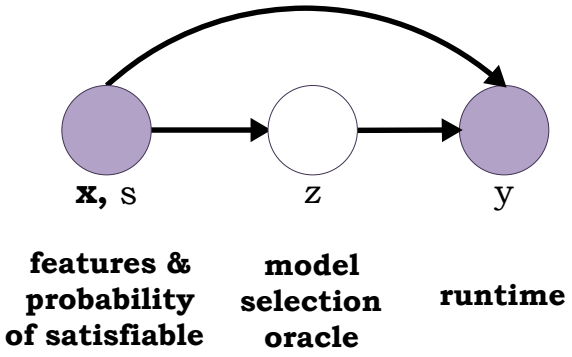



**Figure 5.1:** *Graphical model for our mixture-of-experts approach.*

model is actually $\mathbb{E}(y|\mathbf{\Phi})$, the mean of this distribution conditioned on the observed feature vector.

### 5.1.2 Hierarchical Hardness Models

Previous work [156] has shown that if instances are restricted to be either only satisfiable or only unsatisfiable, very different models are needed to make accurate runtime predictions. Furthermore, models for each type of instance are simpler and more accurate than models that must handle both types, which means that better empirical hardness models can be built if one knows the satisfiability of instances. With accurate satisfiability prediction (shown in the previous chapter), it would be tempting to construct a hierarchical model that uses a classifier to pick the most likely conditional model and then simply return that model's prediction. However, while this approach could sometimes be a good heuristic, it is not theoretically sound. Intuitively, the problem is that the classifier does not take into account the accuracies of the different conditional models. The model trained on one class of instances could have very large (indeed, unbounded) prediction error on instances from another class.

A more principled method of combining conditional models can be derived based on the probabilistic interpretation of empirical hardness models given in Section 5.1.1. As shown previously (see Figure 5.1), there is a set of features that are always observed and a random variable representing runtime that is conditionally dependent on the features. Now the features and our classifier's prediction $s$ are combined into a new feature vector $(\mathbf{\Phi}, s)$. A new random variable, $z \in \{sat, unsat\}$, is introduced to represent the oracle's choice of which conditional model will perform best for a given instance. Instead of selecting one of the predictions from the two conditional models for runtime $y$, we use their weighted sum.

$$P(y|(\mathbf{\Phi}, s)) = \sum_{z \in \{sat, unsat\}} P(z|(\mathbf{\Phi}, s)) \cdot P_{M_z}(y|(\mathbf{\Phi}, s)), \tag{5.1}$$

where $P_{M_z}(y|(\mathbf{\Phi}, s))$ is the probability of $y$ evaluated according to model $M_z$. Since the models are fit using ridge regression, Eq. (5.1) can be written as

$$P(y|(\mathbf{\Phi}, s)) = \sum_{z \in \{sat, unsat\}} P(z|(\mathbf{\Phi}, s)) \cdot \mathcal{N}(y|\boldsymbol{w}_z \mathbf{\Phi}, \beta_z), \tag{5.2}$$

where $\boldsymbol{w}_z$ and $\beta_z$ are the weights and the standard deviation of model $M_z$ respectively. Thus, the weighting functions $P(z|(\mathbf{\Phi}, s))$ are learned to maximize the likelihood of the training data according to $P(y|(\mathbf{\Phi}, s))$. As a hypothesis space for these weighting functions we chose the commonly used softmax function

$$P(z = sat|(\mathbf{\Phi}, s)) = \frac{e^{\boldsymbol{v}^\top (\mathbf{\Phi}, s)}}{1 + e^{\boldsymbol{v}^\top (\mathbf{\Phi}, s)}}, \tag{5.3}$$

where $\boldsymbol{v}$ is a vector of free parameters that must be learned [24]. Therefore, the loss function is as follows, where $\mathbb{E}(y_{i,z}|\mathbf{\Phi_i})$ is the prediction of $M_z$ and $\hat{y}_i$ is the real runtime:

$$\mathscr{L} = \sum_{i=1}^{N} \left( \hat{y}_i - \left( \sum_{k \in \{sat, unsat\}} P(z = k|(\mathbf{\Phi_i}, s_i)) \cdot \mathbb{E}(y_{i,z}|\mathbf{\Phi_i}) \right) \right)^2. \tag{5.4}$$

This can be seen as a mixture-of-experts problem with the experts clamped to $M_{sat}$ and $M_{unsat}$ (see, e.g., [24]). For implementation convenience, we used an existing mixture of experts implementation, which is built around an EM algorithm and

performs iterative reweighed least squares in the M step [150]. This code was modified slightly to clamp the experts and to set the initial values of $P(z|(\mathbf{\Phi}, s))$ to $s$ (i.e., we initialized the choice of experts to the classifier's output). To evaluate the model and obtain a runtime prediction for test data, we simply compute the features $\boldsymbol{x}$ and the classifier's output $s$, and then evaluate

$$\mathbb{E}(y|(\mathbf{\Phi}, s)) = \sum_{k \in \{sat, unsat\}} P(z|(\mathbf{\Phi}, s)) \cdot \boldsymbol{w}_k^{\top} \cdot \Phi, \tag{5.5}$$

where $\boldsymbol{w}_k$ are the weights from $M_k$ and $\Phi$ is the basis function expansion of $\boldsymbol{X}$. Thus, the classifier's output is not directly used to select a model, but rather as a feature upon which the weighting functions $P(z|(\mathbf{\Phi}, s))$ depend, and for initializing the EM algorithm.

## 5.2 Experimental Setup

For the experiments conducted throughout this chapter, two distributions of unstructured SAT instances and two of structured instances were selected, `rand3-fix` and `rand3-var` with 400 variables, `QCP` and `SWGCP`. Each instance set was randomly split into training, validation and test sets, at a ratio of 70:15:15. All parameter tuning was performed with a validation set; test sets were used only to generate the final results reported in this chapter. For each instance, 84 features from nine categories were computed: problem size, variable-clause graph, variable graph, clause graph, balance features, proximity to Horn formulae, LP-based, DPLL search space, and local search space. We used quadratic basis functions in addition to raw features to achieve better runtime prediction accuracy. The accuracy of logarithm runtime prediction was measured by root mean squared error (RMSE).

For uniform random 3-SAT instances, we ran four solvers that are known to perform well on these distributions: `kcnfs` [44], `oksolver` [121], `march_dl` [72], and `satz` [133]. For structured SAT instances, we ran six solvers that are known to perform well on these distributions: `oksolver`, `zchaff` [221], `sato` [220], `satelite` [45], `minisat` [46], and `satzoo` [46]. Note that in the 2005 SAT competition, `satelite` won gold medals for the Industrial and Handmade

SAT+UNSAT categories; `minisat` and `zchaff` won silver and bronze, respectively, for Industrial SAT+UNSAT; and `kcnfs` and `march_dl` won gold and silver, respectively, in the Random SAT+UNSAT category.

For classification, we used Sparse Multinomial Logistic Regression (SMLR) [120], a state-of-the-art sparse classification algorithm. Similar to relevance vector machines and sparse probit regression, SMLR learns classifiers use sparsity-promoting priors to control the expressivity of the learned classifier, thereby tending to result in better generalization. SMLR encourages parameter weights either to be significantly large or exactly zero. It also learns a sparse multi-class classifier that scales favorably in both the number of training samples and the input dimensionality, which is important for our problems since we have tens of thousands of samples per data set. We used the same set of raw features as were available to the regression model, although in this case we did not find it necessary to use a basis-function expansion of these features. Note that, since the goal of this chapter was to obtain the highest classification accuracy, we also used probing features in the study.

We performed all of our experiments using a cluster consisting of 50 computers equipped with dual Intel Xeon 3.2GHz CPUs with 2MB cache and 2GB RAM, running Suse Linux 9.1. All runs of any solver that exceeded 1 CPU hour were terminated and recorded in our database of experimental results with a runtime of 1 CPU hour; this occurred in fewer than 3% of all runs.

## 5.3 Experimental Results

This section presents three sets of results. Section 5.3.1 confirms Nudelman et al.'s observation that much simpler and more accurate empirical hardness models can be learned when all instances are either satisfiable or unsatisfiable [156]. We refer these two models to *conditional* models, while models trained on all instances are referred to *unconditional* models. Let $M_{sat}$ ($M_{unsat}$) denote a model trained only on satisfiable (unsatisfiable) instances. The models equipped with an oracle that knows which conditional model performs better for a given instance are refereed to *oracular* models. (Note that the oracle chooses the *best* model for a particular instance, not the model trained on data with the same *satisfiability status* as the

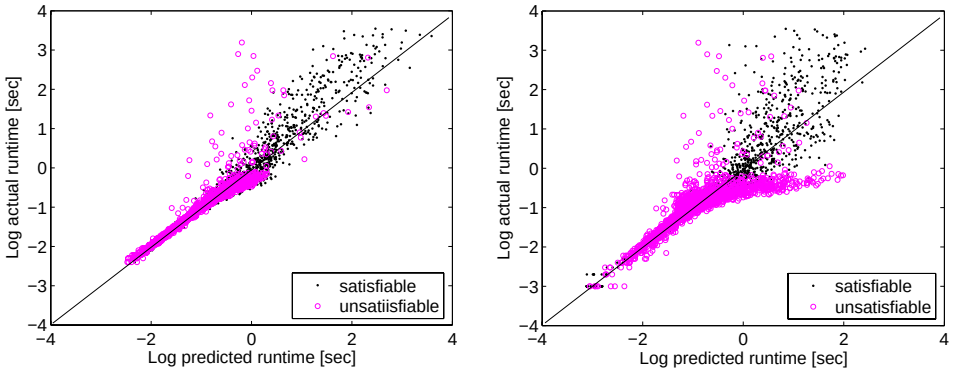



**Figure 5.2:** *Prediction accuracy comparison of oracular model (left, RMSE=0.247) and unconditional model (right, RMSE=0.426). Distribution:* `QCP`*, solver:* `satelite`*.*

instance.) Section 5.3.2 shows that satisfiability can be accurately predicted for a variety of data sets. Section 5.3.3 compares the prediction accuracy of hierarchical hardness models with *unconditional* models and *oracular* models.

### 5.3.1 Performance of Conditional and Oracular Models

Figure 5.2 shows the difference between using oracular models and unconditional models on structured SAT instances (distribution: `QCP`, solver: `satelite`). For oracular models, we observed almost perfect predictions of runtime for unsatisfiable instances and more noisy, but unbiased predictions for satisfiable instances (Figure 5.2, left). Figure 5.2 (right) shows that the runtime prediction for unsatisfiable instances made by unconditional models can exhibit both less accuracy and more bias.

Even though using the best conditional model can result in higher prediction accuracy, there is a large penalty for using the wrong conditional model to predict the runtime of an instance. Figure 5.3 (left) shows that if $M_{sat}$ is used for runtime prediction on an unsatisfiable instance, the prediction error is often very large. The large bias in the inaccurate predictions is due to the fact that models trained on different types of instances are very different. As shown in Figure 5.3 (right), similar phenomena occur when we use $M_{unsat}$ to predict the runtime on a satisfiable

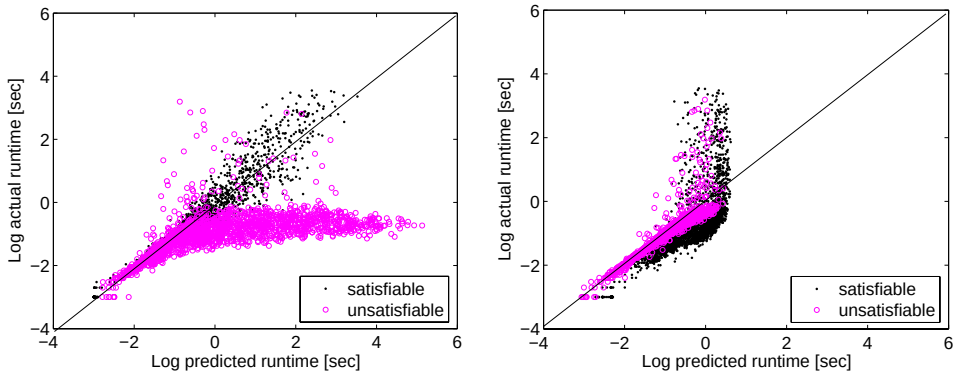



**Figure 5.3:** *Actual vs predicted logarithm runtime using only $M_{sat}$ (left, RMSE=1.493) and only $M_{unsat}$ (right, RMSE=0.683), respectively. Distribution:* `QCP`*, solver:* `satelite`*.*

instance.

The results in Table 5.1 are consistent across data sets and solvers. Oracular models always achieve higher accuracy than unconditional models. The very large prediction errors in Table 5.1 for $M_{sat}$ and $M_{unsat}$ indicate that these models are very different. In particular, the RMSE for using models trained on unsatisfiable instances to predict runtimes on a mixture of instances is as high as 14.914 (distribution: `QCP`, solver: `sato`).

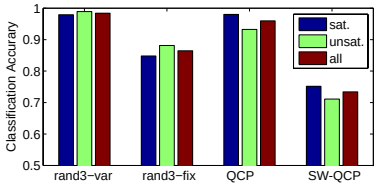
Unfortunately, oracular models rely on information that is unavailable in practice: the respective accuracies of the two conditional models on a given (test) instance. Nevertheless, the prediction accuracies achieved by oracular models suggest that it may be promising to find some practical means of combining conditional models. However, it could also be harmful, as if one makes the wrong choices, prediction error can be much higher than when using an unconditional model.

### 5.3.2 Performance of Classification

Considering the difficulty of the classification task, our experimental results proved very good. Overall accuracy on test data is as high as 98%, and never lower than

| Solvers | RMSE for rand3-var models | | | | RMSE for rand3-fix models | | | |
|---|---|---|---|---|---|---|---|---|
| | sat. | unsat. | unconditional | oracular | sat. | unsat. | unconditional | oracular |
| satz | 5.481 | 3.703 | 0.385 | 0.329 | 0.459 | 0.835 | 0.420 | 0.343 |
| march_dl | 1.947 | 3.705 | 0.396 | 0.283 | 0.604 | 1.097 | 0.542 | 0.444 |
| kcnfs | 4.766 | 4.765 | 0.373 | 0.294 | 0.550 | 0.983 | 0.491 | 0.397 |
| oksolver | 8.169 | 4.141 | 0.443 | 0.356 | 0.689 | 1.161 | 0.596 | 0.497 |
| **Solvers** | **RMSE for QCP models** | | | | **RMSE for SW-GCP models** | | | |
| | sat. | unsat. | unconditional | oracular | sat. | unsat. | unconditional | oracular |
| zchaff | 1.866 | 1.163 | 0.675 | 0.303 | 1.230 | 1.209 | 0.993 | 0.657 |
| minisat | 1.761 | 1.150 | 0.574 | 0.305 | 1.280 | 1.275 | 1.022 | 0.682 |
| satzoo | 1.293 | 0.876 | 0.397 | 0.240 | 0.709 | 0.796 | 0.581 | 0.384 |
| satelite | 1.493 | 0.683 | 0.426 | 0.247 | 1.232 | 1.226 | 0.970 | 0.618 |
| sato | 2.373 | 14.914 | 0.711 | 0.375 | 1.682 | 1.887 | 1.353 | 0.723 |
| oksolver | 1.213 | 1.062 | 0.548 | 0.427 | 1.807 | 2.064 | 1.227 | 0.601 |

**Table 5.1:** *Accuracy of hardness models for different solvers and instance distributions.*




| Dataset | Classification Accuracy | | |
|---|---|---|---|
| | on sat. | on unsat. | overall |
| rand3sat-var | 0.9791 | 0.9891 | 0.9840 |
| rand3sat-fix | 0.8480 | 0.8814 | 0.8647 |
| QCP | 0.9801 | 0.9324 | 0.9597 |
| SW-GCP | 0.7516 | 0.7110 | 0.7340 |

**Figure 5.4:** *Classification accuracy for different data sets*

73%, substantially better than random guessing. Compared to Chapter 4, we show classification can be applied to a large variety of instance distributions. With probing features (e.g., local-search probing), we are able to achieve even better classification accuracy for rand3sat-fix. Furthermore, the classifier is usually very confident about the satisfiability of an instance (i.e., returned probabilities very close to 0 or 1), and the more confident the classifier is, the more accurate it tends to be. These results are summarized in Figures 5.4–5.6.

For the rand3-var data set (Figure 5.5, left), the overall classification error was only 1.6%. Using only the clauses-to-variables ratio (greater or less than 4.26) as the basis for predicting the satisfiability of an instance yielded error of

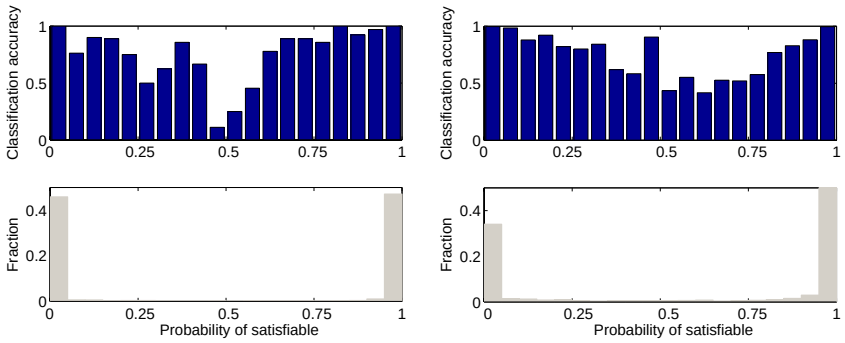



**Figure 5.5:** *Classification accuracy vs classifier output (top) and fraction of instances within the given set vs classifier output (bottom). Left:* `rand3-var`*, right:* `QCP`*.*

3.7%; therefore, by using SMLR rather than this simple classifier, the classification error was halved. On the `QCP` data set (Figure 5.5, right), classification accuracy was 96%, and the classifier was extremely confident in its predictions. For `rand3sat-fix` (Figure 5.6, left), the classification accuracy was even higher, 86% vs. 75% (in the previous chapter), with probing features now included. For `SW-GCP` (Figure 5.6, right), classification accuracy was much lower (73%). This was because the features were less predictive on this instance distribution, which was consistent with the results of unconditional hardness models for `SW-GCP`. Note that the fraction of instances, for which the classifier was confident was smaller for the last two distributions than for `rand3-var` and `QCP`. However, even for `SW-GCP`, there was a strong correlation between the classifier's output and classification accuracy on test data.

One further interesting finding is that classifiers can achieve very high accuracies even given very small sets of features. For example, on the `QCP` data, the SMLR classifier achieved 93% accuracy with only 5 features. The five most important features for classification on all four data sets are shown in Table 5.2. Local-search-based features turned out to be very important for classification in all four data sets, which may be because easy instances can be solved by computing these probing features.

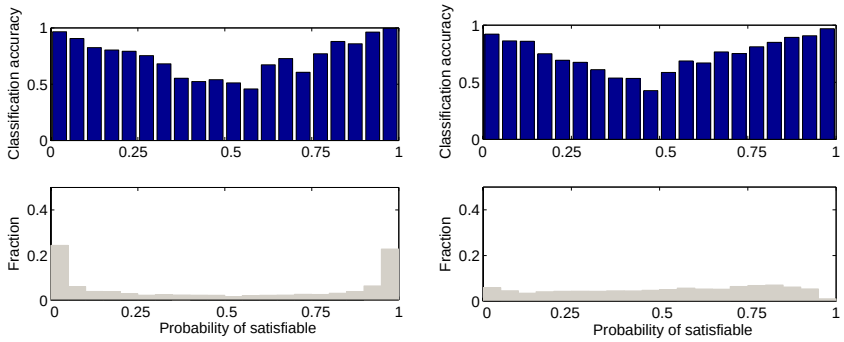



**Figure 5.6:** *Classification accuracy vs classifier output (top) and fraction of the instances within the given set vs classifier output (bottom). Left:* `rand3-fix`*, right:* `SW-GCP`*.*

| **Data sets** | **`rand3-var`** | **`rand3-fix`** |
|---|---|---|
| Five features | gsat_BestCV_Mean<br>saps_BestStep_CoeffVariance<br>lobjois_mean_depth_over_vars<br>VCG_VAR_max<br>saps_BestSolution_Mean | saps_BestSolution_CoeffVariance<br>gsat_BestSolution_Mean<br>saps_BestCV_Mean<br>lobjois_mean_depth_over_vars<br>gsat_BestCV_Mean |
| Accuracy (5 features) | 98.4% | 86.5% |
| Accuracy (all features ) | 98.4% | 86.5% |
| **Data sets** | **`QCP`** | **`SW-GCP`** |
| Five features | lobjois_log_num_nodes_over_vars<br>saps_BestSolution_Mean<br>saps_BestCV_Mean<br>vars_clauses_ratio<br>saps_BestStep_CoeffVariance | vars_reduced_depth<br>gsat_BestCV_Mean<br>nvars<br>VCG_VAR_min<br>saps_BestStep_Mean |
| Accuracy (5 features) | 93.0% | 73.2% |
| Accuracy (all features) | 96.0% | 73.4% |

**Table 5.2:** *The five most important features (listed from most to least important) for classification as chosen by backward selection.*

Overall, the experimental results confirmed that a classifier may be used to make accurate polynomial-time predictions pertaining to the satisfiability of SAT instances. This finding may be useful in its own right. For example, researchers interested in evaluating incomplete SAT algorithms on large numbers of satisfiable instances drawn from a distribution that produces both satisfiable and unsatisfiable

| Solvers | RMSE (rand3-var models) oracular | uncond. | hier. | RMSE (rand3-fix models) oracular | uncond. | hier. |
|---|---|---|---|---|---|---|
| satz | 0.329 | 0.385(85%) | 0.344(96%) | 0.343 | 0.420(82%) | 0.413(83%) |
| march_dl | 0.283 | 0.396(71%) | 0.306(92%) | 0.444 | 0.542(82%) | 0.533(83%) |
| kcnfs | 0.294 | 0.373(79%) | 0.312(94%) | 0.397 | 0.491(81%) | 0.486(82%) |
| oksolver | 0.356 | 0.443(80%) | 0.378(94%) | 0.497 | 0.596(83%) | 0.587(85%) |
| Solvers | RMSE (QCP models) oracular | uncond. | hier. | RMSE (SW-GCP models)* oracular | uncond. | hier. |
| zchaff | 0.303 | 0.675(45%) | 0.577(53%) | 0.657 | 0.993(66%) | 0.983(67%) |
| minisat | 0.305 | 0.574(53%) | 0.500(61%) | 0.682 | 1.022(67%) | 1.024(67%) |
| satzoo | 0.240 | 0.397(60%) | 0.334(72%) | 0.384 | 0.581(66%) | 0.581(66%) |
| satelite | 0.247 | 0.426(58%) | 0.372(66%) | 0.618 | 0.970(64%) | 0.978(63%) |
| sato | 0.375 | 0.711(53%) | 0.635(59%) | 0.723 | 1.352(53%) | 1.345(54%) |
| oksolver | 0.427 | 0.548(78%) | 0.506(84%) | 0.601 | 1.337(45%) | 1.331(45%) |

**Table 5.3:** *Comparison of oracular, unconditional and hierarchical hardness models. The second number of each entry is the ratio of the model's RMSE to the oracular model's RMSE. (*For SW-GCP, even the oracular model exhibits a large runtime prediction error.)*

instances could use a complete search algorithm to label a relatively small training set, and then use the classifier to filter instances.

### 5.3.3 Performance of Hierarchical Models

The broader performance of different unconditional, oracular and hierarchical models is shown in Table 5.3. For rand3-var, classification accuracy was very high (classification error was only 1.6%). Our experiments confirmed that hierarchical hardness models can achieve almost the same runtime prediction accuracy as oracular models for all four solvers considered in our study. Figure 5.7 shows that using the hierarchical hardness model to predict satz's runtime was much better than using the unconditional model.

On the rand3-fix dataset, results for all four solvers were qualitatively similar: hierarchical hardness models gave slightly but consistently better runtime predictions than unconditional models. On this distribution, the gap in prediction accuracy between unconditional and oracular models was already quite small, which made further significant improvements more difficult to achieve. Detailed analysis of actual vs. predicted runtimes for satz (see Figure 5.8) shows that particularly for unsatisfiable instances, the hierarchical model tended to produce slightly more

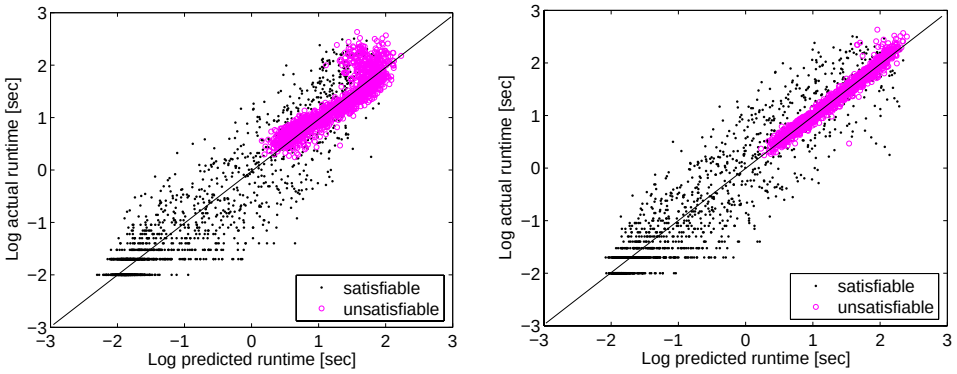



**Figure 5.7:** *Actual vs. predicted logarithm runtime for* `satz` *on* `rand3-var`*. Left: unconditional model (RMSE=0.387); right: hierarchical model (RMSE=0.344).*

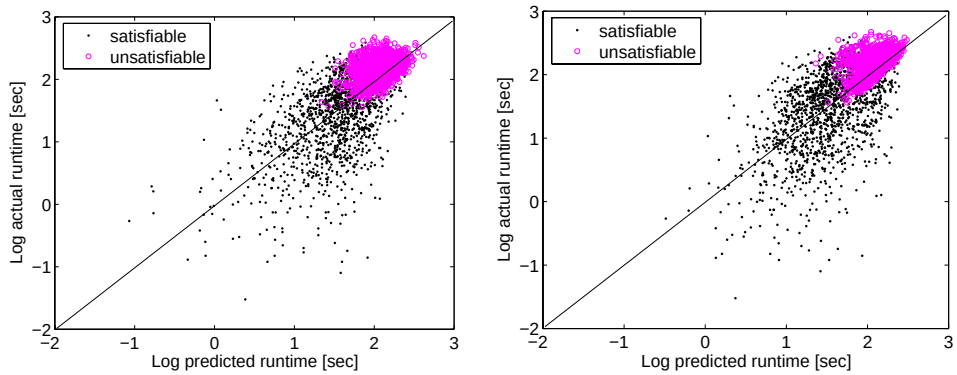



**Figure 5.8:** *Actual vs. predicted logarithm runtime for* `satz` *on* `rand3-fix`*. Left: unconditional model (RMSE=0.420); right: hierarchical model (RMSE=0.413).*

accurate predictions. Further investigation confirmed that those instances in Figure 5.8 (right) that were far away from the ideal prediction line ($y = x$) possessed low classification confidence.

For the structured `QCP` instances, similar runtime prediction accuracy improvements were obtained by using hierarchical models. Since the classification accu-

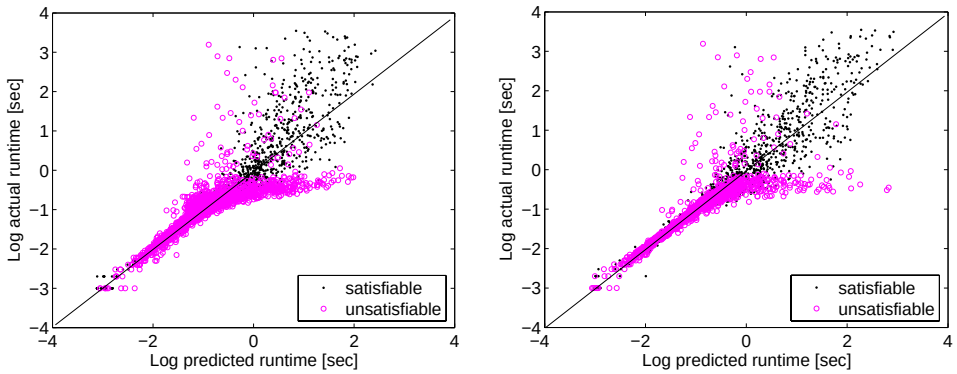



**Figure 5.9:** *Actual vs. predicted logarithm runtime for* `satelite` *on* `QCP`*. Left: unconditional model (RMSE=0.426); right: hierarchical model (RMSE=0.372).*

racy for `QCP` was higher than the classification accuracy for `rand3-fix`, we expected bigger improvements. The experimental results confirm this hypothesis (Figure 5.9). For example, a hierarchical model for the `satelite` solver achieved a RMSE of 0.372, compared to 0.462 obtained from an unconditional model (whereas the oracular model yields RMSE 0.247).

However, the runtime prediction accuracy obtained by hierarchical hardness models depends on the quality of the underlying conditional models (experts). In the case of data set `SW-GCP` (see Figure 5.10), both unconditional and oracular models had fairly large prediction error. This is also consistent with classification error on `SW-GCP` being much higher (26.6%, compared to 4.0% on `QCP` and 13.5% on `rand3sat-fix`).

When investigating the relationship between the classifier's confidence and regression runtime prediction accuracy, we find that higher classification confidence tends to be indicative of more accurate runtime predictions. This relationship is illustrated in Figure 5.11 for the `satelite` solver on the `QCP` data set: when the classifier was more confident about the satisfiability of an instance, both prediction error (Figure 5.11, left) and RMSE (Figure 5.11, right) were smaller.

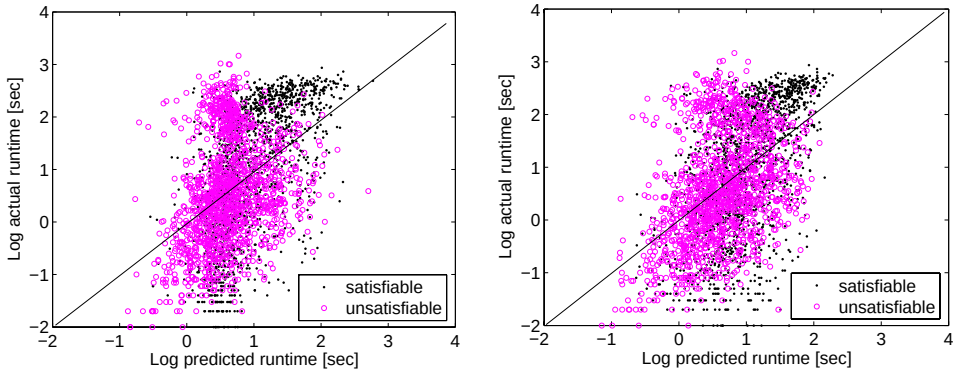



**Figure 5.10:** *Actual vs. predicted logarithm runtime for* `zchaff` *on* `SW-GCP`*. Left: unconditional model (RMSE=0.993); right: hierarchical model (RMSE=0.983).*

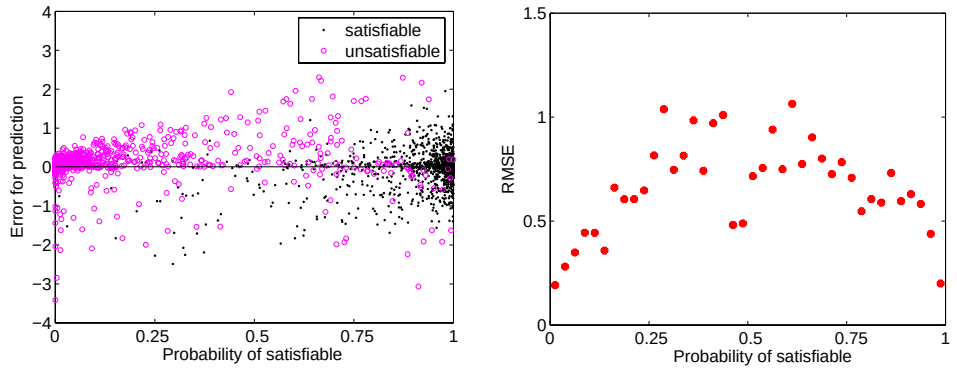



**Figure 5.11:** *Classifier output vs runtime prediction error (left); relationship between classifier output and RMSE (right). Data set:* `QCP`*, solver:* `satelite`*.*

## 5.4 Conclusions

This chapter shows that there are big differences between models trained only on satisfiable and unsatisfiable instances, not only for uniform random 3-SAT (as was previously reported in Nudelman et al. (2004)), but also for distributions of struc-

tured SAT instances, such as QCP and SW-GCP. Furthermore, these models have higher prediction accuracy than the respective unconditional models. A classifier can be used to distinguish between satisfiable and unsatisfiable instances for all above distributions with high accuracy. Compared to Chapter 4, we confirm that adding probing features significantly improves classification accuracy. Furthermore, such a classifier can be combined with conditional hardness models into a hierarchical hardness model using a mixture-of-experts approach. In cases of high classification accuracy, the hierarchical models thus obtained always offered substantial improvements over an unconditional model. In the case of less accurate classification, the hierarchical models could not offer a substantial improvement over the unconditional model; however, hierarchical models were never significantly worse. It should be noted that the hierarchical models come at virtually no additional computational cost, as they depend on the same features as used for the individual regression models.

# Chapter 6

# Performance Prediction with Empirical Performance Models

The previous chapter has demonstrated that an algorithm's runtime can be predicted with high accuracy by using linear regression models. However, we neither have to use linear regression models nor have to predict runtime. This chapter extends *empirical hardness models* (EHMs) to *empirical performance models* (EPMs) to reflect these broadened scopes. Such models have important applications to algorithm analysis, portfolio-based algorithm selection, and the automatic configuration of parameterized algorithms.

Over the past decade, a wide variety of techniques have been studied for building such models. In this chapter, we describe a thorough comparison of different existing and new model building techniques for SAT, MIP, and TSP. Our experiments consider 11 algorithms and 35 instance distributions with the least structured having been generated uniformly at random and the most structured having emerged from real industrial applications. Overall, we demonstrate that our new models yield substantially better runtime predictions than previous approaches. [1]

[1] This chapter is based on the joint work with Frank Hutter, Holger Hoos, and Kevin Leyton-Brown [101].

## 6.1 Methods Used in the Literature

The goal of performance prediction is to find a mapping from instance features to algorithm performance. Therefore, any regression technique can be applied. In this chapter, we only consider machine learning methods that have previously been used to predict algorithm runtimes: ridge regression [28, 29, 88, 92, 127, 130, 156, 209, 210], neural networks [184], Gaussian process regression [92], and regression trees [15].

Let $\boldsymbol{X}_i$ denote a $p$-dimensional feature vector for instance $i$. Let $\boldsymbol{y}_i$ denote the performance of an algorithm A on $i$. An EPM is a stochastic process $f : \boldsymbol{X} \mapsto \boldsymbol{y}$ that defines a probability distribution over performance measures $\boldsymbol{y}$ for A and problem instances with features $\boldsymbol{X}$. The prediction at a particular input is a distribution over performance values. Let $\Pi$ be the set of $n$ training instances drawn from an instance distribution D. The training data for the regression models is simply $\{(\boldsymbol{X}_i, y_i)\}_{i=1}^{n}$. Throughout this chapter, we focus on runtime as a performance measure and use a log transformation. Since many of the methods yield only point-valued runtime predictions, our experimental analysis focuses on the accuracy of mean predictions.

### 6.1.1 Ridge Regression

Ridge regression [see, e.g., 24] is a simple regression method that fits a linear function $f_{\boldsymbol{w}}(\boldsymbol{X})$ of its inputs $\boldsymbol{X}$. Due its simplicity (both conceptual and computational) and its interpretability, ridge regression has been widely used for building EPMs [92, 127, 130, 156, 208].

As mentioned in the previous chapter, two key techniques are widely used to better approximate algorithm performance: feature expansion for extending feature space and feature selection for removing redundant features. Many different methods exist for feature expansion and selection, and we review three different ridge regression variants from the recent literature that only differ in these design decisions.

#### Ridge Regression Variant RR: Two-phase forward selection [209, 210]

Chapter 5 uses a simple and scalable feature selection method based on forward selection [see e.g., 68] to build the regression models. This iterative method starts

with an empty input set, greedily adds one linear input at a time to minimize cross-validation error at each step, and stops when $l$ linear inputs have been selected. It then performs a full quadratic expansion of these $l$ linear features (using the original features, and then normalizing the resulting quadratic features again to have mean zero and standard deviation one). Finally, it carries out another pass of forward selection with the expanded feature set to select $q$ features. One advantage of this two-phase approach is scalability: it can handle a very large number of features. The computational complexity of forward selection can be reduced by exploiting the fact that the inverse matrix $A'^{-1}$ resulting from including one additional feature can be computed incrementally by two rank-one updates of the previous inverse matrix $A^{-1}$, requiring quadratic time rather than cubic time [181].

In the experiments, we fix $l = 30$ to keep the result of a full quadratic basis function expansion manageable in size. Free parameters are the maximum number of quadratic terms $q$ and the ridge penalizer $\varepsilon$; by default, we use $q = 20$ and $\varepsilon = 10^{-3}$.

**Ridge Regression Variant SPORE-FoBa: Forward-backward selection [88]**

Recently, Huang et al. (2010) described a method for predicting algorithm runtime, termed Sparse POlynomial REgression (SPORE), which is based on ridge regression with forward-backward (FoBa) feature selection. In contrast to the two RR variants, SPORE-FoBa employs a cubic feature expansion (based on its own normalization of the original predictors). Essentially, it performs a single pass of forward selection, adding a small *set* of terms at each step determined by a forward-backward phase on a feature's candidate set. Specifically, having already selected a set of terms $T$ based on raw features $S$, SPORE-FoBa loops over all raw features $r \notin S$, constructing a candidate set $T_r$ that consists of all polynomial expansions of $S \cup \{r\}$ that include $r$ with non-zero degree and whose total degree is bounded by 3. For each such candidate set $T_r$, the forward-backward phase iteratively adds the best term $t \in T \setminus T_r$ if its reduction $\chi$ of root mean squared error (RMSE) exceeds a threshold $\gamma$ (forward step), and then removes the worst term $t \in T$, if its reduction of RMSE is below $0.5 \cdot \gamma$ (backward step). This phase terminates when no single term $t \in T \setminus T_r$ can be added to reduce RMSE by more than $\gamma$. Finally,

SPORE-FoBa's outer forward selection loop chooses the $T$ resulting from the best of its forward-backward phases, and iterates until the number of terms in $T$ reach a pre-specified maximum of $t_{\max}$ terms. SPORE-FoBa's free parameters are the ridge penalizer $\varepsilon$, $t_{\max}$, and $\gamma$, with defaults $\varepsilon = 10^{-3}$, $t_{\max} = 10$, and $\gamma = 0.01$.

### 6.1.2 Neural Networks

Neural networks are a well-known regression method inspired by information processing in human brains. The multilayer perceptron (MLP) is a particularly popular type of neural network that organizes single computational units ("neurons") in layers (input, hidden, and output layers), using the outputs of all units in a layer as the inputs of all units in the next layer. Each neuron $n_i$ in the hidden and output layers with $k$ inputs $\mathbf{a_i} = [a_{i,1}, \ldots, a_{i,k}]$ has an associated weight vector $\mathbf{w_i} = [w_{i,1}, \ldots, w_{i,k}]$ and a bias term $b_i$, and computes a function $\mathbf{w_i}^\intercal \mathbf{a_i} + b_i$. For neurons in the hidden layer, the result of this function is further propagated through a nonlinear activation function $g : \mathbb{R} \to \mathbb{R}$ (which is often instantiated as tanh). Given an input $\mathbf{x} = [x_1, \ldots, x_p]$, a network with a single hidden layer of $h$ neurons $n_1, \ldots, n_h$ and a single output neuron $n_{h+1}$ then computes output

$$\hat{f}(\mathbf{x}) = \left( \sum_{j=1}^{h} g(\mathbf{w_j}^\intercal \cdot \mathbf{x} + b_j) \cdot w_{h+1,j} \right) + b_{h+1}.$$

The $p \cdot h + h$ weight terms and $h+1$ bias terms can be combined into a single weight vector $\mathbf{w}$, which can be set to minimize the network's prediction error using any continuous optimization algorithm (e.g., the classic "backpropagation" algorithm performs gradient descent to minimize squared prediction error).

Smith-Miles and van Hemert (2011) used an MLP with one hidden layer of 28 neurons to predict the runtime of local search algorithms for solving timetabling instances. They used the proprietary neural network software Neuroshell, but advised us to compare to an off-the-shelf Matlab implementation instead. We thus employed the popular Matlab neural network package NETLAB [151]. NETLAB uses activation function $g = \tanh$ and supports a regularizing prior on weights, minimizing the error metric $\sum_i^N (\hat{f}(\mathbf{x_i}) - y_i)^2 + \alpha \cdot \mathbf{w}^\intercal \cdot \mathbf{w}$, where $\alpha$ is a parameter determining the prior's strength. In the experiments, we used NETLAB's default

optimizer (scaled conjugate gradients, SCG), to minimize this error metric, stopping the optimization after the default of 100 SCG steps. Free parameters are the regularization prior $\alpha$ and the number of hidden neurons $h$; for the default, we used NETLAB's default $\alpha = 0.01$ and, like Smith-Miles and van Hemert [184], $h = 28$.

### 6.1.3 Gaussian Process Regression

Stochastic Gaussian processes (GPs) [168] are a popular class of regression models with roots in geostatistics, where they are also termed Kriging models [119]. GPs are the dominant modern approach for building response surface models for modeling a process when the underlying fundamental mechanism is largely unknown [14, 109, 172, 174].

To construct a GP regression model, we first select a parameterized kernel function $k_\lambda$ to characterize the degree of similarity between two elements of the input space. We also need to determine the variance $\sigma^2$ of Gaussian-distributed observation noise, which in our setting corresponds to the variance of the target algorithm's runtime distribution. The predictive distribution of a zero-mean Gaussian stochastic process for response $y_{n+1}$ at input $\boldsymbol{X}_{n+1}$ given training data $\mathscr{D} = \{(\boldsymbol{X}_1, y_1), \ldots, (\boldsymbol{X}_n, y_n)\}$, measurement noise variance $\sigma^2$, and kernel function $k$, is then the Gaussian

$$p(y_{n+1}|\boldsymbol{X}_{n+1}, \boldsymbol{X}_{1:n}, \boldsymbol{y}_{1:n}) = \mathscr{N}(y_{n+1}|\mu_{n+1}, \mathrm{Var}_{n+1}) \tag{6.1}$$

with mean and variance

$$\begin{aligned} \mu_{n+1} &= \boldsymbol{k_*}^\mathsf{T} \cdot [\boldsymbol{K} + \sigma^2 \cdot \boldsymbol{I_n}]^{-1} \cdot \boldsymbol{y}_{1:n} \\ \mathrm{Var}_{n+1} &= k_{**} - \boldsymbol{k_*}^\mathsf{T} \cdot [\boldsymbol{K} + \sigma^2 \cdot \mathbf{I}]^{-1} \cdot \boldsymbol{k_*}, \end{aligned}$$

where

$$\begin{aligned}
\boldsymbol{K} &= \begin{pmatrix} k(\boldsymbol{X}_1,\boldsymbol{X}_1) & \dots & k(\boldsymbol{X}_1,\boldsymbol{X}_n) \\ & \ddots & \\ k(\boldsymbol{X}_n,\boldsymbol{X}_1) & \dots & k(\boldsymbol{X}_n,\boldsymbol{X}_n) \end{pmatrix} \\
\boldsymbol{k}_* &= (k(\boldsymbol{X}_1,\boldsymbol{X}_{n+1}),\dots,k(\boldsymbol{X}_n,\boldsymbol{X}_{n+1}))^\mathsf{T} \\
k_{**} &= k(\boldsymbol{X}_{n+1},\boldsymbol{X}_{n+1}) + \sigma^2.
\end{aligned}$$

Refer to Rasmussen and Williams [168] for a derivation.

A variety of kernel functions are possible, but the most common choice for continuous predictors is the squared exponential kernel

$$K_{\text{cont}}(\boldsymbol{X}_i,\boldsymbol{X}_j) = \exp\left[\sum_{l=1}^{p}(-\lambda_l \cdot (X_{i,l} - X_{j,l})^2)\right], \tag{6.2}$$

where $\lambda_1,\dots,\lambda_p$ are kernel parameters. This kernel is most appropriate if the response is expected to vary smoothly in the predictors $\boldsymbol{X}$.

The GP equations above assume fixed kernel parameters $\lambda_1,\dots,\lambda_p$ and fixed observation noise variance $\sigma^2$. These constitute the GP's *hyperparameters*, which are typically set by maximizing the *marginal likelihood* $p(\boldsymbol{y}_{1:N})$ of the data with a gradient-based optimizer. We refer to Rasmussen and Williams [168] for the equations. The choice of optimizer can make a substantial difference in practice. We used `minFunc` [176] with its default setting of a limited-memory version of BFGS [154].

Learning a GP model from data can be computationally expensive. Inverting the $n$ by $n$ matrix $[\boldsymbol{K} + \sigma^2 \cdot \mathbf{I_n}]$ takes $O(n^3)$ time and has to be performed in every step of the hyperparameter optimization ($h$ steps in total), yielding a total complexity of $O(h \cdot n^3)$. Subsequent predictions at a new input are relatively cheap: $O(n)$ and $O(n^2)$ for predictions of the mean and the variance, respectively.

### 6.1.4 Regression Trees

Regression trees [27] are simple tree-based regression models. They are known to handle discrete predictors well; we believe that their first application to the predic-

tion of algorithm runtime was by Bartz-Beielstein and Markon (2004).

The leaf nodes of regression trees partition the input space into disjoint regions $R_1, \ldots, R_M$, and use a simple model for prediction in each region $R_m$; the most common choice is to predict a constant $c_m$. This leads to the following prediction for an input point $\boldsymbol{X}$: $\hat{\mu}(\boldsymbol{X}) = \sum_{m=1}^{M} c_m \cdot \mathbb{I}_{\boldsymbol{X} \in R_m}$, where the indicator function $\mathbb{I}_z$ takes value 1 if the proposition $z$ is true and 0 otherwise. Note that since the regions $R_m$ partition the input space, this sum will always involve only one nonzero term. We denote the subset of training data points in region $R_m$ as $\mathscr{D}_m$. Under the standard squared error loss function $\sum_{i=1}^{n} (y_i - \hat{\mu}(\boldsymbol{X}_i))^2$, the error-minimizing choice of constant $c_m$ in region $R_m$ is then the sample mean of the data points in $\mathscr{D}_m$:

$$c_m = \frac{1}{|\mathscr{D}_m|} \sum_{\boldsymbol{X}_i \in R_m} y_i. \tag{6.3}$$

To construct a regression tree, we use the following standard recursive procedure (see, e.g., Breiman et al. (1984)), which starts at the root of the tree with all available training data points $\mathscr{D} = \{(\boldsymbol{X}_i, y_i)\}_{i=1}^{n}$. We consider binary partitionings of a given node's data along *split variables* $j$ and *split points* $s$. For a real-valued split variable $j$, $s$ is a scalar and data point $\boldsymbol{X}_i$ is assigned to region $R_1(j,s)$ if $X_{i,j} \leq s$ and to region $R_2(j,s)$ otherwise. For a categorical split variable $j$, $s$ is a set, and data point $\boldsymbol{x}_i$ is assigned to region $R_1(j,s)$ if $X_{i,j} \in s$ and to region $R_2(j,s)$ otherwise. At each node, we select split variable $j$ and split point $s$ to minimize the sum of squared differences to the regions' means,

$$l(j,s) = \left[ \sum_{\boldsymbol{X}_i \in R_1(j,s)} (y_i - c_1)^2 + \sum_{\boldsymbol{X}_i \in R_2(j,s)} (y_i - c_2)^2 \right], \tag{6.4}$$

where $c_1$ and $c_2$ are chosen according to Equation (6.3) as the sample means in regions $R_1(j,s)$ and $R_2(j,s)$ respectively. We continue this procedure recursively, finding the best split variable and split point, partitioning the data into two child nodes, and recursing into the child nodes. The process terminates when all training data points in a node share the same $\boldsymbol{x}$ values, meaning that no more splits are possible. This procedure tends to overfit the data, which can be mitigated by recursively pruning away branches that contribute little to the model's predictive

accuracy. We use cost-complexity pruning with 10-fold cross-validation to identify the best tradeoff between complexity and predictive quality (see, e.g., Hastie et al. (2009) for details).

In order to predict the response value at a new input, $\boldsymbol{X}_i$, we *propagate* $\boldsymbol{X}$ *down the tree*, that is, at each node with split variable $j$ and split point $s$, we continue to the left child node if $\boldsymbol{X}_{i,j} \leq s$, and to the right child node otherwise. The predictive mean for $\boldsymbol{X}_i$ is the constant $c_m$ in the leaf that this process selects; there is no variance predictor.

## 6.2 New Modeling Techniques for EPMs

We also introduce some advanced machine learning techniques, and apply them for the first time to algorithm performance prediction. In particular, we introduce two new, more sophisticated, runtime prediction models. The first is based on an approximate version of Gaussian processes that scale gracefully to many data points, and also includes a new kernel function for handling categorical data. The second one is based on random forests, collections of regression trees that yield much better predictions than single trees and are known to perform well for discrete inputs.

### 6.2.1 Scaling to Large Amounts of Data with Approximate Gaussian Processes

Fitting Gaussian processes has complexity cubic in the number of data points, which limits the amount of data that can practically be used to fit these models. To deal with this obstacle, the machine learning literature has proposed various approximations to Gaussian processes [see, e.g., 167]. We experimented with the Bayesian committee machine [199], the informative vector machine [123], and the projected process (PP) approximation [168]. All of these methods performed rather similarly, with a slight edge for the PP approximation. Following, we give the final equations for the PP's predictive mean and variance (see, e.g., Rasmussen and Williams (2006) for a derivation).

The PP approximation to GPs uses a subset of $a$ of the $n$ training data points, the so-called *active set*. Let $v$ be a vector consisting of the indices of these $a$ data

points. We extend the notation for exact GPs (see Section 6.1.3) as follows: let $K_{\text{aa}}$ denote the $a$ by $a$ matrix with $K_{\text{aa}}(i,j) = k(\boldsymbol{x}_{v(i)}, \boldsymbol{x}_{v(j)})$ and let $\boldsymbol{K}_{\text{an}}$ denote the $a$ by $n$ matrix with $\boldsymbol{K}_{\text{an}}(i,j) = k(\boldsymbol{x}_{v(i)}, \boldsymbol{x}_j)$. The predictive distribution of the PP approximation is then a Gaussian with mean and variance

$$\begin{aligned} \mu_{n+1} &= \boldsymbol{k}_*{}^{\mathsf{T}} \cdot (\sigma^2 \cdot \boldsymbol{K}_{\text{aa}} + \boldsymbol{K}_{\text{an}} \cdot \boldsymbol{K}_{\text{an}}^{\mathsf{T}})^{-1} \boldsymbol{K}_{\text{an}} \cdot y_{1:n} \\ \text{Var}_{n+1} &= k_{**} - \boldsymbol{k}_*{}^{\mathsf{T}} \cdot \boldsymbol{K}_{\text{aa}}^{-1} \cdot \boldsymbol{k}_* + \sigma^2 \cdot \boldsymbol{k}_*{}^{\mathsf{T}} \cdot (\sigma^2 \cdot \boldsymbol{K}_{\text{aa}} + \boldsymbol{K}_{\text{an}} \cdot \boldsymbol{K}_{\text{an}}^{\mathsf{T}})^{-1} \cdot \boldsymbol{k}_*. \end{aligned}$$

We perform $h$ steps of hyperparameter optimization based on a standard GP trained using a set of $a$ data points sampled uniformly at random without replacement from the $n$ input data points. We then use the resulting hyperparameters and another independently sampled set of $a$ data points (sampled in the same way) for the subsequent PP approximation. In both cases, if $a > n$, we only use $n$ data points.

The complexity of the PP approximation is superlinear only in $a$, meaning that the approach is much faster when we choose $a << n$. The hyperparameter optimization based on $a$ data points takes $O(h \cdot a^3)$ time. In addition, there is a one-time cost of $O(a^2 \cdot n)$ for evaluating the PP equations. Thus, the complexity for fitting the approximate GP model is $O([ha+n] \cdot a^2)$, as compared to $O(h \cdot n^3)$ for the exact GP model. The complexity for predictions with this PP approximation is $O(a)$ for the mean and $O(a^2)$ for the variance of the predictive distribution [168], as compared to $O(n)$ and $O(n^2)$ for the exact version. In our experiments we set $a = 300$ and $h = 50$ to achieve a good compromise between speed and predictive accuracy.

### 6.2.2 Random Forest Models

Random forests [26] are a flexible tool for regression and classification, and are particularly effective for high-dimensional and discrete input data. To the best of our knowledge, they have not yet been used for algorithm runtime predictions except in the most recent work on algorithm configuration [98, 99] performed by our own group. Here, we describe the standard RF framework and some nonstandard implementation choices.

### The Standard Random Forest Framework

A random forest (RF) consists of a set of regression trees. If grown to sufficient depths, regression trees can capture very complex interactions and thus have low bias. However, they can also have high variance: small changes in the data can lead to a dramatically different tree. Random forests [26] reduce this variance by aggregating predictions across multiple different trees. These trees are made to be different in one of two ways: by training them on different subsamples of the training data, or by permitting only a random subset of the variables as split variables at each node. Our experiments show slightly worse performance with a combination of the two approaches. Therefore, we chose the latter option with the full training set for each tree.

Mean predictions for a new input $\boldsymbol{x}$ are trivial: predict the response for $\boldsymbol{x}$ with each tree and average the predictions. Predictive quality improves as the number of trees $B$ grows, but computational cost also grows linearly in $B$. We used $B = 10$ throughout our experiments to keep computational costs low. Random forests have two additional hyperparameters: the percentage of variables to consider at each split point, *perc*, and the minimal number of data points required in a node to make it eligible to be split further, $n_{\text{min}}$. We set $perc = 0.5$ and $n_{\text{min}} = 5$ by default.

### Modifications to Standard Random Forests

We introduce a simple, yet effective, method for quantifying predictive uncertainty in random forests. (Our method is similar to that of Meinshausen [143], who recently introduced quantile regression trees that allow for predictions of quantiles of the predictive distribution; in contrast, we predict mean and variance.) In each leaf of each regression tree, in addition to the empirical mean of the training data associated with that leaf, we store that data's empirical variance. To avoid making deterministic predictions for leaves with few data points, we round the stored variance up to at least the constant ${\sigma^2}_{\text{min}}$; we set ${\sigma^2}_{\text{min}} = 0.01$ throughout. For any input, each regression tree $T_b$ thus yields a predictive mean $\mu_b$ and a predictive variance $\sigma_b^2$. To combine these estimates into a single estimate, we treat the forest as a mixture model of $B$ different models. We denote the random variable for the prediction of tree $T_b$ as $L_b$ and the overall prediction as $L$, and then have

$L = L_b$ if $Y = b$, where $Y$ is a multinomial variable with $p(Y = i) = 1/B$ for $i = 1, \ldots, B$. The mean and variance for $L$ can then be expressed as follows:

$$\begin{aligned}
\mu = \mathbb{E}[L] &= \frac{1}{B}\sum_{b=1}^{B}\mu_b \\
\sigma^2 = \mathrm{Var}(L) &= \mathbb{E}[\mathrm{Var}(L|Y)] + \mathrm{Var}(\mathbb{E}[L|Y]) \\
&= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2\right) + \left(\mathbb{E}[\mathbb{E}(L|Y)^2] - \mathbb{E}[\mathbb{E}(L|Y)]^2\right) \\
&= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2\right) + \left(\frac{1}{B}\sum_{b=1}^{B}\mu_b^2\right) - \mathbb{E}[L]^2 \\
&= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2 + \mu_b^2\right) - \mu^2.
\end{aligned}$$

Thus, the mean prediction is simply the mean across the individual trees' mean predictions. To compute the variance prediction, we used the law of total variance [206], which computes the total variance as the variance across the individual trees' mean predictions (predictions are uncertain if the trees disagree), plus the average variance of each tree (predictions are uncertain if the individual tree predictions are uncertain).

A second nonstandard ingredient in our models concerns the choice of split points. Consider splits on a real-valued variable $j$. Note that when the loss in Equation (6.4) is minimized by choosing split point $s$ between the values of $\boldsymbol{x}_{k,j}$ and $\boldsymbol{x}_{l,j}$, one is still free to choose the exact location of $s$ anywhere in the interval $(\boldsymbol{x}_{k,j}, \boldsymbol{x}_{l,j})$. In typical implementations, $s$ is chosen as the midpoint between $\boldsymbol{x}_{k,j}$ and $\boldsymbol{x}_{l,j}$. Instead, here we draw it uniformly at random from $(\boldsymbol{x}_{k,j}, \boldsymbol{x}_{l,j})$. In the limit of an infinite number of trees, this leads to a linear interpolation of the training data instead of a partition into regions of constant prediction. Furthermore, it causes variance estimates to vary smoothly and to grow with the distance from observed data points.

### Complexity of Fitting Random Forests

The computational cost for fitting a random forest is relatively low. We need to fit $B$ regression trees, each of which is somewhat easier to fit than a normal regression

| Abbreviation | Reference Section | Description |
|---|---|---|
| RR | 6.1.1 | Ridge regression with 2-phase forward selection |
| SP | 6.1.1 | SPORE-FoBa (ridge regression with forward-backward selection) |
| NN | 6.1.2 | Feedforward neural network with one hidden layer |
| PP | 6.2.1 | Projected process (approximate Gaussian process); optimized via minFunc |
| RT | 6.1.4 | Regression tree with cost-complexity pruning |
| RF | 6.2.2 | Random forest |

**Table 6.1:** Overview of our models.

tree since at each node we only consider $v = \max(1, \lfloor perc \cdot p \rfloor)$ out of the $p$ possible split variables. Building $B$ trees simply takes $B$ times as long as building a single tree. Thus, the complexity of learning a random forest is $O(B \cdot v \cdot n^2 \cdot \log n)$ in the worst case (splitting off one data point at a time) and $O(B \cdot v \cdot n \cdot \log^2 n)$ in the best case (perfectly balanced trees).

Prediction with a random forest model entails predicting with $B$ regression trees (plus an $O(B)$ computation to compute mean and variance across those predictions). The complexity of predictions is thus $O(B \cdot n)$ in the worst case and $O(B \cdot \log n)$ for perfectly balanced trees.

## 6.3 Experimental Setup

Table 6.1 provides an overview of the models we evaluated; we evaluate each method's accuracy at predicting the runtime of a variety of solvers for SAT, MIP, and TSP on multiple benchmarks, which are described in Chapter 3.

For SAT, we used a wide range of instance distributions. Briefly, `INDU`, `HAND`, and `RAND` are collections of industrial, handmade, and random instances from the international SAT competitions and races. `COMPETITON` is their union. `SWV` and `IBM` are sets of software and hardware verification instances, and `SWV-IBM` is their union. Finally, `RANDSAT` is a subset of `RAND` containing only satisfiable instances. For all distributions but this last one, we ran the popular tree search solver, `Minisat 2.0` [46]. For `INDU`, `SWV` and `IBM`, we also ran two more solvers: the winner of the SAT Race 2010, `CryptoMiniSat` [186], and `SPEAR` [8] (which has shown state-of-the-art performance on `IBM` and `SWV` with optimized parameter settings [93]). Finally, to evaluate predictions for local search algorithms, we used the `RANDSAT` instances, and considered two solvers: `TNM` [203] (the winner of the

SAT 2009 competition in the random satisfiable category) and the dynamic local search algorithm SAPS [91] (which we see as a baseline).

For MIP, we used two instance distributions from computational sustainability (RCW and CORLAT), one from winner determination in combinatorial auctions (REG), two unions of these (CR and CRR), and a very heterogeneous mix of publicly available MIP instances (BIGMIX). We used the two state-of-the-art commercial solvers, CPLEX 12.1 [104] and Gurobi 2.0 [67]. We also used the two strongest non-commercial solvers, SCIP 1.2.1.4 [17] and lp_solve 5.5 [16].

For TSP, we used three instance distributions: uniform random instances (PORTGEN), random clustered instances (PORTCGEN), and TSPLIB, a heterogeneous set of prominent TSP instances. We ran the most prominent systematic and local search algorithms, Concorde [5] and LK-H [70]. For the latter, we computed runtimes as the time required to find a solution with optimal quality.

For each algorithm–distribution pair, we executed the algorithm with its default parameter setting on all instances of the distribution, measured runtimes, and collected the results. All algorithm runs were executed on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1; runtimes were measured as CPU time on these reference machines. We cut off each algorithm run after one CPU hour; this gives rise to *capped* runtime observations, because we only observe a lower bound on the runtime. Like most past work on runtime modeling, we simply counted such capped runs as having taken one hour. Due to the resolution of the CPU timer, runtimes below 0.01 seconds are measured as 0 seconds. To make $y_i = \log(r_i)$ well defined in these cases, such low runtimes are counted as 0.005.

Since the goal is to compare different model construction techniques, all the features listed in Chapter 3 were used. The features that have constant value across all training data points are removed and the remaining ones are normalized to have mean 0 and standard deviation 1. For some instances, certain feature values were missing because of timeouts, crashes, or because they were undefined (e.g., because preprocessing already solved an instance). These *missing values* occur relatively rarely, a simple mechanism is used for handling them: disregard missing values for the purposes of normalization, and then set the missing values to zero. This means that the feature's value is at the data mean and is thus minimally in-

formative; in some models (ridge regression and neural networks), this mechanism actually lead us to ignore the feature, since its weight is multiplied by zero.

We evaluate methods for building empirical performance models by assessing the quality of the predictions they make about inputs that were not used to train the model. This can be done visually (for example, in the scatter plots in Figure 6.1), or quantitatively. We considered three complementary quantitative metrics to evaluate mean predictions $\{\mu_i\}_{i=1}^n$ and predictive variances $\{\sigma^2\}_{i=1}^n$ given true performances $\{y_i\}_{i=1}^n$. *Root mean squared error (RMSE)* is defined as $\sqrt{1/n \cdot \sum_{i=1}^n (y_i - \mu_i)^2}$; *Pearson's correlation coefficient (CC)* is defined as $(\sum_{i=1}^n (\mu_i y_i) - n \cdot \bar{\mu} \cdot \bar{y})/((n-1) \cdot s_\mu \cdot s_y)$, where $\bar{x}$ and $s_x$ denote sample mean and standard deviation of $x$; and *log likelihood (LL)* is defined as $\sum_{i=1}^n \log \varphi(\frac{y_i - \mu_i}{\sigma_i})$, where $\varphi$ denotes the probability density function (PDF) of a standard normal distribution. Intuitively, LL is the log probability of observing the true values $y_i$ under the predicted distributions $\mathcal{N}(\mu_i, \sigma_i^2)$. For CC and LL, higher values are better, while for RMSE lower values are better. We used $k$-fold cross-validation and report means of these measures across the $k$ folds. Scatter plots show cross-validated predictions for a random subset of up to 1 000 data points.

## 6.4 Experimental Results

Table 6.2 provides quantitative results for all benchmarks, and Figure 6.1 visualizes results. At the broadest level, we can conclude that most of the methods were able to capture sufficient information pertaining to algorithm performance on training data to make meaningful predictions on test data, most of the time: easy instances tended to be predicted as easy, and hard ones as hard. For example, in the case of predicting the runtime of `Minisat 2.0` on a heterogeneous mix of SAT competition instances (refer to the leftmost column in Figure 6.1 and the top row of Table 6.2), `Minisat 2.0` runtimes varied by almost six orders of magnitude, while predictions with the better models rarely were off by more than one order of magnitude (outliers may draw the eye in the scatter plot, but quantitatively, the RMSE for predicting $\log_{10}$ runtime is low – e.g., 0.47 for random forests, which means an average misprediction of a factor of 3).

In our experiments, random forests were indeed the overall winner among the

| | RMSE | | | | | | Time to learn model (s) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Domain** | **RR** | **SP** | **NN** | **PP** | **RT** | **RF** | **RR** | **SP** | **NN** | **PP** | **RT** | **RF** |
| Minisat-COMPETITON | 1.01 | 1.25 | 0.62 | 0.92 | 0.68 | **0.47** | **6.8** | 28.08 | 21.84 | 46.56 | 20.96 | 22.42 |
| Minisat-HAND | 1.05 | 1.34 | 0.63 | 0.85 | 0.75 | **0.51** | **3.7** | 7.92 | 6.2 | 44.14 | 6.15 | 5.98 |
| Minisat-RAND | 0.64 | 0.76 | **0.38** | 0.55 | 0.5 | **0.37** | **4.46** | 7.98 | 10.81 | 46.09 | 7.15 | 8.36 |
| Minisat-INDU | 0.94 | 1.01 | 0.78 | 0.86 | 0.71 | **0.52** | **3.68** | 7.82 | 5.57 | 48.12 | 6.36 | 4.42 |
| Minisat-SWV-IBM | 0.53 | 0.76 | 0.32 | 0.52 | 0.25 | **0.17** | 3.51 | 6.35 | 4.68 | 51.67 | 4.8 | **2.78** |
| Minisat-IBM | 0.51 | 0.71 | 0.29 | 0.34 | 0.3 | **0.19** | 3.2 | 5.17 | 2.6 | 46.16 | 2.47 | **1.5** |
| Minisat-SWV | 0.35 | 0.31 | 0.16 | 0.1 | 0.1 | **0.08** | 3.06 | 4.9 | 2.05 | 53.11 | 2.37 | **1.07** |
| CryptoMiniSat-INDU | 0.94 | 0.99 | 0.94 | 0.9 | 0.91 | **0.72** | **3.65** | 7.9 | 5.37 | 45.82 | 5.03 | 4.14 |
| CryptoMiniSat-SWV-IBM | 0.77 | 0.85 | 0.66 | 0.83 | 0.62 | **0.48** | 3.5 | 10.83 | 4.49 | 48.99 | 4.75 | **2.78** |
| CryptoMiniSat-IBM | 0.65 | 0.96 | 0.55 | 0.56 | 0.53 | **0.41** | 3.19 | 4.86 | 2.59 | 44.9 | 2.41 | **1.49** |
| CryptoMiniSat-SWV | 0.76 | 0.78 | 0.71 | 0.66 | 0.63 | **0.51** | 3.09 | 4.62 | 2.09 | 53.85 | 2.32 | **1.03** |
| SPEAR-INDU | 0.95 | 0.97 | 0.85 | 0.87 | 0.8 | **0.58** | **3.55** | 9.53 | 5.4 | 45.47 | 5.52 | 4.25 |
| SPEAR-SWV-IBM | 0.67 | 0.85 | 0.53 | 0.78 | 0.49 | **0.38** | 3.49 | 6.98 | 4.32 | 48.48 | 4.9 | **2.82** |
| SPEAR-IBM | 0.6 | 0.86 | 0.48 | 0.66 | 0.5 | **0.38** | 3.18 | 5.77 | 2.58 | 45.72 | 2.5 | **1.56** |
| SPEAR-SWV | 0.49 | 0.58 | 0.48 | 0.44 | **0.47** | **0.34** | 3.09 | 6.24 | 2.09 | 56.09 | 2.38 | **1.13** |
| TNM-RANDSAT | 1.01 | 1.05 | **0.94** | **0.93** | 1.22 | **0.88** | **3.79** | 8.63 | 6.57 | 46.21 | 7.64 | 5.42 |
| SAPS-RANDSAT | 0.94 | 1.09 | 0.73 | 0.78 | 0.86 | **0.66** | **3.81** | 8.54 | 6.62 | 49.33 | 6.59 | 5.04 |
| CPLEX-BIGMIX | 2.7E8 | 0.93 | 1.02 | 1 | 0.85 | **0.64** | **3.39** | 8.27 | 4.75 | 41.25 | 5.33 | **3.54** |
| Gurobi-BIGMIX | 1.51 | **1.23** | 1.41 | 1.26 | 1.43 | **1.17** | **3.35** | 5.12 | 4.55 | 40.72 | 5.45 | 3.69 |
| SCIP-BIGMIX | 4.5E6 | 0.88 | 0.86 | 0.91 | 0.72 | **0.57** | **3.43** | **5.35** | 4.48 | 39.51 | 5.08 | **3.75** |
| lp_solve-BIGMIX | 1.1 | 0.9 | 0.68 | 1.07 | 0.63 | **0.5** | 3.35 | 4.68 | 4.62 | 43.27 | **2.76** | 4.92 |
| CPLEX-CORLAT | 0.49 | 0.52 | 0.53 | **0.46** | 0.62 | **0.47** | **3.19** | 7.64 | 5.5 | 27.54 | 4.77 | **3.4** |
| Gurobi-CORLAT | **0.38** | 0.44 | 0.41 | **0.37** | 0.51 | **0.38** | **3.21** | 5.23 | 5.52 | 28.58 | 4.71 | **3.31** |
| SCIP-CORLAT | 0.39 | 0.41 | 0.42 | **0.37** | 0.5 | **0.38** | **3.2** | 7.96 | 5.52 | 26.89 | 5.12 | 3.52 |
| lp_solve-CORLAT | **0.44** | 0.48 | **0.44** | **0.45** | 0.54 | **0.41** | 3.25 | 5.06 | 5.49 | 31.5 | **2.63** | 4.42 |
| CPLEX-RCW | 0.25 | 0.29 | 0.1 | 0.03 | 0.05 | **0.02** | 3.11 | 7.53 | 5.25 | 25.84 | 4.81 | **2.66** |
| CPLEX-REG | **0.38** | **0.39** | 0.44 | **0.38** | 0.54 | 0.42 | **3.1** | 6.48 | 5.28 | 24.95 | 4.56 | 3.65 |
| CPLEX-CR | 0.46 | 0.58 | 0.46 | **0.43** | 0.58 | 0.45 | **4.25** | 11.86 | 11.19 | 29.92 | 11.44 | 8.35 |
| CPLEX-CRR | 0.44 | 0.54 | 0.42 | **0.37** | 0.47 | **0.36** | **5.4** | 18.43 | 17.34 | 35.3 | 20.36 | 13.19 |
| LK-H-PORTGEN | **0.61** | 0.63 | 0.64 | **0.61** | 0.89 | 0.67 | 4.14 | **1.14** | 12.78 | 22.95 | 11.49 | 11.14 |
| LK-H-PORTCGEN | **0.71** | 0.72 | 0.75 | **0.71** | 1.02 | 0.76 | 4.19 | **2.7** | 12.93 | 24.78 | 11.54 | 10.79 |
| LK-H-TSPLIB | 9.55 | **1.11** | 1.77 | **1.3** | **1.21** | **1.06** | 1.61 | 3.02 | 0.51 | 4.3 | 0.17 | **0.11** |
| Concorde-PORTGEN | **0.41** | 0.43 | 0.43 | **0.42** | 0.59 | 0.45 | 4.18 | **3.6** | 12.7 | 22.28 | 10.79 | 9.9 |
| Concorde-PORTCGEN | **0.33** | 0.34 | 0.34 | 0.34 | 0.46 | 0.35 | 4.17 | **2.32** | 12.68 | 24.8 | 11.16 | 10.18 |
| Concorde-TSPLIB | 120.6 | **0.69** | 0.99 | **0.87** | 0.64 | **0.52** | 1.54 | 2.66 | 0.47 | 4.26 | 0.22 | **0.12** |

**Table 6.2:** Quantitative comparison of models for runtime predictions on previously unseen instances. We report 10-fold cross-validation performance. Lower RMSE values are better (0 is optimal). Note the very large RMSE values for ridge regression on some data sets (we use scientific notation, denoting "$\times 10^x$" as "$Ex$"); these large errors are due to extremely small/large predictions for a few data points. Boldface indicates performance not statistically significantly different from the best method in each row.

| | Spearman rank correlation coefficient | | | | | | Log likelihood | |
|---|---|---|---|---|---|---|---|---|
| **Domain** | RR | SP | NN | PP | RT | RF | PP | RF |
| Minisat-COMPETITON | 0.69 | 0.57 | 0.86 | 0.79 | 0.83 | **0.9** | -4.78 | **−0.33** |
| Minisat-HAND | 0.69 | 0.59 | 0.87 | 0.81 | 0.84 | **0.91** | -2.65 | **−0.43** |
| Minisat-RAND | 0.79 | 0.74 | **0.82** | 0.8 | 0.78 | **0.83** | -1.12 | **−0.18** |
| Minisat-INDU | 0.7 | 0.66 | 0.85 | 0.79 | 0.87 | **0.92** | -5.72 | **−0.43** |
| Minisat-SWV-IBM | 0.95 | 0.89 | 0.97 | 0.96 | 0.98 | **0.99** | -6.64 | **0.12** |
| Minisat-IBM | 0.94 | 0.91 | 0.97 | 0.97 | 0.98 | **0.99** | -6.13 | **0.06** |
| Minisat-SWV | 0.94 | 0.95 | 0.97 | **0.98** | **0.99** | **0.99** | -4.83 | **0.2** |
| CryptoMiniSat-INDU | 0.66 | 0.59 | 0.72 | 0.71 | 0.76 | **0.81** | -5.99 | **−0.9** |
| CryptoMiniSat-SWV-IBM | 0.93 | 0.9 | 0.94 | 0.91 | 0.96 | **0.97** | -6.91 | **−0.37** |
| CryptoMiniSat-IBM | 0.93 | 0.85 | 0.94 | 0.94 | **0.96** | **0.97** | -5.8 | **−0.23** |
| CryptoMiniSat-SWV | 0.92 | 0.94 | **0.95** | 0.93 | **0.97** | **0.97** | -6.88 | **−0.59** |
| SPEAR-INDU | 0.63 | 0.62 | 0.78 | 0.75 | 0.82 | **0.88** | -6.66 | **−0.59** |
| SPEAR-SWV-IBM | 0.94 | 0.91 | 0.95 | 0.92 | 0.97 | **0.98** | -13.6 | **−0.22** |
| SPEAR-IBM | 0.95 | 0.87 | 0.96 | 0.93 | 0.96 | **0.98** | **−2.58** | **−0.18** |
| SPEAR-SWV | 0.95 | 0.93 | 0.94 | 0.95 | 0.96 | **0.97** | -7.33 | **−0.19** |
| TNM-RANDSAT | 0.87 | 0.86 | 0.9 | 0.89 | 0.83 | **0.91** | -4.65 | **−1.32** |
| SAPS-RANDSAT | 0.9 | 0.86 | 0.93 | 0.92 | 0.91 | **0.95** | -3.16 | **−0.79** |
| CPLEX-BIGMIX | 0.82 | 0.81 | 0.81 | 0.76 | 0.84 | **0.9** | -8.06 | **−0.7** |
| Gurobi-BIGMIX | **0.62** | **0.62** | 0.57 | 0.57 | 0.54 | **0.64** | -18.09 | **−2.36** |
| SCIP-BIGMIX | 0.81 | 0.76 | 0.81 | 0.73 | 0.84 | **0.89** | -7.33 | **−0.72** |
| lp_solve-BIGMIX | 0.34 | 0.31 | 0.35 | 0.22 | 0.47 | **0.6** | -13.22 | **−0.24** |
| CPLEX-CORLAT | 0.95 | 0.95 | 0.94 | **0.96** | 0.93 | **0.95** | -4.46 | **−0.53** |
| Gurobi-CORLAT | **0.95** | 0.93 | 0.94 | **0.95** | 0.92 | **0.95** | -3.12 | **−0.38** |
| SCIP-CORLAT | 0.94 | 0.94 | 0.93 | **0.95** | 0.91 | **0.94** | -5.04 | **−0.38** |
| lp_solve-CORLAT | 0.76 | 0.75 | 0.75 | 0.75 | **0.82** | 0.76 | -1.53 | **−0.25** |
| CPLEX-RCW | 0.94 | 0.92 | 0.99 | **1** | 1 | **1** | **2** | 0.23 |
| CPLEX-REG | **0.87** | **0.87** | 0.82 | **0.87** | 0.75 | 0.84 | -8.52 | **−0.59** |
| CPLEX-CR | 0.9 | 0.86 | 0.9 | **0.91** | 0.85 | 0.9 | -15.46 | **−0.54** |
| CPLEX-CRR | 0.89 | 0.85 | 0.9 | **0.92** | 0.88 | **0.92** | -20.04 | **−0.29** |
| LK-H-PORTGEN | **0.82** | 0.81 | 0.8 | 0.82 | 0.7 | 0.77 | -46.04 | **−1.16** |
| LK-H-PORTCGEN | **0.73** | 0.72 | 0.69 | **0.73** | 0.55 | 0.68 | -26.86 | **−1.25** |
| LK-H-TSPLIB | **0.64** | **0.8** | **0.55** | **0.71** | **0.76** | **0.75** | -3.78 | **−2** |
| Concorde-PORTGEN | **0.88** | 0.88 | 0.88 | **0.88** | 0.79 | 0.86 | -34.47 | **−0.66** |
| Concorde-PORTCGEN | **0.86** | 0.85 | 0.85 | 0.85 | 0.76 | 0.84 | -26.36 | **−0.36** |
| Concorde-TSPLIB | **0.73** | **0.86** | 0.72 | 0.67 | **0.86** | **0.91** | **−1.44** | **−1.1** |

**Table 6.3:** Quantitative comparison of models for runtime predictions on unseen instances. We report 10-fold cross-validation performance. Higher rank correlations are better (1 is optimal); log-likelihoods are only defined for models that yield a predictive distribution (here: PP and RF); higher values are better. Boldface indicates results not statistically significantly from the best.

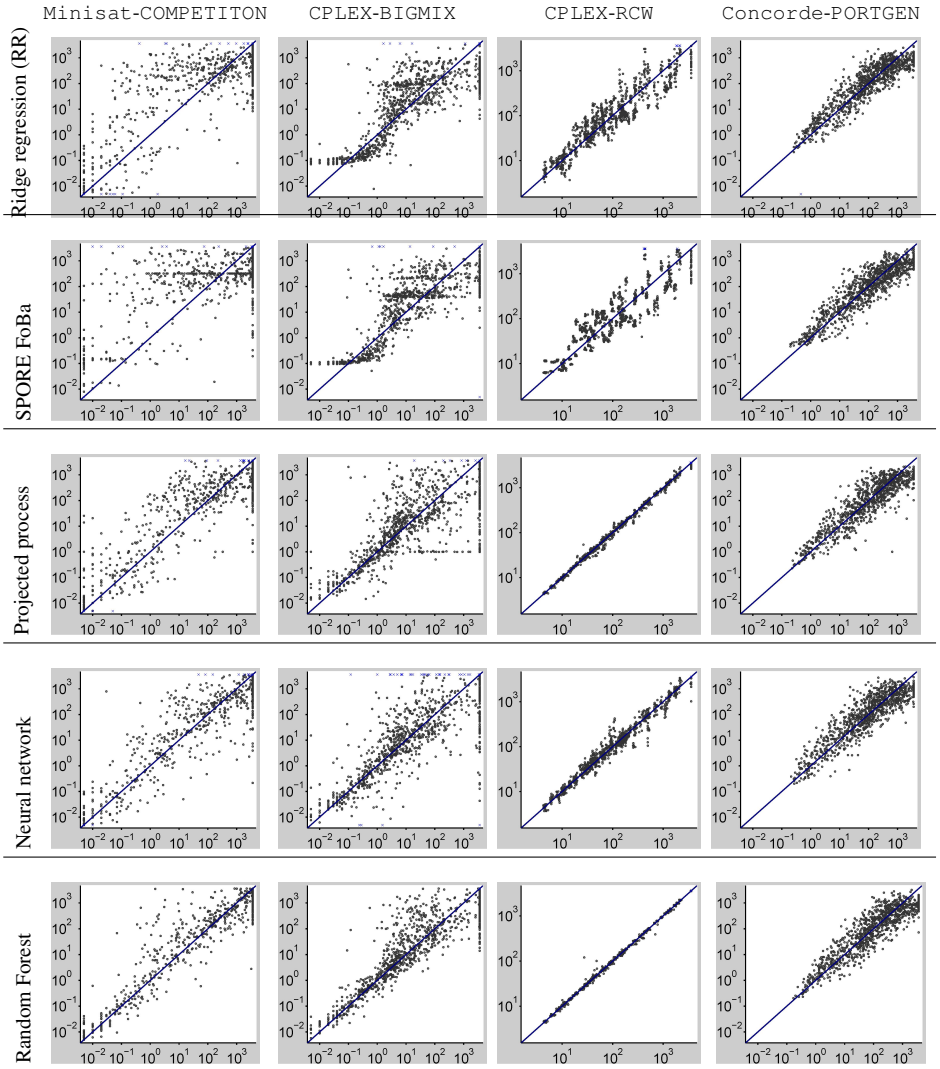



**Figure 6.1:** Visual comparison of models for runtime predictions on previously unseen test instances. The data sets used in each column are shown at the top. The $x$-axis of each scatter plot denotes true runtime and the $y$-axis 2-fold cross-validated runtime as predicted by the respective model; each dot represents one instance. Predictions above 3 000 or below 0.001 are denoted by a blue cross rather than a black dot. Plots for other benchmarks are qualitatively similar.

different methods, yielding the best predictions in terms of all our quantitative measures (refer to root mean squared error results in Table 6.2; correlation coefficients and log likelihoods results in Table 6.3). For SAT, they were always the best method, and for MIP they clearly yielded the best performance for the most heterogeneous instance set, BIGMIX (refer to Column 2 of Figure 6.1). We attribute the strong performance of random forests on highly heterogeneous data sets to a tree-based approach being able to model very different parts of the data separately, whereas methods that fit continuous functions typically allow the fit in one region to affect the fit in another. Indeed, all ridge regression variants posed problems with extremely-badly-predicted outliers for BIGMIX. For the other MIP datasets, either random forests or projected processes performed best, often followed closely by ridge regression variant RR. CPLEX's performance on set RCW was a special case that could be predicted extremely well across models (see Column 3 of Figure 6.1). Finally, for TSP, projected processes and ridge regression had a slight edge for the homogeneous PORTGEN and PORTCGEN benchmarks, whereas tree-based methods (once again) performed best on the most heterogeneous benchmark, TSPLIB. The last column of Figure 6.1 shows that in the case where random forests performed worst the qualitative differences in predictions were small. In terms of computational requirements, random forests were among the cheapest methods, taking between 0.1 and 11 seconds for model learning.

Since the number of instances for which performance data is available can be very limited in practical, we are interested in how the predictive quality of our models depends on the number of training instances. Figure 6.2 visualizes this scaling behaviour for six representative benchmarks (plots for other benchmarks are qualitatively similar). We show CC rather than RMSE, for two reasons. First, plots of RMSE are often cluttered due to poorly performing outliers (mostly of the ridge regression variants). Second, plotting CC allows immediate visual performance comparisons across benchmarks since CC $\in [-1,1]$.

Overall, random forests performed best across training set sizes. Interestingly, both versions of ridge regression (SP and RR) performed poorly for small training sets. This observation is significant since most past work employed ridge regression to construct empirical performance models in situations when data was sparse as in old versions of SATzilla, for example [210].

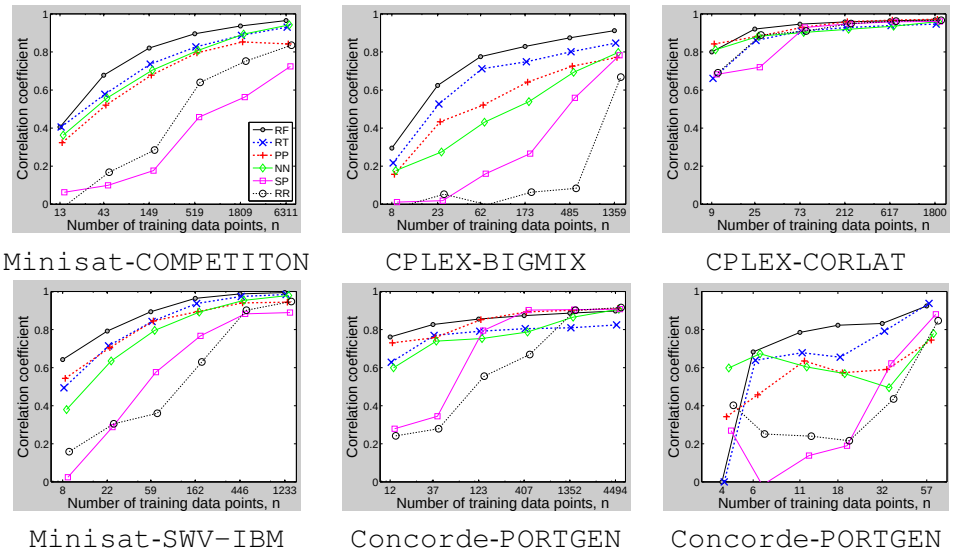



**Figure 6.2:** Prediction quality for varying numbers of training instances. For each model and number of training instances, we plot the mean (taken across 10 cross-validation folds) correlation coefficient (CC) between true and predicted runtimes for new test instances; larger CC is better, 1 is perfect. Plots for other benchmarks are qualitatively similar.

## 6.5 Conclusions

This chapter assessed and advanced the state of the art in predicting the performance of combinatorial algorithms. We proposed new techniques for building such predictive models and conducted the largest experimental study of which we are aware—predicting the performance of 11 algorithms on 35 instance distributions from SAT, MIP and TSP—comparing our new modeling approaches with all those previously used in the literature. We showed that our new approaches—chiefly those based on random forests, but also approximate Gaussian processes—offer the best performance, whether we consider predictions for unseen problem instances. We also demonstrated that very accurate predictions (correlation coefficients between predicted and true runtime exceeding 0.9) are possible based on very small amounts of training data (only hundreds of runtime observations). Overall, we showed that our methods are fast, general, and achieve good, robust performance; we hope they will be useful to a wide variety of researchers who seek to model al-

gorithm performance for algorithm analysis, scheduling, algorithm portfolio construction, automated algorithm configuration, and other applications.

# Chapter 7

# SATzilla: Portfolio-based Algorithm Selection for SAT

It has been widely observed that there is no single "dominant" SAT solver; instead, different solvers perform best on different instances. Rather than following the traditional approach of choosing the best solver for a given class of instances, we advocate making this decision online on a per-instance basis. In particular, this chapter describes SATzilla, an automated approach for constructing per-instance algorithm portfolios for solving SAT, that use machine learning techniques to choose among their constituent solvers. SATzilla takes as input a distribution of problem instances and a set of component solvers, and constructs a portfolio optimizing a given objective function (such as mean runtime, percent of instances solved, or score in a competition).

In addition to demonstrating the design and analysis of the first state-of-the-art portfolio-based algorithm selector on SAT, SATzilla07, this chapter goes well beyond it by making the portfolio construction scalable and completely automated, by improving it with local search solvers as candidate solvers, by predicting performance score instead of runtime, and by using hierarchical hardness models that consider different types of SAT instances. The effectiveness of these new techniques is demonstrated through extensive experimental results.

SATzilla remains an ongoing project. In 2009, a new procedure for prediction feature computation cost was added into SATzilla09 to improve SATzilla's

performance on the industrial category. In 2012, SATzilla2012 introduced a new selection procedure based on an explicit cost-sensitive loss function that punishes misclassifications in direct proportion to their impact on portfolio performance. The excellent performance of SATzilla on SAT was independently verified in the 2007/2009 SAT Competition and the 2012 SAT Challenge, where SATzilla solvers won more than 10 medals. [1]

## 7.1 Procedure of Building Portfolio based Algorithm Selection

The general methodology for building a portfolio based algorithm selector that we use in this work follows Leyton-Brown et al. (2003) in its broad strokes, but we have made significant extensions here. Portfolio construction transpires offline, as part of algorithm development, and comprises the following steps.

1. Identify a target distribution of problem instances. Practically, this means selecting a set of instances believed to be representative of some underlying distribution, or using an instance generator that constructs instances that represent samples from such a distribution.

2. Select a set of candidate solvers that have relatively uncorrelated runtimes on this distribution and are known or expected to perform well on at least some of the instances.

3. Identify features that characterize problem instances. In general this cannot be done automatically, but rather must reflect the knowledge of a domain expert. To be usable effectively for automated algorithm selection, these features must be related to instance hardness and be relatively cheap to compute.

4. On a training set of problem instances, compute these features and run each algorithm to determine its running times.

[1] This chapter is based on the joint work with Frank Hutter, Holger Hoos, and Kevin Leyton-Brown [210, 211, 216].

5. Identify one or more solvers to use for pre-solving instances. These pre-solvers will later be run for a short amount of time before features are computed (refer to step 9), in order to ensure good performance on very easy instances and to allow the empirical performance models to focus exclusively on harder instances.

6. Using a validation data set, determine which solver achieves the best performance for all instances that are not solved by the pre-solvers and on which the feature computation times out. We refer to this solver as the *backup solver*. In the absence of a sufficient number of instances for which pre-solving and feature computation timed out, we employed the single best component solver (i.e., the winner-take-all choice) as a backup solver.

7. Construct an empirical performance model for each algorithm in the portfolio, which predicts the runtime of the algorithm for each instance, based on the instance's features.

8. Choose the best subset of solvers to use in the final portfolio. We formalize and automatically solve this as a simple subset selection problem: from all given solvers, select a subset for which the respective portfolio (which uses the empirical performance models learned in the previous step) achieves the best performance on the validation set. (Observe that because our runtime predictions are not perfect, dropping a solver from the portfolio entirely can increase the portfolio's overall performance.)

Then, online, to solve a given problem instance, the following steps are performed.

9. Run each pre-solver until a predetermined fixed cutoff time is reached.

10. Compute feature values. If feature computation cannot be completed for some reason (error or timeout), select the backup solver identified in step 6.

11. Otherwise, predict each algorithm's runtime using the empirical performance models from step 7.

12. Run the algorithm predicted to be the best. If a solver fails to complete its run (e.g., it crashes), run the algorithm predicted to be next best.

Since 2009, we introduced an additional step prior to Step 7 that constructs a model for predicting the cost of feature computation based on some cheap features (e.g., number of variables, number of clauses). For a new test instance, before feature computation (Step 10), `SATzilla` first extracts the cheap features and predicts the cost of computing all features. If the predicted cost is higher than a given threshold, then `SATzilla` runs the backup solver instead of performing feature computation. In 2012, the empirical performance models were replaced by pairwise cost-sensitive classification models. Nevertheless, the described procedure is the basis of many advanced algorithm selectors.

## 7.2 Algorithm Selection Core: Predictive Models

The effectiveness of an algorithm selector depends on the ability to learn empirical performance models that can accurately predict a solver's performance on a given instance using efficiently computable features. In the experiments presented in this chapter, we use the same ridge regression method as in Chapter 5 that has previously proven to be very successful in predicting runtime on uniform random $k$-SAT, on structured SAT instances, and on combinatorial auction winner determination problems [92, 127, 156]. It should be noted that our portfolio methodology can make use of any regression/classification approach that provides sufficiently accurate estimates of an algorithm's performance and that is adequately computationally efficient where the time spent making a prediction can be compensated for by the performance gain obtained through improved algorithm selection. The detailed information on building predictive models has been described in previous chapters. In what follows, we introduce some special techniques that help to obtain more robust algorithm selectors.

### 7.2.1 Accounting for Censored Data

As is common with heuristic algorithms for solving NP-complete problems, SAT algorithms tend to solve some instances very quickly, while taking an extremely long amount of time to solve other instances. Hence, runtime data can be very

costly to gather, as individual runs can take literally weeks to complete, even when other runs on instances of the same size require only milliseconds. The common solution to this problem is to "censor" some runs by terminating them after a fixed cutoff time.

The question of how to fit good models in the presence of censored data has been extensively studied in the survival analysis literature in statistics, which originated in actuarial questions, such as estimating a person's lifespan given mortality data as well as the ages and characteristics of others who are still alive. Observe that this problem is the same as ours, except that in our case, data points are always censored at the same value, a subtlety that turns out not to matter.

The best approach that we know for dealing with censored data is to build models that use all available information about censored runs by using the censored runtimes as lower bounds on the actual runtimes. To our knowledge, this technique was first used in the context of SAT by Gagliolo and Schmidhuber (2006). This chapter chooses the simple, yet effective method by Schmee and Hahn (1979) to deal with censored samples. In brief, this method first trains a hardness model treating the cutoff time as the true (uncensored) runtime for censored samples, and then repeats the following steps until convergence.

1. Estimate the expected runtime of censored runs using the hardness model. Since in ridge regression, predictions are in fact normal distributions (with fixed variance), the expected runtime conditional on the runtime exceeding the cutoff time is the mean of the corresponding normal distribution truncated at the cutoff time.

2. Train a new hardness model using true runtimes for the uncensored instances and the predictions generated in the previous step for the censored instances.

We compared this approach with two other approaches to managing censored data: dropping such data entirely, and treating censored runs as though they finished at the cutoff threshold. The experimental results [209] demonstrated that both of these methods are significantly worse than the method presented above. Intuitively, both old methods introduce bias into empirical hardness models, whereas the method by Schmee and Hahn (1979) is unbiased.

### 7.2.2 Predicting Performance Score Instead of Runtime

The general portfolio methodology is based on empirical hardness models, which predict an algorithm's runtime. However, one may not simply be interested in using a portfolio to pick the solver with the lowest expected *runtime*. For example, in the 2007 SAT competition, solvers were evaluated based on a complex scoring function that depends only partly on a solver's runtime. Although the idiosyncracies of this scoring function are somewhat particular to the SAT competition, the idea that a portfolio should be built to optimize a performance score more complex than runtime has wide applicability. In this section we describe techniques for building models that predict such a *performance score* directly.

One critical issue is that—as long as one depends on standard supervised learning methods that require independent and identically distributed training data—one can only deal easily with scoring functions that actually associate a score with each single instance and combine the partial scores of all instances to compute the overall score. Given training data labeled with such a scoring function, `SATzilla` can simply learn a model of the score (rather than runtime) and then choose the solver with highest predicted score. Unfortunately, the scoring function used in the 2007 SAT Competition does not satisfy this independence property: the score a solver attains for solving a given instance depends in part on its (and, indeed, other solvers') performance on other, similar instances. More specifically, in the SAT competition each instance $P$ has a solution purse $SolutionP$ and a speed purse $SpeedP$; all instances in a given series (typically 5–40 similar instances) share one series purse $SeriesP$. Algorithms are ranked by summing three partial scores derived from these purses.

1. For each problem instance $P$, its solution purse is equally distributed between the solvers $S_i$ that solve the instance within the cutoff time (thereby rewarding robustness of a solver).

2. The speed purse for $P$ is divided among a set of solvers $S$ that solved the instance as $Score(P,S_i) = \frac{SpeedP \times SF(P,S_i)}{\Sigma_j SF(P,S_j)}$, where the speed factor $SF(P,S) = \frac{timeLimit(P)}{1+timeUsed(P,S)}$ is a measure of speed that discounts small absolute differences in runtime.

3. The series purse for each series is divided equally and distributed between the solvers $S_i$ that solved at least one instance in that series.

$S_i$'s partial score from problem $P$'s solution and speed purses solely depends on the solver's own runtime for $P$ and the runtime of all competing solvers for $P$. Thus, given the runtimes of all competing solvers as part of the training data, we can compute the score contributions from the solution and the speed purses of each instance $P$, and these two components are independent across instances. In contrast, since a solver's share of the series purse will depend on its performance on other instances in the series, its partial score received from the series purse for solving one instance is not independent of its performance on other instances.

Our solution to this problem is to approximate an instance's share of the series purse score by an independent score. If $N$ instances in a series are solved by any of `SATzilla`'s component solvers, and if $n$ solvers solve at least one of the instances in that series, we assign a partial score of $SeriesP/(N \times n)$ to each solver $S_i$ (where $i = 1, \ldots, n$) for each instance in the series it solved. This approximation of a non-independent score as independent is not always perfect, but it is conservative because it defines a lower-bound on the partial score from the series purse. Predicted scores will only be used in `SATzilla` to choose between different solvers on a per-instance basis. Thus, the partial score of a solver for an instance should reflect how much it would contribute to `SATzilla`'s score. If `SATzilla` were perfect (i.e., for each instance, it always selected the best algorithm) our score approximation would be correct: `SATzilla` would solve all $N$ instances from the series that any component solver can solve, and thus would actually achieve the series score $SeriesP/(N \times n) \times N = SeriesP/n$. If `SATzilla` performed very poorly and did not solve any instance in the series, our approximation would also be exact, since it would estimate the partial series score as zero. Finally, if `SATzilla` were to pick successful solvers for some (say, $M$) but not all instances of the series that could be solved by its component solvers (i.e., $M < N$), we would underestimate the partial series purse, since $SeriesP/(N \times n) \times M < SeriesP/n$.

While the learning techniques require an approximation of the performance score as an independent score, our empirical evaluation of solver scores employ the actual SAT competition scoring function. As explained previously, in the SAT com-

petition, the performance score of a solver depends on the score of all other solvers in the competition. In order to simulate a competition, we select a large number of solvers and pretend that these "reference solvers" and `SATzilla` are the only solvers in the competition. Throughout our analysis, we used the 19 solvers listed in Tables 7.3, 7.4 and 7.5. This is not a perfect simulation, since the scores change somewhat when different solvers are added to or removed from the competition. However, we obtained much better approximations of the performance score by following the methodology outlined here than by using cruder measures, such as learning models to predict mean runtime or the numbers of benchmark instances solved.

Finally, predicting performance score instead of runtime has a number of implications for the components of `SATzilla`. First, notice that one can compute an exact score for each algorithm and instance, even if the algorithm times out unsuccessfully or crashes—in these cases, the score from all three components is simply zero. When predicting scores instead of runtimes, we thus no longer need to rely on censored sampling techniques (see Section 7.2.1). Secondly, notice that the oracles for maximizing SAT competition score and for minimizing runtime are identical, since always using the solver with the smallest runtime guarantees that the highest values in all three components are obtained.

### 7.2.3 More General Hierarchical Performance Models

In this chapter, we consider a very heterogeneous instance distribution that consists of all instances from the categories `Random`, `Crafted` and `Industrial`. In order to further improve performance on this benchmark, we extend our previous hierarchical hardness model approach (predicting satisfiability status and then using a mixture of two conditional models as in Chapter 5) to the more general scenario of six underlying empirical hardness models (one for each combination of category and satisfiability status). The output of the general hierarchical model is a linear weighted combination of the output of each component. As described in Chapter 5, we can approximate the model selection oracle by a softmax function whose parameters are estimated using EM.

## 7.3 Portfolio Construction

In this section, we describe the procedure of constructing SATzilla solvers for the SAT Competition that features three main categories of instances, Random, Crafted (also known as Handmade) and Industrial. In order to study SATzilla's performance on an even more heterogeneous instance distribution, a third version of SATzilla is trained on data from all three categories of the competition; we label this new category ALL.

All of the SATzilla solvers were built using the design methodology detailed in Section 7.1. Each of the following subsections corresponds to one step from this methodology.

### 7.3.1 Selecting Instances

In order to train empirical performance models for any of the above scenarios, we needed instances that would be similar to those used in the real competition. For this purpose we used instances from the respective categories of the previous SAT competitions (2002, 2003, 2004, and 2005), as well as from the 2006 SAT Race (which only featured Industrial instances). Instances that were repeated in previous competitions were also repeated in our data sets. Overall, there were 4 811 instances: 2 300 instances in category Random, 1 490 in category Crafted and 1 021 in category Industrial; of course, category ALL included all of these instances. In addition to all instances prior to 2007, we also added the 869 instances from the 2007 SAT Competition into our four data sets. Overall, this resulted in 5 680 instances: 2 811 instances in category Random, 1 676 in category Crafted and 1 193 in category Industrial. Approximately 72% of the instances could be solved by at least one of the 19 solvers considered within the cutoff time of 1 200 CPU seconds on the reference machine; the remaining instances were excluded from our analysis.

We randomly split the above benchmark sets into training, validation and test sets, as described in Table 7.1. All parameter tuning and intermediate testing was performed on validation sets, and test sets were used only to generate the final results reported here.

We will be interested in analyzing SATzilla's performance as we vary the

| | "Old" instances before 2007 | "New" instances in 2007 |
|---|---|---|
| `Training` (40%) | $T_o$ (1 925 instances) | $T_n$ (347 instances) |
| `Validation` (30%) | $V_o$ (1 443 instances) | $V_n$ (261 instances) |
| `Test` (30%) | $E_o$ (1 443 instances) | $E_n$ (261 instances) |

**Table 7.1:** *Instances from before 2007 and from 2007 randomly split into training (T), validation (V) and test (E) data sets. These sets include instances for all categories:* `Random`, `Crafted` *and* `Industrial`.

| Data set | Training | Validation | Test |
|---|---|---|---|
| $D'$ | $T_o$ | $V_o$ | $E_o \cup E_n$ |
| $D^+$ | $T_o \cup T_n$ | $V_o \cup V_n$ | $E_o \cup E_n$ |

**Table 7.2:** *Data sets used in our experiments. Note that all data sets use identical test data, but different test data.*

data that was used to train it. To make it easy to refer to our different data sets, we describe them here and assign them names ($D'$, $D^+$). Table 7.1 shows the division of our data into "old" (pre-2007) and "new" (2007) instances. Table 7.2 shows how we combined this data to construct the two data sets we use for evaluation. Data set $D'$ uses only pre-2007 instances for training, validation, and both old and new instances for testing. Data set $D^+$ combines both old and new instances in its training, validation *and* test sets.

Since both data sets use the same test sets, the performance of portfolios trained using these different sets can be compared directly. However, we expect a portfolio trained using $D^+$ to be at least slightly better, because it has access to more data. For different categories, we use $D'_r$, $D'_h$, $D'_i$ to refer instances from `Random`, `Crafted`, `Industrial` (same for $D^+$).

### 7.3.2 Selecting Solvers

To decide what algorithms to include in our portfolio, we considered a wide variety of solvers that had been entered into previous SAT competitions and into the 2006 SAT Race. We manually analyzed the results of these competitions, identifying all algorithms that yielded the best performance on some subset of instances. Since

the instance sets contain both satisfiable and unsatisfiable instances, we considered a case study that did not chose any incomplete algorithms (the cost of misclassification would be very high if we choose a local search solver on an unsatisfiable instance). Ultimately, we selected the seven high-performance solvers shown in Table 7.3 as candidates for the SATzilla07 portfolio.

| Solver | Reference |
|---|---|
| Eureka | [152] |
| Kcnfs06 | [44] |
| March_dl04 | [75] |
| Minisat | [47] |
| Rsat 1.03 | [162] |
| Vallst | [201] |
| Zchaff_Rand | [139] |

**Table 7.3:** The seven solvers in SATzilla07; we refer to this set of solvers as S.

When we shift to predicting and optimizing performance score instead of runtime, local search solvers always obtain a score of exactly zero on unsatisfiable instances, since they are guaranteed not to solve them within the cutoff time. (Of course, they do not need to be run on an instance during training if the instance is known to be unsatisfiable.) Hence, we can build models for predicting the score of local search solvers using exactly the same methods as for complete solvers. Therefore, we considered eight new complete solvers (Tables 7.4) and four local search solvers (Tabl 7.5) from the 2007 SAT Competition for inclusion in our portfolio.

As with training instances, the sets of candidate solvers are treated as an input parameter of SATzilla, S. The sets of candidate solvers used in our experiments are detailed in Table 7.6.

### 7.3.3 Choosing Features

The choice of instance features has a significant impact on the performance of empirical performance models. Good features need to correlate well with (solver-specific) instance hardness and need to be cheap to compute, since feature computation time counts as part of SATzilla07's runtime.

| Solver | Reference |
|---|---|
| Kcnfs04 | [43] |
| TTS | [188] |
| Picosat | [19] |
| MXC | [25] |
| March_ks | [74] |
| TinisatElite | [87] |
| Minisat07 | [187] |
| Rsat 2.0 | [163] |

**Table 7.4:** Eight complete solvers from the 2007 SAT Competition.

| Solver | Reference |
|---|---|
| Ranov | [160] |
| Ag2wsat0 | [30] |
| Ag2wsat+ | [204] |
| Gnovelty+ | [161] |

**Table 7.5:** Four local search solvers from the 2007 SAT Competition.

| Name of Set | Solvers in the Set |
|---|---|
| S | all 7 solvers from Table 7.3 |
| $S^{+}$ | all 15 solvers from Tables 7.3 and 7.4 |
| $S^{++}$ | all 19 solvers from Tables 7.3, 7.4 and 7.5 |

**Table 7.6:** Solver sets used in our second series of experiments.

We used a subset of features from Figure 3.1. These features can be classified into nine categories: problem size, variable-clause graph, variable graph, clause graph, balance, proximity to Horn formulae, LP-based, DPLL probing and local search probing features. In order to limit the time spent computing features, we excluded a number of computationally expensive features, such as LP-based and clause graph features. The computation time for each of the local search and DPLL probing features was limited to 1 CPU second, and the total feature computation time per instance was limited to 60 CPU seconds. After eliminating some features that had the same value across all instances and some that were too unstable given

only 1 CPU second of local search probing, we ended up using 48 raw features.

### 7.3.4 Computing Features and Runtimes

All our experiments were performed using a computer cluster consisting of 55 machines with dual Intel Xeon 3.2GHz CPUs, 2MB cache and 2GB RAM, running Suse Linux 10.1. As in the SAT competition, all runs of any solver that exceeded a certain runtime were aborted (censored) and recorded as such. In order to keep the computational cost manageable, we chose a cutoff time of 1 200 CPU seconds.

### 7.3.5 Identifying Pre-solvers

As described in Section 7.1, in order to solve easy instances quickly without spending any time for the computation of features, we use one or more *pre-solvers*: algorithms that are run unconditionally but briefly before features are computed. Good algorithms for pre-solving solve a large proportion of instances quickly.

The naive approach for identifying pre-solvers is manual selection of pre-solvers based on an examination of the training runtime data. There are several limitations to this approach. First and foremost, manual pre-solver selection does not scale well. If there are many candidate solvers, manually finding the best combination of pre-solvers and cutoff times can be difficult and requires significant amounts of valuable human time. In addition, the manual pre-solver selection concentrates solely on solving a large number of instances quickly and does not take into account the pre-solvers' effect on model learning. In fact, there are three consequences of pre-solving.

1. Pre-solving solves some instances quickly before features are computed. In the context of the SAT competition, this improves `SATzilla`'s scores for easy problem instances due to the "speed purse" component of the SAT competition scoring function. (See Section 7.2.2 above.)

2. Pre-solving increases `SATzilla`'s runtime on instances not solved during pre-solving by adding the pre-solvers' time to every such instance. Like feature computation itself, this additional cost reduces `SATzilla`'s scores.

3. Pre-solving filters out easy instances, allowing our empirical performance

models to be trained exclusively on harder instances.

Manual selection considers (1) and (2), but not (3). In particular, it ignores the fact that the use of different pre-solvers and/or cutoff times results in different training data and hence in different learned models, which can also affect a portfolio's effectiveness.

The new automatic pre-solver selection technique functions as follows. We committed in advance to using a maximum of two pre-solvers: one of three complete search algorithms and one of three local search algorithms. The three candidates for each of the search approaches are automatically determined for each data set as those with highest score on the validation set when run for a maximum of 10 CPU seconds. We also use a number of possible cutoff times, namely 2, 5 and 10 CPU seconds, as well as 0 seconds (i.e., the pre-solver is not run at all) and consider both orders in which the two pre-solvers can be run. For each of the resulting 288 possible combinations of two pre-solvers and cutoff times, `SATzilla`'s performance on the validation data is evaluated by performing steps 6, 7 and 8 of the general methodology presented in Section 7.1:

6. determine the backup solver for selection when features time out;
7. construct an empirical performance model for each algorithm; and
8. automatically select the best subset of algorithms to use as part of `SATzilla`.

The best-performing subset found in this last step—evaluated on validation data—is selected as the algorithm portfolio for the given combination of pre-solver / cutoff time pairs. Overall, this method aims to choose the pre-solver configuration that yields the best-performing portfolio.

### 7.3.6 Identifying the Backup Solver

We computed average runtime of every solver on every category counting timeouts as runs that completed at the cutoff time of 1 200 CPU seconds. For categories `Random` and `Crafted`, we did not encounter instances for which feature computation timed out. Thus, we employed the winner-take-all solver as a backup solver in both of these domains. For categories `Industrial` and `ALL`, we chose the

solver that performed best on those instances that remained unsolved after pre-solving and for which feature computation timed out.

### 7.3.7 Learning Empirical Performance Models

We learned empirical performance models for predicting each solver's runtime/performance as described in Section 7.1, using the procedure of Schmee and Hahn (1979) for managing censored data and also employing hierarchical hardness models.

### 7.3.8 Solver Subset Selection

For a small number of candidate solvers, we performed automatic exhaustive subset search as outlined in Section 7.1 to determine which solvers to include in `SATzilla`. For a large number of component solvers, such a procedure is infeasible ($N$ component solvers would require the consideration of $2^N$ solver sets, for each of which a model would have to be trained). The automatic pre-solver selection methods described previously in Section 7.3.5 further worsen this situation: solver selection must be performed for every candidate configuration of pre-solvers, because new pre-solver configurations induce new models.

As an alternative to exhaustively considering all subsets, we implemented a randomized iterative improvement procedure to search for a good subset of solvers. The local search neighborhood used by this procedure consists of all subsets of solvers that can be reached by adding or dropping a single component solver. Starting with a randomly selected subset of solvers, in each search step, we consider a neighboring solver subset selected uniformly at random and accept it if validation set performance increases; otherwise, we accept the solver subset anyway with a probability of 5%. Once 100 steps have been performed with no improving step, a new run is started by re-initializing the search at random. After 10 such runs, the search is terminated and the best subset of solvers encountered during the search process is returned. Preliminary evidence suggests that this local search procedure is efficient in finding very good subsets of solvers.

| SATzilla version | Description |
|---|---|
| SATzilla07(S,D′) | Basic version for the 2007 SAT Competition, but evaluated on an extended test set. |
| SATzilla07($S^+$,$D^+$) | The same design as SATzilla07(S,D′), but includes new complete solvers (Table 7.4) and new data (Section 7.3.1). |
| SATzilla07$^+$($S^{++}$,$D^+$) | In addition to new complete solvers and data, this version uses local search solvers (Table 7.5) and all of the new design elements except "more general hierarchical hardness models" (Section 7.2.3). |
| SATzilla07*($S^{++}$,$D^+$) | This version uses all solvers, all data and all new design elements. Unlike for the other versions, we trained only one variant of this solver for use in all data set categories. |

**Table 7.7:** The different SATzilla versions evaluated in our second set of experiments.

### 7.3.9 Different SATzilla Versions

With new design ideas for SATzilla (Section 7.3.5, 7.3.8, 7.2.3, 7.2), new training data (Section 7.3.1) and new solvers (Section 7.3.2), we were interested in evaluating how much our portfolio improved as a result. In order to gain insights into how much performance improvement was achieved by these different changes, we studied several intermediate SATzilla solvers, which are summarized in Table 7.7.

SATzilla07(S,D') used manual *pre-solver* selection, exhaustive search for solver subset selection, and EHMs for predicting runtime. It only considered "old" solvers and training data. The construction of SATzilla07($S^+$,$D^+$) was the same as that for SATzilla07(S,D'), except that it relied on different solvers and corresponding training data.

SATzilla07$^+$($S^{++}$,$D^+$) and SATzilla07*($S^{++}$,$D^+$) incorporated the new techniques. Pre-solvers were identified automatically as described in Section 7.3.5, using the (automatically determined) candidate solvers listed in Table 7.8. We built models to predict the performance score of each algorithm. This score is well defined even in case of timeouts and crashes; thus, there was no need to deal

| | Random | Crafted | Industrial | ALL |
|---|---|---|---|---|
| **Complete** | Kcnfs06 | March_dl04 | Rsat 1.03 | Minisat07 |
| **Pre-solver** | March_dl04 | Vallst | Picosat | March_ks |
| **Candidates** | March_ks | March_ks | Rsat 2.0 | March_dl04 |
| **Local Search** | Ag2wsat0 | Ag2wsat0 | Ag2wsat0 | SAPS |
| **Pre-solver** | Gnovelty+ | Ag2wsat+ | Ag2wsat+ | Ag2wsat0 |
| **Candidates** | SAPS | Gnovelty+ | Gnovelty+ | Gnovelty+ |

**Table 7.8:** Pre-solver candidates for our four data sets. These candidates were automatically chosen based on the scores on validation data achieved by running the respective algorithms for a maximum of 10 CPU seconds.

with censored data. In the manner of SATzilla07, SATzilla07$^+$ used hierarchical empirical hardness models [208] with two underlying models ($M_{sat}$ and $M_{unsat}$) for predicting a solver's score. For SATzilla07$^*$, we built more general hierarchical hardness models for predicting scores; these models were based on six underlying empirical hardness models ($M_{sat}$ and $M_{unsat}$ trained on data from each SAT competition category). We chose solver subsets based on the results of the local search procedure for subset search as outlined in Section 7.3.8.

Observe that all of these solvers were built using identical test data and were thus directly comparable. We generally expected each solver to outperform its predecessors in the list. The exception was SATzilla07$^*$(S$^{++}$,D$^+$): this last solver was designed to achieve good performance across a broader range of instances. Thus, we expected SATzilla07$^*$(S$^{++}$,D$^+$) to outperform the others on category ALL, but not to outperform SATzilla07$^+$(S$^{++}$,D$^+$) on the more specific categories. The resulting final components of SATzilla07, SATzilla07$^+$ and SATzilla07$^*$ for each category are described in detail in the following section.

## 7.4 Performance Analysis of SATzilla

The effectiveness of our new techniques was investigated by evaluating the four SATzilla versions (Table 7.7): SATzilla07(S,D′), SATzilla07(S$^+$,D$^+$), SATzilla07$^+$(S$^{++}$,D$^+$) and SATzilla07$^*$(S$^{++}$,D$^+$). To evaluate their

| SATzilla version | Pre-Solvers (time) | Component solvers |
|---|---|---|
| SATzilla07(S,$D'_r$) | March_dl04(5); SAPS(2) | Kcnfs06, March_dl04, Rsat 1.03 |
| SATzilla07(S$^+$,$D^+_r$) | March_dl04(5); SAPS(2) | Kcnfs06, March_dl04, March_ks, Minisat07 |
| SATzilla07$^+$(S$^{++}$,$D^+_r$) | SAPS(2); Kcnfs06(2) | Kcnfs06, March_ks, Minisat07, **Ranov**, **Ag2wsat+**, **Gnovelty+** |

**Table 7.9:** *SATzilla's configurations for the Random category; cutoff times for pre-solvers are specified in CPU seconds.*

performance, we constructed a simulated SAT competition using the same scoring function as in the 2007 SAT Competition, but differing in a number of important aspects. The participants in our competition were the 19 solvers listed in Tables 7.3, 7.4, and 7.5 (all solvers were considered for all categories), and the test instances were $E_o \cup E_n$ as described in Tables 7.1 and 7.2. Furthermore, our computational infrastructure differed from the 2007 competition, and we also used shorter cutoff times of 1200 seconds. For these reasons some solvers ranked slightly differently in our simulated competition than in the 2007 competition.

### 7.4.1 Random Category

Table 7.9 shows the configuration of the three SATzilla versions for the Random category. Note that the automatic solver selection in SATzilla07$^+$(S$^{++}$,$D^+_r$) included different solvers than the ones used in SATzilla07(S$^+$,$D^+_r$); in particular, it chose three local search solvers, **Ranov**, **Ag2wsat+**, and **Gnovelty+**, that were not available to SATzilla07. Also, the automatic pre-solver selection chose a different order and cutoff time of pre-solvers than our manual selection: it chose to first run SAPS for two CPU seconds, followed by two CPU seconds of Kcnfs06. Even though running the local search algorithm SAPS did not help for solving unsatisfiable instances, we see in Figure 7.1 (left) that SAPS solved many more instances than March_dl04 in the first few seconds.

Table 7.10 shows the performance of different versions of SATzilla com-

pared to the best solvers in the Random category.

All versions of SATzilla outperformed every non-portfolio solver in terms of average runtime and number of instances solved. SATzilla07$^+$ and SATzilla07$^*$, the variants optimizing score rather than another objective function, also clearly achieved higher scores than the non-portfolio solvers. This was not always the case for the other versions; for example, SATzilla07(S$^+$,D$_r^+$) achieved only 86.6% of the score of the best solver, Gnovelty+ (where scores were computed based on a reference set of 20 reference solvers: the 19 solvers from Tables 7.3, 7.4, and 7.5, as well as SATzilla07(S$^+$,D$_r^+$)). Table 7.10 and Figure 7.1 show that adding complete solvers and training data did not greatly improve SATzilla07. At the same time, substantial improvements were achieved by the new mechanisms in SATzilla07$^+$, leading to 11% more instances solved, a reduction of average runtime by more than half, and an increase in score of over 50%. Interestingly, the performance of the more general SATzilla07$^*$(S$^{++}$,D$^+$) trained on instance mix ALL and tested on the Random category was quite close to the best version of SATzilla specifically designed for Random instances, SATzilla07$^+$(S$^{++}$,D$_r^+$). Note that due to their excellent performance on satisfiable instances, the local search solvers in Table 7.10 (Gnovelty+ and Ag2wsat variants) tended to have higher overall scores than the complete solvers (Kcnfs04 and March_ks), even though they solved fewer instances and in particular could not solve any unsatisfiable instance. In the 2007 SAT Competition, however, all winners of the random SAT+UNSAT category were complete solvers, which led to speculation that local search solvers were not considered in this category (while in the Random SAT category, all winners were indeed local search solvers).

Figure 7.1 presents CDFs summarizing the performance of the best non-portfolio solvers, SATzilla solvers and two oracles. All non-portfolio solvers omitted had CDFs below those shown. The oracles represent ideal versions of SATzilla that choose among component solvers perfectly and without any computational cost. More specifically, given an instance, an oracle picks the fastest algorithm; it is allowed to consider SAPS (with a maximum runtime of 10 CPU seconds) and any solver from the given set (S for one oracle and S$^{++}$ for the other).

Table 7.11 indicates how often each component solver of SATzilla07$^+$(S$^{++}$,D$_r^+$) was selected, how often it was successful, and the amount of its average runtime.

| Solver | Avg. runtime [s] | Solved [%] | Performance score |
|---|---|---|---|
| Kcnfs04 | 852 | 32.1 | 38309 |
| March_ks | **351** | **78.4** | 113666 |
| Ag2wsat0 | 479 | 62.0 | 119919 |
| Ag2wsat+ | 510 | 59.1 | 110218 |
| Gnovelty+ | 410 | 67.4 | **131703** |
| SATzilla07(S, $D'_r$) | 231 | 85.4 | — (86.6%) |
| SATzilla07($S^+$, $D^+_r$) | 218 | 86.5 | — (88.7%) |
| SATzilla07$^+$($S^{++}$, $D^+_r$) | **84** | **97.8** | **189436 (143.8%)** |
| SATzilla07*($S^{++}$, $D^+$) | 113 | 95.8 | — (137.8%) |

**Table 7.10:** *The performance of SATzilla compared to the best solvers on Random. The cutoff time was 1 200 CPU seconds; SATzilla07* ($S^{++}$, $D^+$) was trained on ALL. Scores were computed based on 20 reference solvers: the 19 solvers from Tables 7.3, 7.4, and 7.5, as well as one version of SATzilla. To compute the score for each non-SATzilla solver, the SATzilla version used as a member of the set of reference solvers was SATzilla07$^+$ ($S^{++}$, $D^+_r$). Since we did not include SATzilla versions other than SATzilla07$^+$ ($S^{++}$, $D^+_r$) in the set of reference solvers, scores for these solvers are incomparable to the other scores given here, and therefore, we do not report them. Instead, for each SATzilla solver, we indicate in parentheses its performance score as a percentage of the highest score achieved by a non-portfolio solver, given a reference set in which the appropriate SATzilla solver took the place of SATzilla07$^+$ ($S^{++}$, $D^+_r$).*

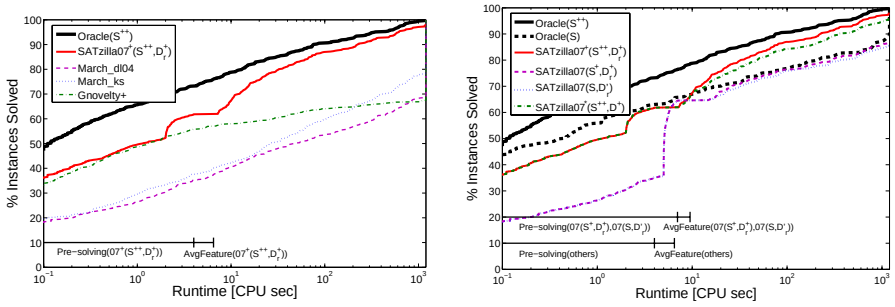



**Figure 7.1:** *Left: CDFs for SATzilla07$^+$ ($S^{++}$, $D^+_r$) and the best non-portfolio solvers on Random; right: CDFs for the different versions of SATzilla on Random shown in Table 7.9, where SATzilla07* ($S^{++}$, $D^+$) was trained on ALL. All other solvers' CDFs are below the ones shown here.*

| Pre-Solver (Pre-Time) | Solved [%] | | Avg. Runtime [CPU sec] |
|---|---|---|---|
| SAPS(2) | 52.2 | | 1.1 |
| March_dl04(2) | 9.6 | | 1.68 |
| **Selected Solver** | **Selected [%]** | **Success [%]** | **Avg. Runtime [CPU sec]** |
| March_dl04 | 34.8 | 96.2 | 294.8 |
| Gnovelty+ | 28.8 | 93.9 | 143.6 |
| March_ks | 23.9 | 92.6 | 213.3 |
| Minisat07 | 4.4 | 100 | 61.0 |
| Ranov | 4.0 | 100 | 6.9 |
| Ag2wsat+ | 4.0 | 77.8 | 357.9 |

**Table 7.11:** *The solvers selected by SATzilla07$^+$ ($S^{++}$, $D_r^+$) for the Random category. Note that column "Selected [%]" shows the percentage of instances remaining after pre-solving for which the algorithm was selected, and this sums to 100%. Cutoff times for pre-solvers are specified in CPU seconds.*

We found that the solvers picked by SATzilla07$^+$ ($S^{++}$, $D_r^+$) solved the given instance in most cases. Another interesting observation is that when a solver's success ratio was high, its average runtime tended to be lower.

### 7.4.2 Crafted Category

The configurations of the three SATzilla versions designed for the Crafted category are shown in Table 7.12. Again, SATzilla07$^+$ ($S^{++}$, $D_h^+$) included three local search solvers, **Ranov**, **Ag2wsat+** and **Gnovelty+**, which were not available to SATzilla07. Similar to the manual choice in SATzilla07, the automatic pre-solver selection chose to run March_dl04 for five CPU seconds. Unlike the manual selection, it abstained from using SAPS (or indeed any other solver) as a second pre-solver. Table 7.13 shows the performance of the different versions of SATzilla compared to the best solvers for category Crafted. Here, about half of the observed performance improvement was achieved by using more solvers and more training data; the other half was due to the improvements in SATzilla07$^+$. Note that for the Crafted category, SATzilla07$^*$ ($S^{++}$, $D^+$) performed quite poorly. We attribute this to a weakness of the feature-based classifier on Crafted instances, an issue discussed further in Section 7.4.4.

Table 7.14 indicates how often each component solver of SATzilla07$^+$ ($S^{++}$, $D_h^+$)

| SATzilla | Pre-Solver (time) | Components |
|---|---|---|
| SATzilla07(S,$D'_h$) | March_dl04(5); SAPS(2) | Kcnfs06, March_dl04, Minisat, Rsat 1.03 |
| SATzilla07($S^+$,$D^+_h$) | March_dl04(5); SAPS(2) | Vallst, Zchaff_rand, TTS, MXC, March_ks, Minisat07, Rsat 2.0 |
| SATzilla07$^+$($S^{++}$,$D^+_h$) | March_ks(5) | Eureka, March_dl04; Minisat, Rsat 1.03, Vallst, TTS, Picosat, MXC, March_ks, TinisatElite, Minisat07, Rsat 2.0, Ranov, Ag2wsat0, Gnovelty+ |

**Table 7.12:** *SATzilla's configurations for the Crafted category.*

| Solver | Avg. runtime [s] | Solved [%] | Performance score |
|---|---|---|---|
| TTS | 729 | 41.1 | 40669 |
| MXC | 527 | 61.9 | 43024 |
| March_ks | 494 | 63.9 | 68859 |
| Minisat07 | 438 | 68.9 | 59863 |
| March_dl04 | **408** | **72.4** | **73226** |
| SATzilla07(S,$D'_h$) | 284 | 80.4 | — (93.5%) |
| SATzilla07($S^+$,$D^+_h$) | 203 | 87.4 | — (118.8%) |
| SATzilla07$^+$($S^{++}$,$D^+_h$) | **131** | **95.6** | **112287 (153.3%)** |
| SATzilla07$^*$($S^{++}$,$D^+$) | 215 | 88.0 | — (110.5%) |

**Table 7.13:** *The performance of SATzilla compared to the best solvers on Crafted. Scores for non-portfolio solvers were computed using a reference set in which the only SATzilla solver was SATzilla07$^+$($S^{++}$,$D^+_h$). Cutoff time:* 1 200 *CPU seconds; SATzilla07$^*$($S^{++}$,$D^+$) was trained on ALL.*

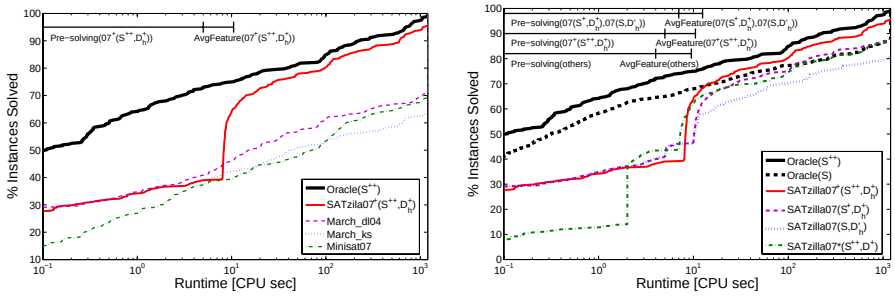



**Figure 7.2:** *Left: CDFs for* `SATzilla07+` $(S^{++}, D_h^+)$ *and the best non-portfolio solvers on* `Crafted`*; right: CDFs for the different versions of* `SATzilla` *on* `Crafted` *shown in Table 7.12, where* `SATzilla07*` $(S^{++}, D^+)$ *was trained on* `ALL`*. All other solvers' CDFs are below the ones shown here.*

| Pre-Solver (Pre-Time) | Solved [%] | | Avg. Runtime [CPU sec] |
|---|---|---|---|
| March_ks(5) | 39.0 | | 3.2 |
| **Selected Solver** | **Selected [%]** | **Success [%]** | **Avg. Runtime [CPU sec]** |
| Minisat07 | 40.4 | 89.3 | 205.1 |
| TTS | 11.5 | 91.7 | 133.2 |
| MXC | 7.2 | 93.3 | 310.5 |
| March_ks | 7.2 | 100 | 544.7 |
| Eureka | 5.8 | 100 | 0.34 |
| March_dl04 | 5.8 | 91.7 | 317.6 |
| Rsat 1.03 | 4.8 | 100 | 185.1 |
| Picosat | 3.9 | 100 | 1.7 |
| Ag2wsat0 | 3.4 | 100 | 0.5 |
| TinisatElite | 2.9 | 100 | 86.5 |
| Ranov | 2.9 | 83.3 | 206.1 |
| Minisat 2.0 | 1.4 | 66.7 | 796.5 |
| Rsat 2.0 | 1.4 | 100 | 0.9 |
| Gnovelty+ | 1.0 | 100 | 3.2 |
| Vallst | 0.5 | 100 | <0.01 |

**Table 7.14:** *The solvers selected by* `SATzilla07+` $(S^{++}, D_h^+)$ *for the* `Crafted` *category.*

| SATzilla | Pre-Solver (time) | Components |
|---|---|---|
| SATzilla07(S,D'$_i$) | Rsat 1.03 (2) | Eureka, March_dl04, Minisat, Rsat 1.03 |
| SATzilla07(S$^+$,D$_i^+$) | Rsat 2.0 (2) | Eureka, March_dl04, Minisat, Zchaff_Rand, TTS, Picosat, March_ks |
| SATzilla07$^+$(S$^{++}$,D$_i^+$) | Rsat 2.0 (10); Gnovelty+(2) | Eureka, March_dl04, Minisat, Rsat 1.03, TTS, Picosat, Minisat07, Rsat 2.0 |

**Table 7.15:** *SATzilla's configuration for the* `Industrial` *category.*

was selected, how many problem instances it solved, and its average runtime for these runs. There are many solvers that SATzilla07$^+$(S$^{++}$,D$_h^+$) picked quite rarely; however, in most cases, their success ratios are close to 100%, and their average runtimes are very low.

### 7.4.3 `Industrial` Category

Table 7.15 shows the configuration of the three SATzilla versions designed for the Industrial category. Local search solvers performed poorly for the instances in this category, with the best local search solver, Ag2wsat0, only solving 23% of the instances within the cutoff time. Consequently, no local search solver was selected by the automatic solver subset selection in SATzilla07$^+$(S$^{++}$,D$_i^+$). However, automatic pre-solver selection did include the local search solver Gnovelty+ as the second pre-solver, to be run for 2 CPU seconds after 10 CPU seconds of running Rsat 2.0.

Table 7.16 compares the performance of different versions of SATzilla and the best solvers on Industrial instances. It is not surprising that more training data and more solvers helped SATzilla07 to improve in terms of all our metrics (avg. runtime, percentage solved and score). A bigger improvement was due to the new mechanisms in SATzilla07$^+$ that led to SATzilla07$^+$(S$^{++}$,D$_i^+$) outperforming every non-portfolio solver with respect to every metric, particularly in terms of performance score. Note that the general SATzilla version

| Solver | Avg. runtime [s] | Solved [%] | Performance score |
|---|---|---|---|
| Rsat 1.03 | 353 | 80.8 | 52740 |
| Rsat 2.0 | 365 | 80.8 | 51299 |
| Picosat | **282** | **85.9** | 66561 |
| TinisatElite | 452 | 70.8 | 40867 |
| Minisat07 | 372 | 76.6 | 60002 |
| Eureka | 349 | 83.2 | **71505** |
| SATzilla07(S,$D'_i$) | 298 | 87.6 | — (91.3%) |
| SATzilla07(S$^+$,D$_i^+$) | 262 | 89.0 | — (98.2%) |
| SATzilla07$^+$(S$^{++}$,D$_i^+$) | **233** | **93.1** | **79724 (111.5%)** |
| SATzilla07$^*$(S$^{++}$,D$^+$) | 239 | 92.7 | — (104.8%) |

**Table 7.16:** *The performance of SATzilla compared to the best solvers on Industrial. Scores for non-portfolio solvers were computed using a reference set in which the only SATzilla solver was SATzilla07$^+$(S$^{++}$,D$_i^+$). Cutoff time: 1 200 CPU seconds; SATzilla07$^*$(S$^{++}$,D$^+$) was trained on ALL.*

SATzilla07$^*$(S$^{++}$,D$^+$) trained on ALL achieved performance very close to that of SATzilla07$^+$(S$^{++}$,D$_i^+$) on the Industrial data set in terms of average runtime and percentage of solved instances.

As can be seen from Figure 7.3, the performance improvements achieved by SATzilla over non-portfolio solvers were smaller for the Industrial category than for other categories. Note that the best Industrial solver, Picosat, performed very well, solving 85.9% of the instances within the cutoff time of 1 200 CPU seconds. Recall that this number means the solver solved 85.9% of the instances that could be solved by at least one solver. Compared to our other data sets, it would appear that either solvers exhibited more tightly correlated behavior on Industrial instances or that instances in this category exhibited greater variability in hardness. Nevertheless, SATzilla07$^+$(S$^{++}$,D$_i^+$) had significantly smaller average runtime (17%) and solved 7.2% more instances than the best component solver, Picosat. Likewise, the score for SATzilla07$^+$(S$^{++}$,D$_i^+$) was 11.5% higher than that of the top-ranking component solver (in terms of score), Eureka.

Table 7.17 indicates how often each component solver of SATzilla07$^+$(S$^{++}$,D$_i^+$) was selected, how many problem instances it solved, and its average runtime for

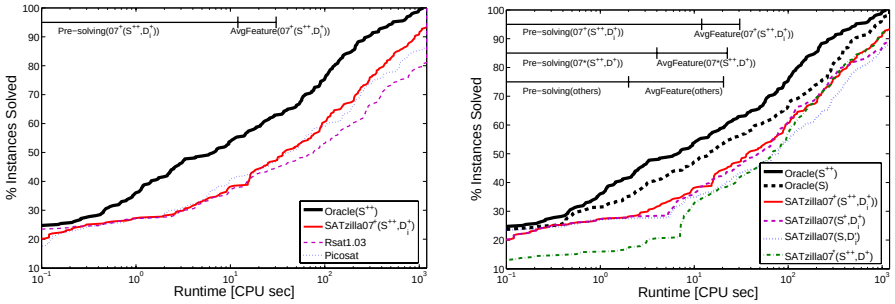



**Figure 7.3:** *Left: CDFs for* `SATzilla07+(S++,Di+)` *and the best non-portfolio solvers on* `Industrial`*; right: CDFs for the different versions of* `SATzilla` *on* `Industrial` *shown in Table 7.15, where* `SATzilla07*(S++,D+)` *was trained on* `ALL`*. All other solvers' CDFs (including* `Eureka`*'s) are below the ones shown here.*

| Pre-Solver (Pre-Time) | Solved [%] | | Avg. Runtime [CPU sec] |
|---|---|---|---|
| `Rsat 2.0`(10) | 38.1 | | 6.8 |
| `Gnovelty+` (2) | 0.3 | | 2.0 |
| **Selected Solver** | **Selected [%]** | **Success [%]** | **Avg. Runtime [CPU sec]** |
| `Eureka (BACKUP)` | 29.1 | 88.5 | 385.4 |
| `Eureka` | 15.1 | 100 | 394.2 |
| `Picosat` | 14.5 | 96.2 | 179.6 |
| `Minisat07` | 14.0 | 84.0 | 306.3 |
| `Minisat 2.0` | 12.3 | 68.2 | 709.2 |
| `March_dl04` | 8.4 | 86.7 | 180.8 |
| `TTS` | 3.9 | 100 | 0.7 |
| `Rsat 2.0` | 1.7 | 100 | 281.6 |
| `Rsat 1.03` | 1.1 | 100 | 10.6 |

**Table 7.17:** *The solvers selected by* `SATzilla07+(S++,Di+)` *for the* `Industrial` *category.*

these runs. In this case, the backup solver `Eureka` was used for problem instances for which feature computation timed out and pre-solvers did not produce a solution.

| SATzilla | Pre-Solver (time) | Components |
|---|---|---|
| SATzilla07(S,D') | March_dl04(5); SAPS(2) | Eureka, Kcnfs06, March_dl04, Minisat,Zchaff_rand |
| SATzilla07($S^+$,$D^+$) | March_dl04(5); SAPS(2) | Eureka, March_dl04, Zchaff_rand, Kcnfs04, TTS, Picosat, March_ks, Minisat07 |
| SATzilla07$^+$($S^{++}$,$D^+$) | SAPS(2); March_ks(2) | Eureka, Kcnfs06, Rsat 1.03, Zchaff_rand, TTS, MXC, TinisatElite, Rsat 2.0, Ag2wsat+, Ranov |
| SATzilla07$^*$($S^{++}$,$D^+$) | SAPS(2); March_ks(2) | Eureka, Kcnfs06, March_dl04, Minisat, Rsat 1.03, Picosat, MXC, March_ks, Minisat07, Ag2wsat+, Gnovelty+ |

**Table 7.18:** *SATzilla's configurations for the ALL category.*

### 7.4.4 ALL

There are four versions of SATzilla specialized for category ALL. Their detailed configurations are listed in Table 7.18. The results of automatic pre-solver selection were identical for SATzilla07$^+$ and SATzilla07$^*$: both chose to first run the local search solver SAPS for two CPU seconds, followed by two CPU seconds of March_ks. These solvers were similar to our manual selection, but their order was reversed. For solver subset selection, SATzilla07$^+$ and SATzilla07$^*$ yielded somewhat different results, but both of them kept two local search algorithms, Ag2wsat+ & Ranov, and Ag2wsat+ & Gnovelty+, respectively.

Table 7.19 compares the performance of the four versions of SATzilla on our ALL test set. Roughly equal improvements in terms of all our performance metrics were due to more training data and solvers on the one hand, and to the improvements in SATzilla07$^+$ on the other hand. The best performance in terms of all our performance metrics was obtained by SATzilla07$^*$($S^{++}$,$D^+$). Recall that the only difference between SATzilla07$^+$($S^{++}$,$D^+$) and SATzilla07$^*$($S^{++}$,$D^+$) was the use of more general hierarchical hardness models, as described in Sec-

| Solver | Avg. runtime [s] | Solved [%] | Performance score |
|---|---|---|---|
| Rsat 1.03 | 542 | 61.1 | 131399 |
| Kcnfs04 | 969 | 21.3 | 46695 |
| TTS | 939 | 22.6 | 74616 |
| Picosat | 571 | 57.7 | 135049 |
| March_ks | 509 | 62.9 | 202133 |
| TinisatElite | 690 | 47.3 | 93169 |
| Minisat07 | 528 | 61.8 | 162987 |
| Gnovelty+ | 684 | 43.9 | 156365 |
| March_dl04 | **509** | **62.7** | **205592** |
| SATzilla07(S,D') | 282 | 83.1 | — (125.0%) |
| SATzilla07($S^+$,$D^+$) | 224 | 87.0 | — (139.2%) |
| SATzilla07$^+$($S^{++}$,$D^+$) | 194 | 91.1 | — (158%) |
| SATzilla07$^*$($S^{++}$,$D^+$) | **172** | **92.9** | **344594 (167.6%)** |

**Table 7.19:** *The performance of SATzilla compared to the best solvers on ALL. Scores for non-portfolio solvers were computed using a reference set in which the only SATzilla solver was SATzilla07$^*$($S^{++}$,$D^+$). Cutoff time:* 1 200 *CPU seconds.*

tion 7.2.3.

Note that using a classifier is of course not as good as using an oracle for determining the distribution an instance comes from; thus, the success ratios of the solvers selected by SATzilla07$^*$ over the instances in the test set for distribution ALL (see Table 7.20) were slightly lower than those for the solvers picked by SATzilla07$^+$ for each of the distributions individually (see Tables 7.11, 7.14, and 7.17). However, when compared to SATzilla07$^+$ on distribution ALL, SATzilla07$^*$ performed significantly better: achieving overall performance improvements of 11.3% lower average runtime, 1.8% more solved instances and 9.6% higher score. This supports our initial hypothesis that SATzilla07$^*$ would perform slightly worse than specialized versions of SATzilla07$^+$ in each single category, yet would yield the best result when applied to a broader and more heterogeneous set of instances.

The runtime cumulative distribution function (Figure 7.4, right) shows that SATzilla07$^*$($S^{++}$,$D^+$) dominated the other versions of SATzilla on ALL and solved approximately 30% more instances than the best non-portfolio solver,

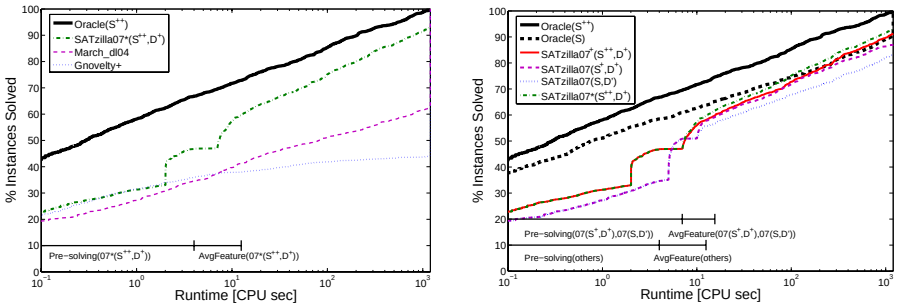



**Figure 7.4:** *Left: CDF for* `SATzilla07*(S++,D+)` *and the best non-portfolio solvers on* `ALL`*; right: CDFs for different versions of* `SATzilla` *on* `ALL` *shown in Table 7.18. All other solvers' CDFs are below the ones shown here.*

`March_dl04` (Figure 7.4, left).

| Pre-Solver (Pre-Time) | Solved [%] | | Avg. Runtime [CPU sec] |
|---|---|---|---|
| `SAPS`(2) | 33.0 | | 1.4 |
| `March_ks` (2) | 13.9 | | 1.6 |
| **Selected Solver** | **Selected [%]** | **Success [%]** | **Avg. Runtime [CPU sec]** |
| `Minisat07` | 21.2 | 85.5 | 247.5 |
| `March_dl04` | 14.5 | 84.0 | 389.5 |
| `Gnovelty+` | 12.5 | 85.2 | 273.2 |
| `March_ks` | 9.1 | 89.8 | 305.6 |
| `Eureka (BACKUP)` | 8.9 | 89.7 | 346.1 |
| `Eureka` | 7.2 | 97.9 | 234.6 |
| `Picosat` | 6.6 | 90.7 | 188.6 |
| `Kcnfs06` | 6.5 | 95.2 | 236.3 |
| `MXC` | 5.5 | 88.9 | 334.0 |
| `Rsat 1.03` | 4.0 | 80.8 | 364.9 |
| `Minisat 2.0` | 3.5 | 56.5 | 775.7 |
| `Ag2wsat+` | 0.5 | 33.3 | 815.7 |

**Table 7.20:** *The solvers selected by* `SATzilla07*(S++,D+)` *for the* `ALL` *category.*

Table 7.21 shows the performance of the general classifier in `SATzilla07*(S++,D+)`. We note several patterns: Firstly, classification performance for `Random` and `Industrial` instances was much better than for `Crafted` instances. Sec-

| | R, sat | R, unsat | H, sat | H, unsat | I, sat | I, unsat |
|---|---|---|---|---|---|---|
| classified R, sat | **92%** | 5% | 1% | – | 1% | 1% |
| classified R, unsat | 4% | **94%** | – | 1% | – | 1% |
| classified H, sat | – | – | **57%** | 38% | – | 5% |
| classified H, unsat | – | 1% | 23% | **71%** | 1% | 4% |
| classified I, sat | – | – | 8% | – | **81%** | 11% |
| classified I, unsat | – | – | – | 5% | 6% | **89%** |

**Table 7.21:** *Confusion matrix for the 6-way classifier on data set* `ALL`.

ondly, for `Crafted` instances, most misclassifications were not due to a misclassification of the instance type, but rather due to the satisfiability status. Finally, one can observe that `Random` instances were almost perfectly classified as `Random`, and only very few other instances were classified as `Random`, while `Crafted` and `Industrial` instances were confused somewhat more often. The comparably poor classification performance for `Crafted` instances partly explains why `SATzilla07`$^*$`(S`$^{++}$`,D`$^{+}$`)` did not perform as well for the `Crafted` category as for the others.

## 7.5 Further Improvements over the Years

Our group is actively working on introducing new techniques for improving `SATzilla`'s performance. Currently, `SATzilla` is still considered the state-of-the-art, even when compared to many new portfolio-building techniques.

### 7.5.1 SATzilla09 for `Industrial`

`SATzilla07` achieved less improvement in the domain of `Industrial`. One reason is that `Industrial` instances are often very large; the feature computation could be very costly and take a large proportion of the total CPU budget. The other reason is that the state-of-the-art solvers for solving `Industrial` are complete algorithms based on the DPLL procedure. To better handle `Industrial` instances, we introduced two new techniques in `SATzilla09`.

1. **New instance features.** After 2007, we introduced several new classes of instance features: 18 features based on clause learning, 18 based on survey

propagation, and 5 based on graph diameter. For the Industrial category, we also discarded 12 computationally expensive features based on DPLL probing and graph diameter.

2. **Prediction of feature computation time**. Before feature computation, we introduced an additional step that predicts the time required for feature computation. If that prediction exceeds two minutes, we run the backup solver; otherwise we continue with feature computation. In order to predict the feature computation time for an instance based on its number of variables and clauses, we built a simple linear regression model with quadratic basis functions. This was motivated by the fact that SATzilla's feature computation timed out on over 50% of the Industrial instances in the 2007 SAT competition and the 2008 SAT Race. By applying feature cost prediction, we force SATzilla to use a default solver on very large Industrial instances without paying the large cost of feature computation.

With the above two improvements and updated candidate solvers, training data, SATzilla09 performed very well on the 2009 SAT Competition. For the first time, SATzilla won a gold medal in the INDUSTRIAL category.

### 7.5.2 SATzilla2012 with New Algorithm Selector

Previous versions of SATzilla perform algorithm selection based on empirical performance models for predicting algorithm's performance. However, the goal of algorithm selection is to select solvers in order to optimize some performance objective. If multiple solvers have similar performance on instance $i$, selecting any of them does not reduce the performance of SATzilla. By contrast, if solvers have very different performance on instance $j$, then picking an incorrect solver for $j$ is more harmful than picking an incorrect solver for $i$. Therefore, the cost of misclassification depends on the performance difference among multiple candidate solvers. The new SATzilla2012 [216] is based on cost-sensitive classification models that punish misclassifications in direct proportion to their impact on portfolio performance. In addition, we also introduced a new procedure that generates a stand-alone SATzilla executable based on models learned within Matlab. These two improvements are described in detail in what follows.

1. **New algorithm selector.** Our new selection procedure is based on classification models with cost-sensitive loss function. To the best of our knowledge, this is the first time this approach has been applied to algorithm selection. We construct cost-sensitive decision forests (DFs) as collections of 99 cost-sensitive decision trees for every pair of algorithms. Each DF casts a vote for the better solver. The best solver is selected based on the number of votes received.

2. **New SATzilla executable.** To reduce the hassle of installing the free Matlab runtime environment (MRE) for making predictions based on models built with Matlab, we then converted our Matlab-built models to Java and provide Java code to make predictions using them. Thus, running `SATzilla2012` now only requires the scripting language Ruby (which is used for running the SATzilla pipeline).

`SATzilla2012` won first place in `Application`, `Hard Combinatorial`, and `Sequential Portfolio`; second place in `Application`, `Hard Combinatorial`, and `Random SAT`; third place in `Random SAT` (see [124] for detailed information).

## 7.6 Conclusions

Algorithms can be combined into portfolios to build a whole greater than the sum of its parts. We have significantly extended earlier work on algorithm portfolios for SAT that select solvers on a per-instance basis using empirical hardness models for runtime prediction. We have demonstrated the effectiveness of the portfolio construction method, `SATzilla07`, on four large sets of SAT competition instances. The experiments reveal that the `SATzilla07` portfolio solvers always outperformed their components. Furthermore, `SATzilla07`'s excellent performance in the 2007 SAT Competition demonstrates the practical effectiveness of our approach.

Following this work, we pushed the `SATzilla` approach further beyond `SATzilla07`. For the first time, we showed that portfolios can optimize complex scoring functions and integrate local search algorithms as component solvers. Furthermore, we

showed how to automate the process of pre-solver selection, one of the last aspects of our approach that was previously based on manual engineering. In 2012, we introduced a completely new algorithm selector based on cost-sensitive classification models that punished misclassifications in direct proportion to their impact on portfolio performance. As demonstrated in extensive computational experiments and the competition results, these enhancements improved SATzilla07's performance substantially.

SATzilla is now at a stage where it can be applied "out of the box" given a set of possible component solvers along with representative training and validation instances. In an automated built-in meta-optimization process, the component solvers to be used and the solvers to be used as pre-solvers are automatically determined from the given set of solvers, without any human effort. The computational bottleneck is to execute the possible component solvers on a representative set of instances in order to obtain adequate runtime data to build reasonably accurate empirical hardness models. However, these computations can be parallelized very easily and require no human intervention, only computer time, which becomes ever cheaper. The code for building empirical hardness models and SATzilla portfolios that use these models are available online at http://www.cs.ubc.ca/labs/beta/Projects/SATzilla.

SATzilla's performance ultimately depends on the power of all its component solvers and automatically gets better as they are improved. Furthermore, SATzilla takes advantage of solvers that are only competitive for certain kinds of instances and perform poorly otherwise, and thus SATzilla's success demonstrates the value of such solvers. Indeed, the identification of more such solvers, which are otherwise easily overlooked, still has the potential to further improve SATzilla's performance substantially.

# Chapter 8

# Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors

Having established SATzilla's effectiveness in 2007 and 2009, our team decided not to compete in the solver track of the 2011 competition, to avoid discouraging new work on (non-portfolio) solvers. Instead, we entered SATzilla in a new "analysis track", hoping other portfolio authors would do the same. However, other portfolio-based methods did feature prominently among the winners in every solver track: the algorithm selection and scheduling system 3S [112] and the simple, yet efficient parallel portfolio ppfolio [171] won a combined seven gold and 16 other medals (out of 18 categories overall).

Considering that portfolio-based solvers often achieve state-of-the-art performance, we believe that the community could benefit from rethinking how the value of individual solvers is measured. In this chapter, we demonstrate techniques for analyzing the extent to which state-of-the-art (*SOTA*) portfolio's performance depends on each of its component solvers. Such measures of solver contributions may also be applied to other portfolio approaches, including parallel portfolios.

We hope that this analysis serves as an encouragement to the community to focus on creative approaches that complement the strengths of existing solvers, even though they may (at least initially) be effective only on certain classes of instances. [1]

## 8.1 Measuring the Value of a Solver

One of the main reasons for holding a solver competition is to answer the question: *what is the current state of the art (SOTA)?* The traditional answer to this question has been *the winner of the respective category of the competition*; we call such a winner a *single best solver (SBS)*. However, as clearly demonstrated by the efficacy of algorithm portfolios, different solver strategies are (at least sometimes) complementary. This fact suggests a second answer to the SOTA question: *the virtual best solver (VBS),* defined as the best competition entry on a per-instance basis. The VBS typically achieves much better performance than the SBS, and does provide a useful theoretical upper bound on the performance currently achievable. However, this bound is typically not tight: the VBS is not an actual solver, because it only informs which underlying solver to run after the performance of each solver on a given instance has been measured, and thus the VBS cannot be (efficiently) run on new instances. Here, we propose a third answer to the SOTA question: *the best portfolio that can be constructed in a fully automated fashion from available solvers*; we call such a portfolio a *SOTA portfolio*. Since algorithm portfolios often substantially outperform their component solvers, SOTA portfolios can be expected to achieve better performance than SBS's; unlike the VBS, a SOTA portfolio is an executable algorithm that can be run on novel instances.

The most natural way of assessing the performance of a solver is by means of some statistic of its performance over a set (or distribution) of instances, such as the number of instances solved in a time budget, or its average runtime on an instance set. While there is value in these natural performance measures, we believe that they are not sufficient for capturing the *value a solver brings to the community*. Take, for example, two solvers MiniSAT′++ and NewSAT, where MiniSAT′++

[1] This chapter is based on the joint work with Frank Hutter, Holger Hoos, and Kevin Leyton-Brown [215].

is based on MiniSAT [46] and improves some of its components, while NewSAT is a (hypothetical) radically different solver that performs extremely well on a limited class of instances and poorly elsewhere. While solver MiniSAT′++ has a good chance to win medals in the SAT competition's Application track, solver NewSAT may not even be submitted, since (due to its poor average performance) it would be unlikely to even survive Phase 1 of the competition. However, MiniSAT′++ may only very slightly improve on the previous (MiniSAT-based) incumbent's performance, while NewSAT might represent deep new insights into the solution of instances that are intractable for all other known techniques.

The notion of state-of-the-art (SOTA) contributors [193] captures a solver's value to the community much more effectively than does average algorithm performance. The drawback is that it describes idealized solver contributions rather than contributions to an actual executable method. We propose instead measuring the SOTA contribution of a solver as its contribution to a SOTA portfolio that can be automatically constructed from the available solvers. This new notion resembles the prior notion of SOTA contributors, but directly quantifies their contributions to an *executable* portfolio solver, rather than to an abstract virtual best solver (VBS).

We must still describe exactly how we should assess a solver $A$'s contribution to a portfolio. One may measure the frequency with which the portfolio selects $A$, or the number of instances the portfolio solves using $A$. However, neither of these measures accounts for the fact that if $A$ were not available other solvers would be chosen instead, and might perform nearly as well. (Consider again Solver MiniSAT′++, and presume that it is chosen frequently by a portfolio. However, if it had not been created, the set of solved instances may be the same, and the portfolio's performance may be only slightly less.) We argue that a solver $A$ should be judged by its *marginal contribution* to the SOTA: the difference between the SOTA portfolio's performance including $A$ and the portfolio's performance excluding $A$. (Here, we measure portfolio performance as the percentage of instances solved since this is the main performance metric in the SAT competition.)

## 8.2 Experimental setup

**Solvers.** In order to evaluate the SOTA portfolio contributions of the SAT competition solvers, we constructed SATzilla portfolios using all sequential, non-portfolio solvers from Phase 2 of the 2011 SAT Competition as component solvers: 9, 15, and 18 candidate solvers for the Random, Crafted, and Application categories, respectively. (These solvers are listed in Table 8.2; see, e.g., [124] for their detailed information.) We hope that in the future, fully automated construction procedures will also be made publicly available for other portfolio builders, such as 3S [112]; if so, our analysis could be easily and automatically repeated for them. For each category, we also computed the performance of an *oracle* over sequential non-portfolio solvers (an idealized algorithm selector that picks the best solver for each instance) and the *virtual best solver* (*VBS*, an oracle over all 17, 25 and 31 entrants for the Random, Crafted and Application categories, respectively). These oracles do not represent the current state of the art in SAT solving, since they cannot be run on new instances; however, they serve as upper bounds on the performance that any portfolio-based selector over these solvers could achieve. We also compared to the performance of the winners of all three categories (including other portfolio-based solvers).

**Features.** We used a subset of 115 features from Figure 3.1. They fall into 9 categories: problem size, variable graph, clause graph, variable-clause graph, balance, proximity to Horn formula, local search probing, clause learning, and survey propagation. Feature computation averaged 31.4, 51.8 and 158.5 CPU seconds on Random, Crafted, and Application instances, respectively; this time counted as part of SATzilla's runtime budget.

**Methods.** We constructed SATzilla11 portfolios using the improved procedure described in Section 7.5.2. We set the feature computation cutoff $t_f = 500$ CPU seconds (a tenth of the time allocated to solve an instance). To demonstrate the effectiveness of our improvement, we also constructed a version of SATzilla09 (which uses ridge regression models), using the same training data.

We used 10-fold cross-validation to obtain an unbiased estimate of SATzilla's performance. First, we eliminated all instances that could not be solved by any can-

didate solver (we denote those instances as $U$). Then, we randomly partitioned the remaining instances (denoted $S$) into 10 disjoint sets. Treating each of these sets in turn as the test set, we constructed SATzilla using the union of the other 9 sets as training data, and measured SATzilla's runtime on the test set. Finally, we computed SATzilla's average performance across the 10 test sets.

To evaluate how important each solver was for SATzilla, for each category we quantified the marginal contribution of each candidate solver, as well as the percentage of instances solved by each solver during SATzilla's presolving (Pre1 or Pre2), backup, and main stages. Note that our use of cross-validation means that we constructed 10 different SATzilla portfolios using 10 different subsets ("folds") of instances. These 10 portfolios can be qualitatively different (e.g., selecting different presolvers); we report aggregates over the 10 folds.

**Data.** Runtime data was provided by the organizers of the 2011 SAT competition. All feature computations were performed by Daniel Le Berre on a quad-core computer with 4GB of RAM and running Linux, using our code. Four out of 1200 instances (from the Crafted category) had no feature values, due to a database problem caused by duplicated file names. We treated these instances as timeouts for SATzilla, thus obtaining a lower bound on SATzilla's true performance.

## 8.3 Experimental Results

We begin by assessing the performance of our SATzilla portfolios, to confirm that they did indeed yield SOTA performance. Table 8.1 compares SATzilla11 to the other solvers discussed above. In all categories SATzilla11 outperformed all of its component solvers. It also always outperformed SATzilla09, which in turn was slightly worse than the best component solver on Application.

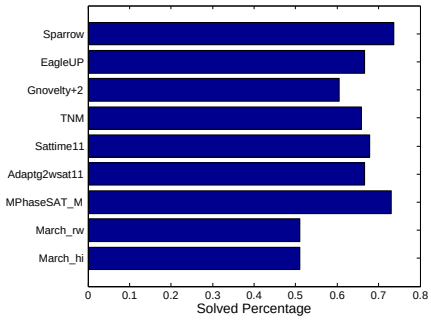
SATzilla11 also outperformed each category's gold medalist (including portfolio solvers such as 3S and ppfolio). Note that this does not constitute a fair comparison of the underlying portfolio construction procedures, as SATzilla had access to data and solvers unavailable to portfolios that competed in the solver track. This finding does, however, give us reason to believe that SATzilla portfolios either represent or at least closely approximate the best performance reachable by current methods. Indeed, in terms of instances solved, SATzilla11 reduced

| Solver | **Application** Runtime (Solved) | **Crafted** Runtime (Solved) | **Random** Runtime (Solved) |
|---|---|---|---|
| VBS | 1104 (84.7%) | 1542 (76.3%) | 1074 (82.2%) |
| Oracle | 1138 (84.3% ) | 1667 (73.7%) | 1087 (82.0%) |
| SATzilla11 | 1685 (75.3%) | 2096 (66.0%) | 1172 (80.8%) |
| SATzilla09 | 1905 (70.3%) | 2219(63.0%) | 1205 (80.3%) |
| Gold medalist | Glucose2: 1856 (71.7%) | 3S: 2602 (54.3%) | 3S: 1836 (68.0%) |
| Best comp. | Glucose2: 1856 (71.7%) | Clasp2: 2996 (49.7%) | Sparrow: 2066 (60.3%) |

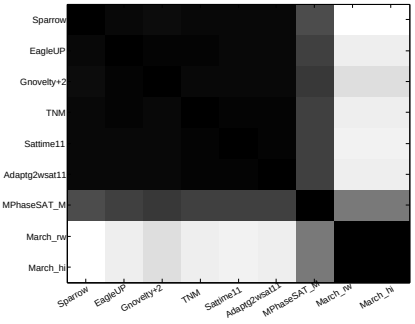
**Table 8.1:** *Comparison of SATzilla11 to the VBS, an Oracle over its component solvers, SATzilla09, the 2011 SAT competition winners, and the best single SATzilla11 component solver for each category. We counted timed-out runs as 5000 CPU seconds (the cutoff).*

the gap between the gold medalists and the (upper performance bound defined by the) VBS by 27.7% on Application, by 53.2% on Crafted and 90.1% on Random. The remainder of this chapter studies the contributions of each component solver to these portfolios. To substantiate our previous claim that marginal contribution is the most informative measure, here we contrast it with various other measures.

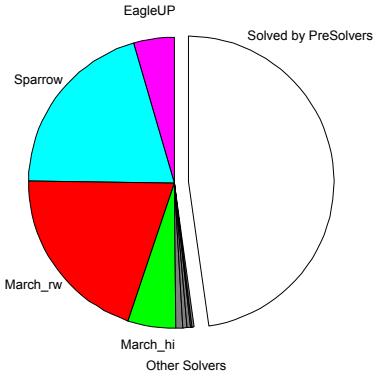
**Random.** Figure 8.1 presents a comprehensive visualization of our findings for the Random category; Table 8.2 (top) shows the underlying data. First, Figure 8.1a considers the set of instances that could be solved by at least one solver, and shows the percentage that each component solver is able to solve. By this measure, the two best solvers were Sparrow and MPhaseSAT_M. The former is a local search algorithm; it solved 362 + 0 satisfiable and unsatisfiable instances, respectively. The latter is a complete search algorithm; it solved 255 + 104 = 359 instances. Neither of these solvers won medals in the combined SAT + UNSAT Random category, as they were outperformed by portfolio solvers that combined both local and complete solvers. Figure 8.1b shows a correlation matrix of component solver performance: the entry for solver pair (A,B) is computed as the Spearman rank correlation coefficient between A's and B's runtime, with black and white representing perfect correlation and perfect independence respectively. Two clusters are apparent: six local search solvers (EagleUP, Sparrow, Gnovelty+2, Sattime11, Adaptg2wsat11, and TNM), and two versions of the complete solver March, which achieved almost identical performance. MPhaseSAT_M performed well on both satisfiable and unsatisfiable solvers; it was strongly correlated



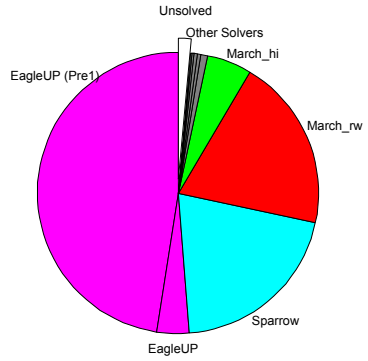

**(a)** Percentage of solvable instances solved by each component solver alone



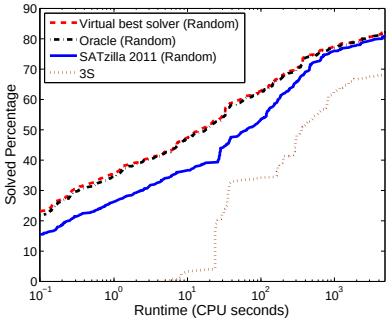

**(b)** Correlation matrix of solver runtimes (darker = more correlated)



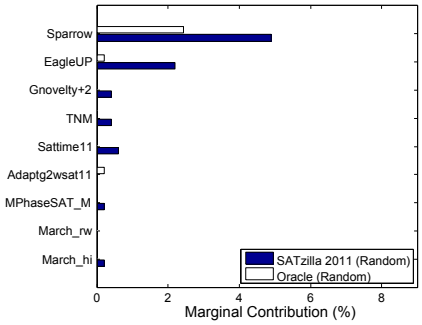

**(c)** Solver selection frequencies for `SATzilla11`; names only shown for solvers picked for $> 5\%$ of the instances




**(d)** Fraction of instances solved by `SATzilla11` pre-solver(s), backup solver, and main stage solvers




**(e)** Runtime CDF for `SATzilla11`, `VBS`, `Oracle`, and the gold medalist `3S`

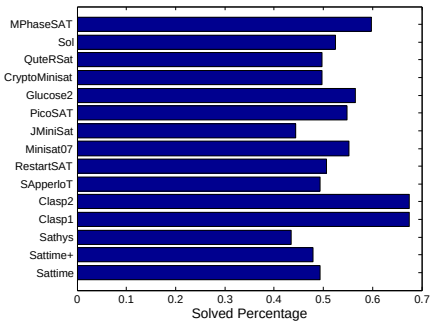



**(f)** Marginal contribution to `Oracle` and `SATzilla11`

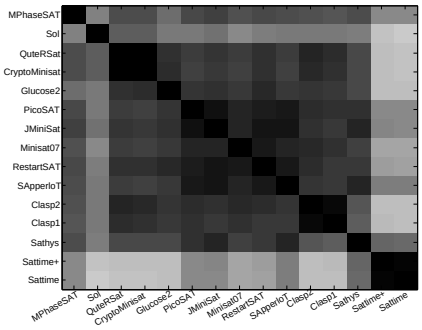
**Figure 8.1:** Visualization of results for category `Random`

with local search solvers on the satisfiable instance subset, and very strongly correlated with the March variants on the unsatisfiable subset. Figure 8.1c shows the frequency with which different solvers were selected in SATzilla11. The main solvers selected in SATzilla11's main phase were the best-performing local search solver Sparrow and the best-performing complete solver March. As shown in Figure 8.1d, the local search solver EagleUP was consistently chosen as a presolver and was responsible for more than half (51.3%) of the instances solved by SATzilla11 overall. We observe that MPhaseSAT_M did not play a large role in SATzilla11: it was only run for 2 out of 492 instances (0.4%). Although MPhaseSAT_M achieved very strong overall performance, its versatility appears to have come at the price of not excelling on either satisfiable or unsatisfiable instances, being largely dominated by local search solvers on the former and by March variants on the latter. Figure 8.1e shows that SATzilla11 closely approximated both the Oracle over its component solvers and the VBS, and stochastically dominated 3S, the gold medalist. Finally, Figure 8.1f shows the metric that we previously argued is the most important: each solver's marginal contribution to SATzilla11's performance. The most important portfolio contributor was Sparrow, with a marginal contribution of 4.9%, followed by EagleUP with a marginal contribution of 2.2%. EagleUP's low marginal contribution may be surprising at first glance (recall that it solved 51.3% of the instances SATzilla11 solved overall); however, 49.1% of these instances were also solvable by other local search solvers. Similarly, both March variants had very low marginal contributions (0% and 0.2%, respectively) since they were essentially interchangeable (correlation coefficient 0.9974). Further insight can be gained by examining the marginal contribution of *sets* of highly correlated solvers. The marginal contribution of the set of both March variants was 4.0% (MPhaseSAT_M could still solve most instances), while the marginal contribution of the set of six local search solvers was 22.5% (nearly one-third of the satisfiable instances were not solvable by any complete solver).

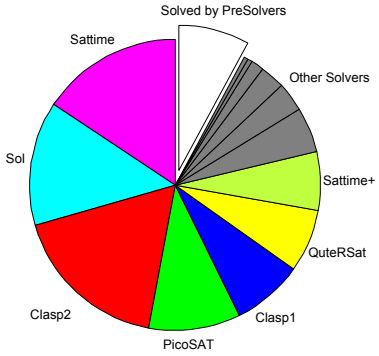
**Crafted.** Overall, sufficiently many solvers were relatively uncorrelated in the Crafted category (Figure 8.2) to yield a portfolio with many important contributors. The most important of these was Sol, which solved all of the 13.7% of the



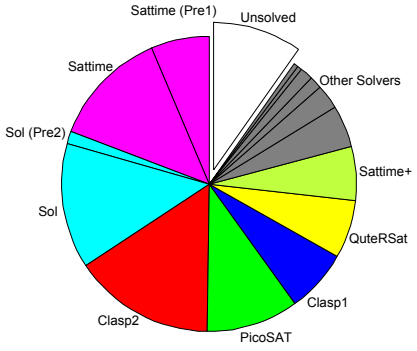

**(a)** Percent of solvable instances solved by each component solver alone



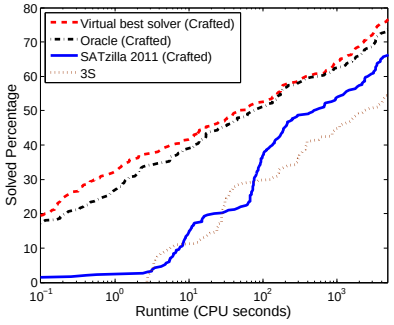

**(b)** Correlation matrix of solver runtimes (darker = more correlated)



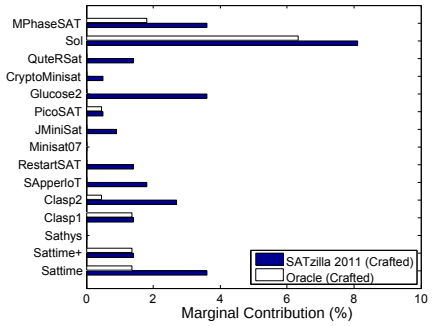

**(c)** Solver selection frequencies for `SATzilla11`; names only shown for solvers picked for $> 5\%$ of the instances




**(d)** Fraction of instances solved by `SATzilla11` pre-solver(s), backup solver, and main stage solvers




**(e)** Runtime CDF for `SATzilla11`, `VBS`, `Oracle`, and the gold medalist `3S`




**(f)** Marginal contribution for `Oracle` `SATzilla11`

**Figure 8.2:** Visualization of results for category `Crafted`

instances for which SATzilla11 selected it; without it, SATzilla11 would have solved 8.1% fewer instances. We observe that Sol was not identified as an important solver in the SAT competition results, ranking 11th of 24 solvers in the SAT+UNSAT category. Similarly, MPhaseSAT_M, Glucose2, and Sattime each solved a 3.6% fraction of instances that would have gone unsolved without them. (This is particularly noteworthy for MPhaseSAT_M, which was only selected for 5% of the instances in the first place.) Considering the marginal contributions of sets of highly correlated solvers, we observed that {Clasp1, Clasp2} was the most important at 6.3%, followed by {Sattime, Sattime11} at 5.4%. {QuteRSat, CryptoMiniSat} and {PicoSAT, JMiniSat, Minisat07, RestartSAT, SApperloT} were relatively unimportant even as sets, with marginal contributions of 0.5% and 1.8% respectively.

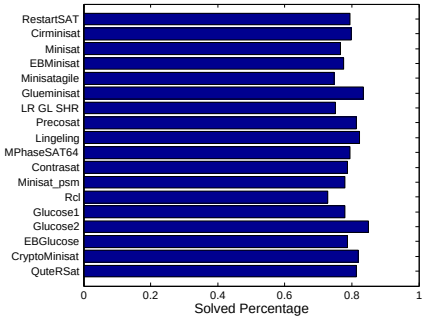
**Application.** All solvers in the Application category (Figure 8.3) exhibited rather highly correlated performance. It is thus not surprising that in 2011, no medals were awarded to portfolio solvers in the sequential Application track, and that in 2007 and 2009, SATzilla versions performed worst in this track, only winning a single gold medal in the 2009 satisfiable category. As mentioned earlier, SATzilla11 did outperform all competition solvers, but here the margin was only 3.6% (as compared to 12.8% and 11.7% for Random and Crafted, respectively). All solvers were rather strongly correlated and each solver could be replaced in SATzilla11 without a large decrease in performance; for example, dropping the competition winner only decreased SATzilla11's percentage of solved instances by 0.4%. The highest marginal contribution across all 18 solvers was four times larger: 1.6% for MPhaseSAT64. Similar to MPhaseSAT in the Crafted category, it was selected infrequently (only for 3.6% of the instances) but was the only solver able to solve about half of these instances. We conjecture that this was due to its unique phase selection mechanism. Both MPhaseSAT64 and Sol (in the Crafted category) thus come close to the hypothetical solver NewSAT mentioned earlier: they showed outstanding performance on certain instances and thus contributed substantially to a portfolio, while having achieved an unremarkable ranking in the competition (9th of 26 for MPhaseSAT64, 11th of 24 for Sol). We did observe one larger marginal contribution than that of

MPhaseSAT64 when dropping sets of solvers: 2.3% for {Glueminisat, LR GL SHR}. The other three highly correlated clusters also gave rise to relatively high marginal contribution: 1.5% for {CryptoMiniSat, QuteRSat}, 1.5% for {Glucose1,Glucose2,EBGlucose}, and 1.2% for {Minisat,EBMiniSAT, MiniSATagile}.

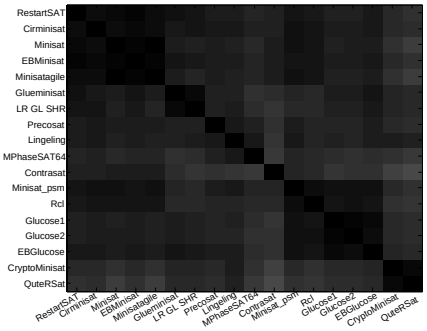
## 8.4 Conclusions

This chapter investigates the question of assessing the contributions of individual SAT solvers by examining their value to SATzilla, a portfolio-based algorithm selector. SATzilla11 is an improved version of this procedure based on cost-based decision forests, which entered into the new analysis track of the 2011 SAT competition. Its automatically generated portfolios achieved state of the art performance across all competition categories, and consistently outperformed both its constituent solvers, other competition entrants, and our previous version of SATzilla. The experimental results show that the selection frequency of a component solver is a poor measure of that solver's contribution to SATzilla11's performance. Instead, we advocate assessing solvers in terms of their marginal contributions to the state of the art in SAT solving.

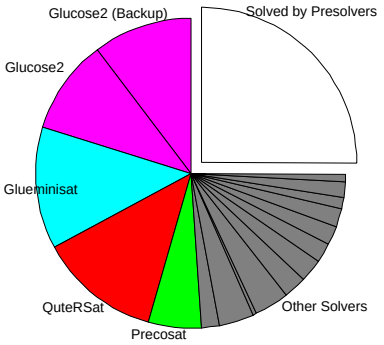
One main observation was that the solvers with the largest marginal contributions to SATzilla were often not competition winners (e.g., , MPhaseSAT64 in Application SAT+UNSAT; Sol in Crafted SAT+UNSAT). To encourage improvements to the state of the art in SAT solving and taking into account the practical effectiveness of portfolio-based approaches, we suggest rethinking the way future SAT competitions are conducted. In particular, we suggest that all solvers that solve some instances that no other solver can handle pass Phase 1 of the competition, and that solvers contributing most to the best-performing portfolio-based approaches be given formal recognition. We also recommend that portfolio-based solvers be evaluated separately—and with access to all submitted solvers as components—rather than competing with traditional solvers.



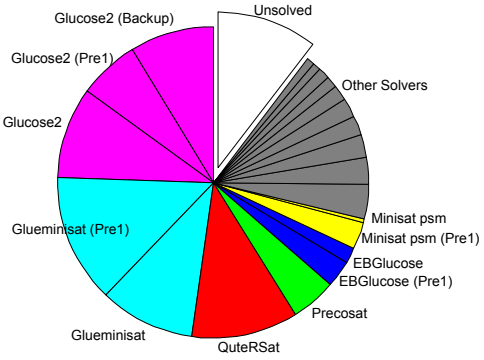

**(a)** Percent of solvable instances solved by each component solver alone



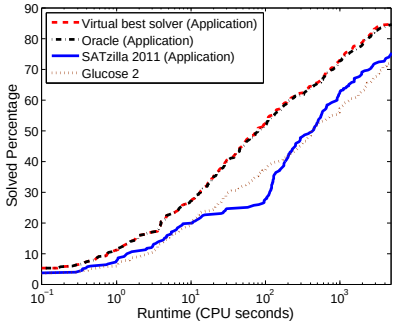

**(b)** Correlation matrix of solver runtimes (darker = more correlated)



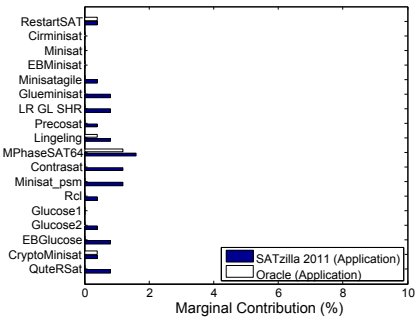

**(c)** Solver selection frequencies for `SATzilla11`; names only shown for solvers picked for $> 5\%$ of the instances




**(d)** Fraction of instances solved by `SATzilla11` pre-solver(s), backup solver, and main stage solvers




**(e)** Runtime CDF for `SATzilla11`, `VBS`, `Oracle`, and the gold medalist `Glucose2`




**(f)** Marginal contribution for `Oracle` and `SATzilla11`

**Figure 8.3:** Visualization of results for category `Application`

| | Solver | Indiv. Perform. Runtime (solved) | Average Correlation | Used as Backup (Solved) | Used as Pre (Solved) | Picked by Model (Solved) | Marg. contrib. SATzilla(Oracle) |
|---|---|---|---|---|---|---|---|
| | EagleUP | 1761 (66.7%) | 0.70 | - (-) | 10/10 (47.6%) | 4.5% (3.7%) | 2.2% (0.2%) |
| R | Sparrow | 1422 (73.6%) | 0.67 | - (-) | - (-) | 20.3% (20.3%) | 4.9% (2.4%) |
| A | March_rw | 2714 (51.0%) | 0.23 | - (-) | - (-) | 20.1% (19.9%) | 0.0% (0.0%) |
| N | March_hi | 2725 (51.0%) | 0.23 | - (-) | - (-) | 5.3% (5.1%) | 0.2% (0.0%) |
| D | Gnovelty+2 | 2146 (60.6%) | 0.71 | - (-) | - (-) | 0.8% (0.8%) | 0.4% (0.0%) |
| O | MPhaseSAT_M | 1510 (73.0%) | 0.67 | - (-) | - (-) | 0.4% (0.4%) | 0.2% (0.0%) |
| M | Sattime11 | 1850 (67.9%) | 0.70 | - (-) | - (-) | 0.4% (0.4%) | 0.6% (0.0%) |
| | Adaptg2wsat11 | 1847 (66.7%) | 0.70 | - (-) | - (-) | 0.4% (0.2%) | -0.4% (0.2%) |
| | TNM | 1938 (65.9%) | 0.70 | - (-) | - (-) | 0.2% (0.2%) | 0.4% (0.0%) |
| | Sattime | 2638 (49.3%) | 0.39 | - (-) | 2/10 (6.4%) | 15.5% (12.8%) | 3.6% (1.4%) |
| | Sol | 2563 (52.5%) | 0.48 | - (-) | 1/10 (1.4%) | 13.7% (13.7%) | 8.1% (6.4%) |
| | Clasp2 | 2280 (67.4%) | 0.69 | - (-) | - (-) | 17.4% (15.5%) | 2.7% (0.4%) |
| | PicoSAT | 2729 (54.8%) | 0.73 | - (-) | - (-) | 10.1% (10.1%) | 0.5% (0.4%) |
| C | Clasp1 | 2419 (67.4%) | 0.67 | - (-) | - (-) | 7.8% (6.9%) | 1.4% (1.4%) |
| R | QuteRSat | 2793 (49.8%) | 0.69 | - (-) | - (-) | 6.9% (6.4%) | 1.4% (0.0%) |
| A | Sattime+ | 2681 (48.0%) | 0.40 | - (-) | - (-) | 6.4% (5.9%) | 1.4% (1.4%) |
| F | MPhaseSAT | 2398 (59.7%) | 0.62 | - (-) | - (-) | 5.0% (4.6%) | 3.6% (1.8%) |
| T | CryptoMiniSat | 2766 (49.8%) | 0.68 | - (-) | - (-) | 3.2% (2.7%) | 0.5% (0.0%) |
| E | RestartSAT | 2773 (50.7%) | 0.73 | - (-) | - (-) | 2.7% (1.4%) | 1.4% (0.0%) |
| D | SApperloT | 2798 (49.3%) | 0.73 | - (-) | - (-) | 1.4% (1.4%) | 1.8% (0.0%) |
| | Glucose2 | 2644 (56.6%) | 0.66 | - (-) | - (-) | 0.5% (0.5%) | 3.6% (0.0%) |
| | JMiniSat | 3026 (44.3%) | 0.74 | - (-) | - (-) | 0.5% (0.5%) | 0.9% (0.0%) |
| | Minisat07 | 2738 (55.2%) | 0.70 | - (-) | - (-) | 0.0% (0.0%) | 0.0% (0.0%) |
| | Sathys | 2955 (43.4%) | 0.69 | - (-) | - (-) | 0.0% (0.0%) | 0.0% (0.0%) |
| | Glucose2 | 1272 (85.0%) | 0.86 | 10/10 (8.7%) | 3/10 (6.3%) | 9.9% (9.5%) | 0.4% (0.0%) |
| | Glueminisat | 1391 (83.4%) | 0.86 | - (-) | 5/10 (13.4%) | 12.7% (9.9%) | 0.8% (0.0%) |
| | QuteRSat | 1380 (81.4%) | 0.80 | - (-) | - (-) | 12.7% (11.1%) | 0.8% (0.0%) |
| A | Precosat | 1411 (81.4%) | 0.85 | - (-) | - (-) | 5.5% (4.7%) | 0.4% (0.0%) |
| P | EBGlucose | 1630 (78.7%) | 0.87 | - (-) | 1/10 (2.8%) | 1.9% (1.6%) | 0.8% (0.0%) |
| P | CryptoMiniSat | 1328 (81.8%) | 0.82 | - (-) | - (-) | 3.6% (3.6%) | 0.4% (0.4%) |
| L | Minisat psm | 1564 (77.9%) | 0.88 | - (-) | 1/10 (2.8%) | 0.4% (0.4%) | 1.2% (0.0%) |
| I | MPhaseSAT64 | 1529 (79.4%) | 0.82 | - (-) | - (-) | 3.6% (2.8%) | 1.6% (1.2%) |
| C | Lingeling | 1355 (82.2%) | 0.86 | - (-) | - (-) | 2.4% (2.4%) | 0.8% (0.4%) |
| A | Contrasat | 1592 (78.7%) | 0.80 | - (-) | - (-) | 2.4% (2.0%) | 1.2% (0.0%) |
| T | Minisat | 1567 (76.7%) | 0.88 | - (-) | - (-) | 2.0% (2.0%) | -0.4% (0.0%) |
| I | LR GL SHR | 1667 (75.1%) | 0.85 | - (-) | - (-) | 2.0% (1.6%) | 0.8% (0.0%) |
| O | RestartSAT | 1437 (79.4%) | 0.88 | - (-) | - (-) | 1.9% (1.2%) | 0.4% (0.4%) |
| N | Rcl | 1752 (72.7%) | 0.86 | - (-) | - (-) | 1.2% (1.2%) | 0.4% (0.0%) |
| | MiniSATagile | 1626 (74.7%)) | 0.87 | - (-) | - (-) | 1.6% (0.8%) | 0.4% (0.0%) |
| | Cirminisat | 1514 (79.8%) | 0.88 | - (-) | - (-) | 0.8% (0.8%) | 0.0% (0.0%) |
| | Glucose1 | 1614 (77.8%) | 0.86 | - (-) | - (-) | 0.0% (0.0%) | 0.0% (0.0%) |
| | EBMiniSAT | 1552 (77.5%) | 0.89 | - (-) | - (-) | 0.0% (0.0%) | 0.0% (0.0%) |

**Table 8.2:** *Performance of `SATzilla11` component solvers, disregarding instances that could not be solved by any component solver. We counted timed-out runs as 5 000 CPU seconds (the cutoff). Average correlation for s is the mean of Spearman correlation coefficients between s and all other solvers. Marginal contribution for s is negative if dropping s improved test set performance. (Usually, `SATzilla`'s solver subset selection avoids such solvers, but they can slip through when the training set is too small.) `SATzilla11(Application)` ran its backup solver `Glucose2` for 10.3% of the instances (and thereby solved 8.7%). `SATzilla11` only chose one presolver for all folds of `Random` and `Application`; for `Crafted`, it chose `Sattime` as the first presolver in 2 folds, and `Sol` as the second presolver in 1 of these; for the remaining 8 folds, it did not select presolvers.*

# Chapter 9

# Automatically Building High-performance Algorithms from Components

Designing high-performance solvers for computationally hard problems is a difficult and often time-consuming task. Although such design problems are traditionally solved by the application of human expertise, we argue instead for the use of automatic methods. In this work, we consider the design of stochastic local search (SLS) solvers for the propositional satisfiability problem (SAT). We first introduce a generalized, highly parameterized solver framework, dubbed SATenstein, that includes components drawn from or inspired by existing high-performance SLS algorithms for SAT. The parameters of SATenstein determine which components are selected and how these components behave; they allow SATenstein to instantiate many high-performance solvers previously proposed in the literature, along with trillions of novel solver strategies. We used an automated algorithm configuration procedure to find instantiations of SATenstein that perform well on several well-known, challenging distributions of SAT instances. Our experiments show that SATenstein solvers achieved dramatic performance improvements as compared to the previous state of the art in SLS algorithms; for many benchmark distributions, our new solvers also significantly outperformed all automatically tuned variants of previous state-of-the-art algorithms. To better understand the novel algorithm

designs generated in our work, we propose a new metric for quantitatively measuring the similarity between algorithm configurations, and show how to leverage this metric for visualizing the relative similarities between different solver designs. [1]

## 9.1 SATenstein-LS

SATenstein advocates designing new solvers by inducing a single parameterized solver from distinct examples in the literature, and then searching this parameter space automatically [115]. This approach is an example of—and indeed was part of the inspiration for—a design philosophy we call Programming by Optimization (PbO) [82]. In general, PbO means seeking and exposing design choices during a development process, and then automatically finding instantiations of these choices that optimize performance in a given use context. SATenstein-LS can be seen as an example of PbO in which the algorithm design space has been obtained by unifying a large number of local search schemes for SAT into a tightly integrated, highly parametric algorithm framework. However, the PbO philosophy goes further and is ultimately more general: it emphasizes encouraging developers to identify and expose design choices as parameters, rather than merely recovering parameters from existing, fully implemented examples. Because of its emphasis on changing the software development process, the PbO paradigm is also supported by programming language extensions that allow parameters and design choices to be exposed quickly and transparently. For more information, please see the PbO website at www.prog-by-opt.net.

### 9.1.1 Design

As discussed in Section 3.1.2, most SLS algorithms for SAT can be categorized into four broad categories: GSAT, WalkSAT, dynamic local search and G$^2$WSAT. Since no recent, state-of-the-art SLS solver is GSAT-based, we constructed `SATenstein-LS` by drawing components from algorithms belonging to the three remaining categories.

As shown in the high-level algorithm outline (Procedure SATenstein-LS), `SATenstein-LS`

[1] This chapter is based on the joint work with Ashiqur KhudaBukhsh, Holger Hoos, and Kevin Leyton-Brown [115].

is comprised of five major building blocks, $B1$–$B5$. Any instantiation of SATenstein-LS follows the same high-level structure:

1. Optionally execute $B1$, which performs search diversification.
2. Execute either $B2$, $B3$ or $B4$, thus performing a G$^2$WSAT-based, WalkSAT-based, or dynamic local search procedure, respectively.
3. Optionally execute $B5$, to update data structures such as promising list, clause penalties, dynamically adaptable parameters or tabu attributes.

Each of our building blocks is composed of one or more components (listed in Table 9.1); some of these components are shared across different building blocks. Each component is configurable by one or more parameters. Out of 42 parameters overall, 6 of SATenstein-LS's parameters are integer-valued (listed in Table 9.5), 19 are categorical (listed in Table 9.4), and 17 are real-valued (listed in Table 9.6). All of these parameters are exposed on the command line so that they can be optimized using an automatic configurator. After fixing the domains of integer- and real-valued parameters to between 3 and 16 values each (as we did in our experiments, reported later) the total number of valid SATenstein-LS instantiations was $2.01 \times 10^{14}$.

We now give a high-level description of each of the building blocks. $B1$ is constructed using the *SelectClause()*, *DiversificationStrategy()* and *Diversification-Probability()* components. *SelectClause()* is configured by one categorical parameter and, depending on its value, either selects an unsatisfied clause uniformly at random or selects a clause with probability proportional to its clause penalty [198]. Component *diversificationStrategy()* can be configured by a categorical parameter to do any of the following with probability *diversificationProbability()*: flip the least recently flipped variable [131], flip the least frequently flipped variable [166], flip the variable with minimum variable weight [166], or flip a randomly selected variable [80].

Block $B2$ instantiates G$^2$WSAT-based algorithms that use a data structure *promising list* that keeps track of a set of variables considered for being flipped. In the literature on G$^2$WSAT, there are two strategies for selecting a variable from the *promising list*: choosing the variable with the highest score [131] or choosing

**Procedure** SATenstein-LS(...)

```
    Input: CNF formula φ; real number cutoff;
           Booleans performDiversification, singleClauseAsNeighbor,
                    usePromisingList;
    Output: Satisfying variable assignment
    Start with random assignment A;
    Initialize parameters;
    while runtime < cutoff do
        if A satisfies φ then
            return A;
        varFlipped ← FALSE;
        if performDiversification then
B1          with probability diversificationProbability() do
B1              c ← selectClause();
B1              y ← diversificationStrategy(c) ;
B1              varFlipped ← TRUE;
        if not varFlipped then
            if usePromisingList then
B2              if promisingList is not empty then
B2                  y ← selectFromPromisingList() ;
                else
B2                  c ← selectClause();
B2                  y ← selectHeuristic(c) ;
            else
                if singleClauseAsNeighbor then
B3                  c ← selectClause();
B3                  y ← selectHeuristic(c) ;
                else
B4                  sety ← selectSet();
B4                  y ← tieBreaking(sety);
            flip y ;
B5          update();
```

the least recently flipped variable [135]. We added nine novel strategies based on variable selection heuristics from other solvers. These, to the best of our knowledge, have never been used before in the context of *promising variable* selection for G$^2$WSAT-based algorithms. For example, in previous work, variable selection mechanisms used in Novelty variants are only applied to variables of unsatisfiable clauses, not to *promising lists*. Table 9.2 lists the eleven possible strategies for *SelectFromPromisingList*.

| Component | Block | Parameters | Instantiations | Detailed Info |
|---|---|---|---|---|
| diversificationStrategy() | 1 | searchDiversificationStrategy | 4 | Table 9.4 |
| SelectClause() | 1, 2, 3 | selectClause | 2 | Table 9.4 |
| diversificationProbability() | 1 | rdp, rfp, rwp | 216 | Table 9.6 |
| selectFromPromisingList() | 2 | selectPromVariable | 4312 | Table 9.2, 9.4 |
| | | promDp, promWp, promNovNoise | | Table 9.6 |
| selectHeuristic() | 2, 3 | heuristic | | Table 9.3, 9.4 |
| | | performAlternateNovelty | $1.83 \times 10^6$ | Table 9.4 |
| | | wp, dp, wpWalk, novNoise, s, c | | Tabel 9.6 |
| selectSet() | 4 | scoringMeasure, smoothingScheme | | Table 9.4 |
| | | maxinc | 24576 | Table 9.5 |
| | | alpha,rho, sapsthresh, pflat | | Table 9.6 |
| tiebreaking() | 4 | tieBreaking | 4 | Table 9.4 |
| update() | 5 | useAdaptiveMechanism, adaptivenoisescheme, | | Table 9.4 |
| | | adaptWalkProb, performTabuSearch, | | Table 9.4 |
| | | useClausePenalty, adaptiveProm, | | Table 9.4 |
| | | adaptpromwalkprob, updateSchemePromList, | $1.76 \times 10^8$ | Table 9.4 |
| | | tabuLength, phi, theta, promPhi,promTheta, | | Table 9.5 |
| | | ps | | Table 9.6 |

**Table 9.1:** `SATenstein-LS` components.

| Param Value | Design choice | Based on |
|---|---|---|
| 1 | If freebie exists, use tieBreaking(); else, select uniformly at random | [179] |
| 2 | Variable with best score | [131] |
| 3 | Least-recently-flipped variable | [135] |
| 4 | Variable with best `VW1` score | [166] |
| 5 | Variable with best `VW2` score | [166] |
| 6 | Variable selected uniformly at random | [79] |
| 7 | Variable selection from `Novelty` | [142] |
| 8 | Variable selection from `Novelty++` | [131] |
| 9 | Variable selection from `Novelty`$^+$ | [79] |
| 10 | Variable selection from `Novelty++`$'$ | [132] |
| 11 | Variable selection from `Novelty+p` | [132] |

**Table 9.2:** Design choices for *selectFromPromisingList()*.

If *promising list* is empty, $B2$ behaves exactly as $B3$, which instantiates WalkSAT-based algorithms. As previously described in the context of $B1$, component *SelectClause()* is used to select an unsatisfiable clause $c$. The *SelectHeuristic()* component selects a variable from $c$ for flipping. Depending on a categorical parameter, *SelectHeuristic()* can behave as any of the thirteen well-known WalkSAT-based heuristics that include `Novelty` variants, `VW1` and `VW2`. Table 9.3 lists these heuristics and related continuous parameters. We also extended the `Novelty` variants with an optional "flat move" mechanism as found in the selection strategy of `gNovelty`$^+$ [161, 195].

Block $B4$ instantiates dynamic local search algorithms. The *selectSet()* compo-

| Param. Value | Selected Heuristic | Dependent Parameters |
|---|---|---|
| 1 | Novelty [142] | novnoise |
| 2 | Novelty$^+$ [80] | novnoise, wp |
| 3 | Novelty++ [131] | novnoise, dp |
| 4 | Novelty++$'$ [132] | novnoise, dp |
| 5 | R-Novelty [142] | novnoise |
| 6 | R-Novelty$^+$ [80] | novnoise, wp |
| 7 | VW1 [166] | wpwalk |
| 8 | VW2 [166] | s, c, wp |
| 9 | WalkSAT-SKC [179] | wpwalk |
| 10 | Noveltyp [132] | novnoise |
| 11 | Novelty$^+$p [132] | novnoise, wp |
| 12 | Novelty++p [132] | novnoise, dp |
| 13 | Novelty++$'$p [132] | novnoise, dp |

**Table 9.3:** List of heuristics chosen by the parameter *heuristic* and dependent parameters.

nent considers the set of variables that occur in any unsatisfied clause. It associates with each such variable $v$ a score, which depends on the *clause weights* of each clause that changes satisfiability status when $v$ is flipped. These clause weights reflect the perceived importance of satisfying each clause. For example, weights might increase the longer a clause has been unsatisfied, and decrease afterwards [91, 195]. After scoring the variables, *selectSet()* returns all variables with maximal score. Our implementation of this component incorporates three different scoring functions, including those due to [142], [179], and a novel, greedier variant that only considers the number of previously unsatisfied clauses that are satisfied by a variable flip. The *tieBreaking()* component selects a variable from the maximum-scoring set according to the same strategies used by the *diversificationStrategy()* component.

Block *B*5 updates data structures required by the previously mentioned mechanisms, (e.g., dynamic local search) after a variable has been flipped. Performing these updates in an efficient manner is of crucial importance for the performance of many SLS algorithms. As the SATenstein-LS framework supports the combination of mechanisms from many different SLS algorithms, each depending on different data structures, the implementation of the *update()* function was technically quite challenging.

| Parameter | Active When | Domain | Description |
|---|---|---|---|
| performSearchDiversification | Base level parameter | {0,1} | If true, block $B_1$ is performed |
| usePromisingList | Base level parameter | {0,1} | If true, block $B_2$ is performed |
| singleClauseAsNeighbor | Base level parameter | {0,1} | If true, block $B_3$ is performed<br>else, block $B_4$ is performed |
| selectPromVariable | usePromisingList = 1 | {1, 11} | See Table 9.2 |
| heuristic | singleClauseAsNeighbor = 1 | {1, 13} | See Table 9.3 |
| performAlternateNovelty | singleClauseAsNeighbor = 1 | {0,1} | If true, performs `Novelty`<br>variant with "flat move". |
| useAdaptiveMechanism | Base level parameter | {0,1} | If true, uses adaptive mechanisms. |
| adaptivenoisescheme | useAdaptiveMechanism = 1<br>usePromisingList = 1 | {1,2} | Specifies adaptive noise mechanisms. |
| adaptWalkProb | useAdaptiveMechanism = 1 | {0,1} | If true, walk probability or diversification<br>probability of a heuristic is adaptively<br>tuned. |
| performTabuSearch | Base level parameter | {0,1} | If true, tabu variables are<br>not considered for flipping. |
| useClausePenalty | Base level parameter | {0,1} | If true, clause penalties are computed. |
| selectClause | singleClauseAsNeighbor = 1 | {1,2} | 1 selects an UNSAT clause uniformly<br>at random.<br>2 selects an UNSAT clause with a<br>probability proportional to its<br>clause penalty. |
| searchDiversificationStrategy | performSearchDiversification = 1 | {1,2,3,4} | 1 randomly selects a variable from an<br>UNSAT clause.<br>2 selects the least-recently-flipped<br>-variable from an UNSAT clause.<br>3 selects the least-frequently-flipped<br>variable from an UNSAT clause.<br>4 selects the variable with least<br>`VW2` weight from an UNSAT clause. |
| adaptiveProm | usePromisingList = 1 | {0,1} | If true, performs adaptive versions of<br>`Novelty` variants to select variable<br>from promising list. |
| adaptpromwalkprob | usePromisingList = 1<br>adaptiveProm = 1 | {0,1} | If true, walk probability or diversification<br>probability of `Novelty` variants used<br>on promising list is adaptively tuned. |
| scoringMeasure | usePromisingList = 0<br>singleClauseAsNeighbor = 0 | {1,2,3} | Specifies the scoring measure.<br>1 uses MakeCount - BreakCount<br>2 uses MakeCount<br>3 uses -BreakCount |
| tieBreaking | usePromisingList = 1<br>selectPromVariable $\in$ { 1,4,5 }<br>*or* singleClauseAsNeighbor = 0 | {1,2,3,4} | 1 breaks ties randomly.<br>2 breaks ties in favor of the<br>least-recently-flipped variable.<br>3 breaks tie in favor of the<br>least-frequently-flipped variable.<br>4 breaks tie in favor of the<br>variable with least `VW2` score. |
| updateSchemePromList | usePromisingList = 1 | {1,2,3} | 1 and 2 follow $G^2WSAT$.<br>3 follows `gNovelty+`. |
| smoothingScheme | useClausePenalty = 1 | {1,2} | When singleClauseAsNeighbor = 1 :<br>1 performs smoothing for only random<br>3-SAT instances with 0.4 fixed<br>smoothing probability.<br>2 performs smoothing for all instances.<br>When singleClauseAsNeighbor = 0 :<br>1 performs `SAPS`-like smoothing.<br>2 performs `PAWS`-like smoothing. |

**Table 9.4:** Categorical parameters of `SATenstein-LS`. Unless otherwise mentioned, multiple "active when" parameters are combined together using AND.

### 9.1.2 Implementation and Validation

`SATenstein-LS` is built on top of UBCSAT [198], a well-known framework for

| Parameter | Active When | Description | Values considered |
|---|---|---|---|
| tabuLength | performTabuSearch = 1 | Specifies tabu step-length | 1, 3, 5, 7, **10**, 15, 20 |
| phi | useAdaptiveMechanism = 1<br>singleClauseAsNeighbor = 1 | Parameter for adaptively setting noise | 3, 4, **5**, 6, 7, 8, 9, **10** |
| theta | useAdaptiveMechanism = 1<br>singleClauseAsNeighbor = 1 | Parameter for adaptively setting noise | 3, 4, **5**, **6**, 7, 8, 9, 10 |
| promPhi | usePromisingList = 1<br>adaptiveProm = 1<br>selectPromVariable ∈ {7,8,9,10,11} | Parameter for adaptively setting noise | 3, 4, 5, 6, 7, 8, 9, 10 |
| promTheta | usePromisingList = 1<br>adaptiveProm = {1}<br>selectPromVariable ∈ {7,8,9,10,11} | Parameter for adaptively setting noise | 3, 4, 5, 6, 7, 8, 9, 10 |
| maxinc | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 2 | PAWS [195] parameter for additive clause weighting | 5, **10**, 15, 20 |

**Table 9.5:** Integer parameters of SATenstein-LS and the values considered during ParamILS tuning. Multiple "active when" parameters are combined together using AND. Existing defaults are highlighted in bold. For parameters first introduced in SATenstein-LS, default values are underlined.

developing and empirically evaluating SLS algorithms for SAT. UBCSAT makes use of a trigger-based architecture that facilitates the reuse of existing mechanisms. While designing and implementing SATenstein-LS, we not only studied existing SLS algorithms, as presented in the literature, but we also analyzed the SAT competition submissions of such algorithms. We found that the pseudocode of VW2 according to [166] differed from its SAT competition 2005 version, which includes a reactive mechanism; we included both versions in SATenstein-LS's implementation. We also found that in the SAT competition implementation of gNovelty$^+$, Novelty uses a PAWS-like [195] "flat move" mechanism. We implemented this alternate version of Novelty in SATenstein-LS and exposed a categorical parameter to choose between the two implementations. While examining the implementations of various SLS solvers, we noticed that certain key data structures were implemented in different ways. In particular, different G$^2$WSAT variants use different realizations of the update scheme of *promising list*. We included all these update schemes in SATenstein-LS and declared parameter *updateSchemePromList* to select between them.

Since SATenstein-LS is quite complex, we took great care in validating its implementation of existing SLS-based SAT solvers. We compared our SATenstein-LS implementation with ten well-known algorithms' reference implementations (specifically, every algorithm listed in Table 9.7 except for Ranov), measuring running

| Parameter | Active When | Description | Discrete Values Considered |
|---|---|---|---|
| wp | singleClauseAsNeighbor = 1<br>heuristic ∈ {2,6,11}<br>useAdaptiveMechanism = 0<br>*or* smoothingScheme = 1<br>singleClauseAsNeighbor = 0<br>useClausePenalty = 0 | Randomwalk probability for Novelty$^+$ | 0, **0.01**, 0.03, 0.04, 0.05, 0.06, 0.07, 0.1, 0.15, 0.20 |
| dp | singleClauseAsNeighbor = 1<br>heuristic ∈ {3,4,12,13}<br>useAdaptiveMechanism = 0 | Diversification probability for Novelty++ and Novelty++′ | 0.01, 0.03, **0.05**, 0.07, 0.1, 0.15, 0.20 |
| promDp | usePromisingList = 1<br>selectPromVariable ∈ {8,10}<br>adaptivePrem = 0 | Diversification probability for Novelty variants used to select variable from promising list | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.20 |
| novNoise | singleClauseAsNeighbor = 1<br>heuristic ∈ {1,2,3,4,5,6,10,11,12,13}<br>useAdaptiveMechanism = 0 | Noise parameter for all Novelty variants | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.20 |
| wpWalk | singleClauseAsNeighbor = 1<br>heuristic ∈ {7,9}<br>useAdaptiveMechanism = 0 | Noise parameter for WalkSAT and VW1 | 0.1, 0.2, 0.3, 0.4, **0.5**, 0.6, 0.7 , 0.8 |
| promWp | usePromisingList = 1<br>selectPromVariable ∈ {9,11} | Randomwalk probability for Novelty variants used to select variable from promising list | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.20 |
| promNovNoise | usePromisingList = 1<br>selectPromVariable ∈ 7,8,9,10,11 | Noise parameter for all Novelty variants used to select variable from promising list | 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7 , 0.8 |
| alpha | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 1 | Parameter for SAPS | 1.01, 1.066, 1.126, 1.189, **1.3**, 1.256, 1.326, 1.4 |
| rho | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 1 | Parameter for SAPS | 0, 0.17, 0.333, 0.5, 0.666, **0.8**, 0.83, 1 |
| sapsthresh | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 1 | Parameter for SAPS | -**0.1**, -0.2, -0.3, -0.4 |
| ps | useClausePenalty = 1<br>singleClauseAsNeighbor = 1<br>*or* singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>useAdaptiveMechanism = 0<br>smoothingScheme = 1 | Smoothing parameter for SAPS, RSAPS, and gNovelty$^+$ | 0, 0.033, **0.05**, 0.066, 0.1, 0.133, 0.166, 0.2, 0.3, **0.4**, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 |
| s | singleClauseAsNeighbor = 1<br>useAdaptiveMechanism = 0<br>*or* singleClauseAsNeighbor = 0<br>tieBreaking = 4<br>useAdaptiveMechanism = 0 | VW parameter for smoothing | **0.1**, 0.01, 0.001 |
| c | singleClauseAsNeighbor = 1<br>useAdaptiveMechanism = 0<br>*or* singleClauseAsNeighbor = 0<br>tieBreaking = 4<br>useAdaptiveMechanism = 0 | VW parameter for smoothing | 0.1, **0.01**, 0.001, 0.0001, 0.00001, 0.000001 |
| rdp | performSearchDiversification = 1<br>searchDiversificationStrategy ∈ {2,3} | Parameter for search diversification | 0.01, 0.03, **0.05**, 0.07, 0.1, 0.15 |
| rfp | performSearchDiversification = 1<br>searchDiversificationStrategy = 4 | Parameter for search diversification | 0.01, 0.03, 0.05, 0.07, 0.1, 0.15 |
| rwp | performSearchDiversification = 1<br>searchDiversificationStrategy = 1 | Parameter for search diversification | **0.01**, 0.03, 0.05, 0.07, 0.1, 0.15 |
| pflat | singleClauseAsNeighbor = 0<br>useClausePenalty = 1<br>smoothingScheme = 2 | Parameter for PAWS that controls "flat-moves" | 0.05, 0.10, **0.15**, 0.20 |

**Table 9.6:** Continuous parameters of SATenstein-LS and values considered during ParamILS tuning. Unless otherwise mentioned, multiple "active when" parameters are combined together using AND. Existing defaults are highlighted in bold. For parameters first introduced in SATenstein-LS, default values are underlined.

times as the number of variable flips.[2] These ten algorithms span G$^2$WSAT-based,

[2]Since SATenstein-LS does not use any preprocessor, we manually disabled the preprocessing steps of G2, AG2p, AG2+, and AG20 when performing this validation.

WalkSAT-based, and dynamic local search procedures, and also make use of all the prominent SLS solver mechanisms discussed earlier. Our validation results showed that in every case reference solvers and their SATenstein-LS implementations had the same run-length distributions on a small set of validation instances choosen from block world and software verification, based on a Kolmogorov-Smirnov test (5000 runs per solver–instance pair with significance threshold 0.05).

## 9.2 Experimental Setup

In order to study the effectiveness of our proposed approach for algorithm design, we configured SATenstein-LS on training sets from various distributions of SAT instances and compared the performance of the SATenstein-LS solvers thus obtained against that of several existing high-performance SAT solvers on disjoint test sets.

### 9.2.1 Benchmarks

We considered six sets of well-known benchmark instances for SAT. They can be roughly categorized into three broad categories of SAT instances, namely industrial (CBMC(SE), FAC), handmade (QCP, SW-GCP), and random (R3SAT, HGEN). Because SLS algorithms are unable to prove unsatisfiability, we constructed our benchmark sets to include only satisfiable instances.

The instance generators for HGEN [76] and FAC [200] only produce satisfiable instances. For each of these two distributions, we generated 2000 instances. For QCP [60] and SW-GCP [58], we first filtered out unsatisfiable instances using complete solvers and then randomly chose 2000 satisfiable instances. For R3SAT [180], we generated a set of 1000 instances with 600 variables and a clauses-to-variables ratio of 4.26. We identified 521 satisfiable instances and randomly chose 500 instances. Finally, we used the CBMC generator [34] to generate 611 SAT-encoded software verification instances. We preprocessed these instances using SatELite [45], identifying 604 of them as satisfiable and the remaining 7 as unsatisfiable.

We randomly split each of the six instances sets thus obtained into training and test sets of equal size.

### 9.2.2 Tuning Scenario and PAR

In order to perform automatic algorithm configuration, we first had to quantify performance using an objective function. Consistent with most previous work on SLS algorithms for SAT, we chose to focus on mean runtime. In order to deal with runs that had to be terminated at a given cutoff time, following Hutter *et al.*, [2009], we used a variant of mean runtime known as penalized average runtime (PAR-10), defined as the average runtime over a given set of runs, where timed-out runs are counted as 10 times the given cutoff time. Unless explicitly stated otherwise, all runtimes reported in this chapter were measured using PAR-10 over the respective set of instances.

To perform automated configuration, we used the FocusedILS procedure from the ParamILS framework, version 2.3 [95]. We chose this method because it has been demonstrated to operate effectively on many extremely large, discrete parameter spaces (see, e.g., 94, 97, 165, 197), and because it supports conditional parameters (discussed below). FocusedILS takes as input a parameterized algorithm (the so-called target algorithm), a specification of domains and (optionally) conditions for all parameters, a set of training instances, and an evaluation metric. It outputs a parameter configuration of the target algorithm that approximately minimizes the given evaluation metric.

As just mentioned, FocusedILS supports conditional parameters, which are important to `SATenstein-LS`. For example, condition $A|B = b$ means that $A$ is activated if $B$ takes the value $b$. When more than one such condition is given for the same parameter $A$, these are interpreted as being connected by logical "AND". For example, the two conditions, $A|B = b$ and $A|C = c$, are interpreted as $A|(B = b) \wedge (C = c)$. Some parameters in `SATenstein-LS` can be activated in more than one way. While this cannot be directly specified in the input to FocusedILS, we can express such disjunctive conditions using dummy parameters, as illustrated in the following example. Consider an algorithm `S` with four parameters, $\{A,B,C,D\}$, and where $A$ is activated if $B = b$ or $C = c$, while $D$ is activated if $A = a$. As it is impossible to express the condition $A|(B = b) \vee (C = c)$ directly in the input to FocusedILS, we introduce two dummy parameters, $A^*$ and $D^*$. Using these additional parameters, the given conditions can be expressed as $A|B = b$;

$A^*|C = c$; $A^*|B \neq b$; $D|A = a$; $D^*|A^* = a$. Since only one of $(A,A^*)/(D,D^*)$ is activated, we can simply map $A^*$ to $A$ and $D^*$ to $D$ when instantiating `S` with a parameter configuration found by FocusedILS.

We used a cutoff time of 5 CPU seconds for each target algorithm run, and allotted 7 days to each run of FocusedILS; we note that, while 5 CPU seconds is unrealistically short for assessing the performance of SAT solvers, using short cutoff times during configuration is important for the efficiency of the configuration process and typically works well, as demonstrated by our `SATenstein-LS` results. Since ParamILS cannot operate directly on continuous parameters, each continuous parameter was discretized into sets containing between 3 and 16 values that we considered reasonable (see Table 9.5). Except for a small number of cases (e.g., the parameters *s,c*) for which we used the same discrete domains as mentioned in the publication first describing it [166]), we selected these values using a regular grid over a range of values that appeared reasonable. For each integer parameter, we specified 4 to 10 values, always including the known defaults (see Table 9.6). In all cases, these choices included the parameter values required to cover the default configurations of the solvers whose components were integrated into `SATenstein-LS`'s design space. Categorical parameters and their respective domains are listed in Table 9.4. As mentioned before, based on this discretization, `SATenstein-LS`'s parameter configuration space consists of $2.01 \times 10^{14}$ distinct configurations.

Since the performance of FocusedILS can vary significantly depending on the order in which instances appear in the training set, we ran FocusedILS 20 times on the training set, using different, randomly determined instance orderings for each run. From the 20 parameter configurations obtained from FocusedILS for each instance distribution `D`, we selected the parameter configuration with the best penalized average runtime on the training set. We then evaluated this configuration on the test set. For a given distribution `D`, we refer to the corresponding instantiation of a solver `S` as `S[D]`.

### 9.2.3 Solvers Used for Performance Comparison

For each instance distribution *D*, we compared the performance of SATenstein-LS[D] against that of 11 high-performance SLS-based SAT solvers on the respective test sets. We included every SLS algorithm that won a medal in any category of a SAT competition between 2002 and 2007, because those algorithms are all part of the SATenstein-LS design space. Although dynamic local search (DLS) algorithms have not won medals in recent SAT competitions, we also included three prominent, high-performing DLS algorithms for two reasons. First, some of them represented the state of the art when introduced (e.g., SAPS [91]) and still offer competitive performance on many instances. Second, techniques used in these algorithms have been incorporated into other recent high-performance SLS algorithms. For example, the additive clause weighting scheme used in PAWS is also used in the 2007 SAT Competition winner gNovelty$^+$ [161]. We call these algorithms *challengers* and list them in Table 9.7. In order to demonstrate the full performance potential of these solvers, we also tuned the parameters for all parameterized challengers using the same configuration procedure and protocol as for SATenstein-LS, including the same choices of discrete values for continuous and integer parameters.

SATenstein-LS can be instantiated such that it emulates all 11 challenger algorithms (except for preprocessing components used in Ranov, AG2p, AG2plus, and AG20). However, in some cases, the original implementations of these algorithms are more efficient—on our data, by at most a factor of two on average per instance set—mostly, because SATenstein-LS's generality rules out some data structure optimizations. Thus, we based all of our experimental comparisons on the original algorithm implementations, as submitted to the respective SAT Competitions. The exceptions are PAWS, whose implementation within UBCSAT is almost identical to the original in terms of runtime, as well as SAPS, RSAPS, and ANOV, whose UBCSAT implementations are those used in the competitions. All of our comparisons on the test set are based on running each solver 25 times per instance, with a per-run cutoff of 600 CPU seconds.

Our goal was to improve the state of the art in SAT solving. Thus, although the design space of SATenstein-LS consists solely of SLS solvers, we have also

| Algorithm | Abbrev | Reason for Inclusion | Parameters |
|---|---|---|---|
| Ranov [160] | Ranov | gold 2005 SAT Competition (random) | wp |
| $G^2$WSAT [131] | G2 | silver 2005 SAT Competition (random) | novNoise, dp |
| VW [166] | VW | bronze 2005 SAT Competition (random) | c, s, wpWalk |
| gNovelty$^+$ [161] | GNOV | gold 2007 SAT Competition (random) | novNoise, wpWalk, ps |
| adaptG$^2$WSAT$_0$ [132] | AG20 | silver 2007 SAT Competition (random) | NA |
| adaptG$^2$WSAT$_+$ [135] | AG2+ | bronze 2007 SAT Competition (random) | NA |
| adaptNovelty$^+$ [80] | ANOV | gold 2004 SAT Competition (random) | wp |
| textttadaptG$^2$WSAT$_p$ [135] | AG2p | performance comparable to $G^2$WSAT [131], Ranov, and adaptG$^2$WSAT$_+$; see [132] | NA |
| SAPS [91] | SAPS | prominent DLS algorithm | alpha, ps, rho, sapsthresh, wp |
| RSAPS [91] | RSAPS | prominent DLS algorithm | alpha, ps, rho, sapsthresh, wp |
| PAWS [195] | PAWS | prominent DLS algorithm | maxinc, pflat |

**Table 9.7:** Our eleven *challenger* algorithms.

| Category | Solver | Reason for Inclusion |
|---|---|---|
| Industrial (CBMC(SE) and FAC) | Picosat [18, 19] | gold, silver 2007 SAT Competition (industrial) |
| | Minisat2.0 [187] | bronze, silver 2007 SAT Competition (industrial) |
| Handmade (QCP and SW-GCP) | Minisat2.0 [187] | bronze, silver 2007 SAT Competition (handmade) |
| | March_pl [73] | Improved, bug-free version of March_ks [74], gold in 2007 SAT Competition (handmade) |
| Random (HGEN and R3SAT) | Kcnfs_04 [44] | silver 2007 SAT Competition (random) |
| | March_pl [73] | Improved, bug-free version of March_ks [74], silver in 2007 SAT Competition (random) |

**Table 9.8:** Complete solvers we compared against.

compared its performance to that of high-performance complete solvers (listed in Table 9.8). Unlike SLS solvers, these complete solvers are deterministic. Thus, for every instance in each distribution, we ran each complete solver once with a per-run cutoff of 600 CPU seconds.

### 9.2.4 Execution Environment

We performed our experiments on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1. Our computer cluster was managed by a distributed resource manager, Sun Grid Engine software (version 6.0). Runtimes for all algorithms (including FocusedILS) were measured as CPU time on these reference machines. Each run of any solver only used one CPU.

## 9.3 Performance Results

In this section, we present the results of performance comparisons between SATenstein-LS and the 11 challenger SLS solvers (listed in Table 9.7), configured versions of these challengers, and two complete solvers for each of our benchmark distributions (listed in Table 9.8). Although in our configuration experiment, we optimized SATenstein-LS for penalized average runtime (PAR-10), we also examine its performance in terms of other performance metrics, such as median runtime and percentage of instances solved within the given cutoff time.

### 9.3.1 Comparison with Challengers

For every one of our six benchmark distributions, we were able to find a SATenstein-LS configuration that outperformed all 11 challengers. Our results are summarized in Table 9.9 and Figure 9.1.

In terms of penalized average runtime, the performance metric we explicitly optimized using ParamILS (with a cutoff time of 5 CPU seconds rather than the 600 CPU seconds used here for testing, as explained in Section 5.2), our SATenstein-LS solvers achieved better performance than every challenger on every distribution. For QCP, HGEN, and CBMC(SE), SATenstein-LS achieved a PAR-10 score that was orders of magnitude better than the respective best challengers. For SW-GCP, R3SAT, and FAC, there was substantial, but less dramatic improvement. The modest improvement in R3SAT was not very surprising (Figure 9.1: Left); R3SAT is a well-known SAT distribution on which SLS solvers have been evaluated and optimized for decades. Conversely, on a new benchmark distribution, CBMC(SE), where DPLL solvers represent the state of the art,

| | | | | | | |
|---|---|---|---|---|---|---|
| | **0.08** | **0.03** | **1.11** | **0.02** | **10.89** | **4.75** |
| SATenstein-LS[D] | **0.01** | **0.02** | 0.14 | **0.01** | **7.90** | **0.02** |
| [115] | **100%** | **100%** | **100%** | **100%** | **100%** | **100%** |
| **Solvers** | **QCP** | **SW-GCP** | **R3SAT** | **HGEN** | **FAC** | **CBMC(SE)** |
| | 1054.99 | 0.64 | 2.14 | 137.02 | 3594.40 | 2169.77 |
| AG20 | 0.03 | 0.11 | 0.13 | 0.57 | N/A | 0.56 |
| [132] | 81.2% | **100%** | **100%** | 98.1% | 35.9% | 61.1% |
| | 1119.96 | 0.43 | 2.35 | 105.30 | 1954.83 | 2294.24 |
| AG2p | 0.02 | 0.06 | 0.14 | 0.48 | 330.26 | 2.57 |
| [135] | 80.1% | **100%** | **100%** | 98.4% | 80.6% | 61.1% |
| | 1091.37 | 0.67 | 3.04 | 148.28 | 1450.89 | 2181.92 |
| AG2+ | 0.03 | 0.08 | 0.16 | 0.59 | 238.31 | 0.64 |
| [135] | 80.3% | **100%** | **100%** | 98.0% | 91.0% | 61.1% |
| | 25.42 | 4.86 | 11.17 | 109.94 | 2897.52 | 2021.22 |
| ANOV | 0.02 | 0.04 | 0.15 | 0.50 | 588.23 | 3.10 |
| [80] | 99.6% | **100%** | **100%** | 98.6% | 51.4% | 61.1% |
| | 2942.13 | 4092.29 | 3.69 | 104.55 | 5947.80 | 2139.12 |
| G2 | 341.60 | N/A | 0.13 | 0.60 | N/A | 0.57 |
| [131] | 50.9% | 31.0% | **100%** | 98.7% | 0% | 65.4% |
| | 414.69 | 1.20 | 11.14 | 52.58 | 5935.39 | 2236.85 |
| GNOV | 0.03 | 0.09 | 0.15 | 0.71 | N/A | 0.67 |
| [161] | 93.3% | **100%** | **100%** | 99.4% | 0% | 61.5% |
| | 1127.84 | 4495.50 | 1.77 | 62.18 | 22.05 | 1693.82 |
| PAWS | 0.03 | N/A | **0.08** | 0.82 | 10.41 | 0.18 |
| [195] | 81.0% | 24.3% | **100%** | 99.4% | **100%** | 70.8% |
| | 73.38 | 0.15 | 18.29 | 151.11 | 887.33 | 1227.07 |
| RANOV | 0.1 | 0.12 | 0.36 | 0.90 | 152.16 | 0.58 |
| [160] | 99.1% | **100%** | **100%** | 98.2% | 96.8% | 79.7% |
| | 1255.94 | 5635.54 | 18.42 | 33.28 | 17.86 | 827.81 |
| RSAPS | 0.05 | N/A | 1.86 | 2.33 | 11.53 | **0.02** |
| [91] | 79.2% | 5.4% | **100%** | 99.7% | **100%** | 85.0% |
| | 1248.34 | 3864.74 | 22.93 | 40.17 | 16.41 | 646.89 |
| SAPS | 0.04 | N/A | 1.77 | 2.65 | 10.56 | **0.02** |
| [91] | 79.4% | 34.2% | **100%** | 99.5% | **100%** | 89.7% |
| | 1022.69 | 161.74 | 12.45 | 176.18 | 3382.02 | 385.12 |
| VW | 0.25 | 40.26 | 0.82 | 3.13 | N/A | 0.23 |
| [166] | 81.9% | 99.4% | **100%** | 97.8% | 35.3% | 93.4% |

**Table 9.9:** Performance of `SATenstein-LS` and the 11 challengers. Every algorithm was run 25 times with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the the PAR10 score; $b$ (middle) is the median of the median runtime(where the outer median is taken over the instances and the inner median over the runs); $c$ (bottom) is the percentage of instances solved (median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold, and the best-scoring challenger(s) are underlined.

SATenstein-LS solvers performed markedly better than every SLS-based challenger. We were surprised to see the amount of improvement we obtained for HGEN, a hard random SAT distribution very similar to R3SAT, and QCP, a widely-known SAT distribution. We noticed that on HGEN, some older solvers such as SAPS and PAWS performed much better than more recent medal winners such as GNOV and AG20. Also, for QCP, a somewhat older algorithm, ANOV, turned out to be the best challenger. These observations led us to believe that the strong performance of SATenstein-LS was partly due to the fact that the past seven years of SLS SAT solver development have not taken these types of distributions into account and have not yielded across-the-board improvements in SLS solver performance.

We also evaluated the performance of SATenstein-LS solvers using two other performance metrics: median-of-median runtime and percentage of solved instances. If a solver finishes most of the runs on most instances, the capped runs will not affect its median-of-median performance, and hence the metric does not need a way of accounting for the cost of capped runs. (When the median of medians is a capped run, we say that the metric is undefined.) Table 9.9 shows that, although the SATenstein-LS solvers were obtained by optimizing for PAR-10, they still outperformed every challenger in every distribution except for R3SAT, in which the challengers achieved slightly better performance than SATenstein-LS. Finally, we measured the percentage of instances on which the median runtime was below the cutoff used for capping runs. According to this measure, SATenstein-LS either equalled or beat every challenger, since it solved 100% of the instances in every benchmark set. In contrast, only 4 challengers managed to solve more than 50% of instances in every test set. Overall, SATenstein-LS solvers scored well on these measures, even though its performance was not explicitly optimized for them.

The relative performance of the challengers varied significantly across different distributions. For example, the three dynamic local search solvers (SAPS, PAWS, and RSAPS) performed substantially better than the other challengers on factoring instances (FAC). However, on SW-GCP, their performance was weak. Similarly, GNOV (SAT Competition 2007 winner in the random satisfiable category) performed very poorly on our two industrial benchmark distributions, CBMC(SE)

| Challengers | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| AG20 | 76.1 (23.3) | 95.8 (4.2) | 45.6 (17.6) | 98.0 (1.5) | 100.0 (0.0) | 100.0 (0.0) |
| AG2p | 70.6 (28.6) | 88.9 (10.7) | 47.6 (15.2) | 98.2 (1.1) | 100.0 (0.0) | 100.0 (0.0) |
| AG2+ | 75.4 (24.1) | 94.3 (5.7) | 61.6(12.4) | 98.5 (1.1) | 100.0 (0.0) | 100.0 (0.0) |
| ANOV | **57.7** (**40.4**) | 68.5 (27.2) | 57.2 (8.0) | 97.6 (1.3) | 99.9 (0.0) | 100.0 (0.0) |
| G2 | 81.4 (18.6) | 100.0 (0.0) | 34.0 (15.2) | 98.0 (1.4) | 100.0 (0.0) | 100.0 (0.0) |
| GNOV | 97.5 (2.4) | 99.6 (0.4) | 48.8 (16.4) | 99.4 (0.4) | 100.0 (0.0) | 100.0 (0.0) |
| PAWS | 69.0 (30.1) | 100.0 (0.0) | **19.6** (**3.2**) | 100.0 (0.0) | 68.8 (0.0) | 100.0 (0.0) |
| RANOV | 100.0 (0.0) | **100.0** (**0.0**) | 99.2 (0.0) | 100.0 (0.0) | 100.0 (0.0) | 100.0 (0.0) |
| RSAPS | 71.5 (28.0) | 99.8 (0.2) | 96.8 (3.2) | **100.0** (**0.0**) | 81.1 (0.0) | 42.2 (54.5) |
| SAPS | 70.9 (28.5) | 100.0 (0.0) | 96.8 (2.4) | 100.0 (0.0) | **73.7** (**0.2**) | 48.8 (48.5) |
| VW | 85.3 (14.7) | 100.0 (0.0) | 100.0 (0.0) | 100.0 (0.0) | 100.0 (0.0) | **100.0** (**0.0**) |

**Table 9.10:** Percentage of instances on which SATenstein-LS achieved better (equal) median runtime than each of the 11 challengers. Medians were taken over 25 runs on each instance with a cutoff time of 600 CPU seconds per run.

and FAC, but solved SW-GCP and HGEN instances quite efficiently.[3] This suggests that different distributions are most efficiently solved by rather different solvers. We are thus encouraged that our automatic algorithm construction process was able to find good configurations for each distribution.

So far, we have discussed performance metrics that describe aggregate performance over the entire test set. One might wonder if SATenstein-LS's strong performance is due its ability to solve relatively few instances very efficiently, while performing poorly on others. Table 9.10 shows that this is typically not the case, summarizing the performance of each SATenstein-LS solver compared to each challenger on a per-instance basis. Except for R3SAT, SATenstein-LS solvers outperformed the respective best challengers for each distribution on a per-instance basis. R3SAT was an exception: PAWS outperformed SATenstein-LS[R3SAT] most frequently (77.2%), but still achieved a lower PAR-10 score, indicating that SATenstein-LS[R3SAT] achieved dramatically better performance than PAWS on a relatively small number of hard instances.

[3] Interestingly, on both types of random instances we considered, GNOV failed to outperform some of the older solvers, in particular, PAWS and RSAPS.

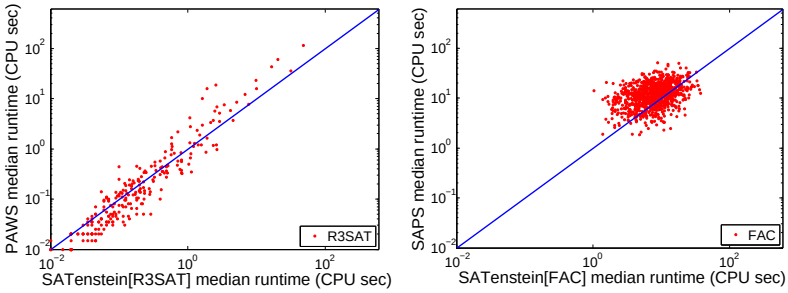



**Figure 9.1:** Performance comparison of SATenstein-LS and the best challenger. Left: R3SAT; Right: FAC. Medians were taken over 25 runs on each instance with a cutoff time of 600 CPU seconds per run.

### 9.3.2 Comparison with Automatically Configured Versions of Challengers

The fact that SATenstein-LS solvers achieved significantly better performance than all 11 challengers with default parameter configurations (i.e., those selected by their designers) admits two possible explanations. First, it could be due to the fact that SATenstein-LS's (vast) design space includes useful new configurations that combine solver components in novel ways. Second, the performance gains may have been achieved simply by better configuring existing SLS algorithms within their existing, and quite small, design spaces. To determine which of these two hypotheses holds, we compared SATenstein-LS solvers against challengers configured for optimized performance on our benchmark sets, using the same automated configuration procedure and protocol.

Table 9.11 summarizes the performance thus obtained, and Figure 9.2 shows the PAR-10 ratios of SATenstein-LS solvers over the default and configured challengers. Compared to challengers with default configurations (see Table 9.9), the specifically optimized versions of the challenger solvers often achieved significantly better performance, reducing their performance gaps to SATenstein-LS solvers. For example, automatic configuration of G2 led to a speedup of 5 orders of magnitude in terms of PAR-10 on SWGCP and solved 100% of the instances in that benchmark set within a 600 second cutoff (vs. 31% for G2 default). However, it is worth noting that the configured challengers sometimes also exhibited worse performance than the default configurations (in the worst case, VW[SWGCP] was

| Solvers | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| | 26.13 | 0.06 | 2.68 | 119.75 | 1731.16 | 994.94 |
| ANOV[D] | **0.02** | **0.04** | 0.12 | **0.54** | 296.84 | 0.50 |
| [80] | 99.6% | **100%** | **100%** | 98.2% | 90.1% | 83.4% |
| | 514.29 | **0.05** | 3.64 | 98.70 | 617.83 | 1084.60 |
| G2[D] | 0.03 | 0.05 | 0.15 | 0.75 | 110.42 | 0.58 |
| [131] | 91.4% | **100%** | **100%** | 99.1% | 97.8% | 81.4% |
| | 417.33 | 0.22 | 8.87 | 68.24 | 5478.75 | 2195.76 |
| GNOV[D] | 0.03 | 0.09 | 0.17 | 0.62 | N/A | 0.19 |
| [161] | 92.9% | **100%** | **100%** | 99.4% | 0.3% | 61.8% |
| | 68.06 | 0.70 | 1.91 | 64.48 | 22.01 | 1925.56 |
| PAWS[D] | **0.02** | 0.35 | **0.09** | 0.83 | 10.39 | 0.50 |
| [195] | 99.2% | **100%** | **100%** | 99.4% | **100%** | 67.7% |
| | 75.06 | 0.15 | 13.85 | 141.61 | 336.27 | 1223.83 |
| RANOV[D] | 0.1 | 0.12 | 0.24 | 0.77 | 95.53 | 0.47 |
| [160] | 98.9% | **100%** | **100%** | 98.1% | 100% | 80.4% |
| | 868.37 | 0.19 | 1.32 | 42.99 | 12.17 | 67.59 |
| RSAPS[D] | 0.04 | 0.15 | 0.11 | 0.64 | 7.86 | **0.02** |
| [91] | 85.2% | **100%** | **100%** | 99.5% | **100%** | 99.0% |
| | 27.69 | 0.31 | 1.54 | **31.77** | **10.68** | 62.63 |
| SAPS[D] | 0.06 | 0.21 | 0.16 | 0.75 | **7.00** | **0.02** |
| [91] | 99.8% | **100%** | **100%** | 99.6% | **100%** | 99.0% |
| | **0.33** | 417.71 | **1.26** | 57.44 | 32.38 | **16.45** |
| VW[D] | **0.02** | 8.43 | 0.15 | 1.00 | 17.60 | **0.02** |
| [166] | **100%** | 94.8% | **100%** | 99.6% | 100% | **100%** |

**Table 9.11:** Performance summary of the automatically configured versions of 8 challengers (three challengers have no parameters). Every algorithm was run 25 times on each problem instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the the penalized average runtime; $b$ (middle) is the median of the median runtimes over all instances (not defined if fewer than half of the median runs failed to find a solution within the cutoff time); $c$ (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold.

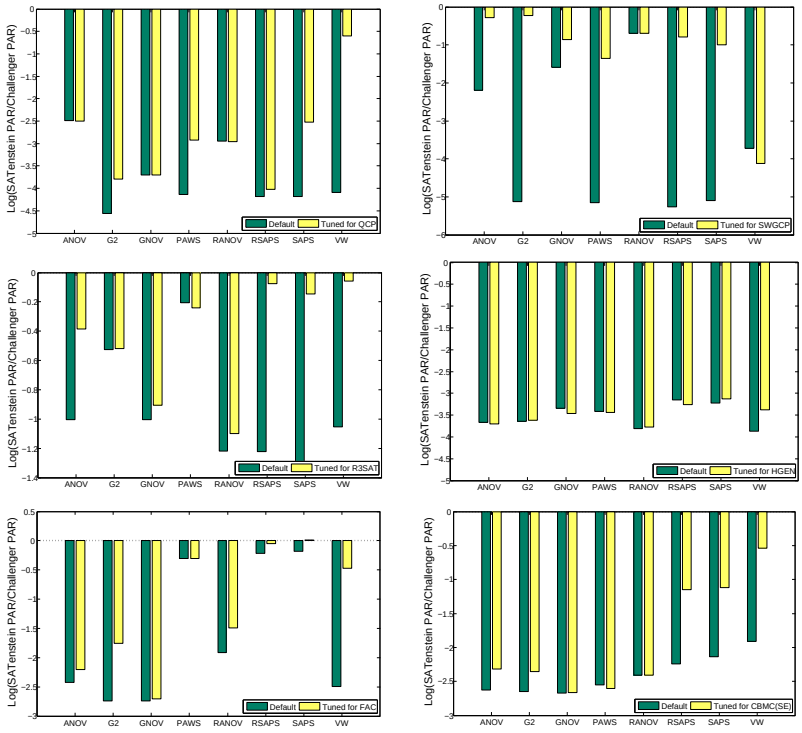



**Figure 9.2:** Performance of `SATenstein-LS` solvers *vs* challengers with default and optimized configurations. For every benchmark distribution `D`, the base-10 logarithm of the ratio between `SATenstein[D]` and one challenger (default and optimized) is shown on the y-axis, based on data from Tables 9.9 and 9.11. Top-left: QCP; Top-right: SWGCP; Middle-left: R3SAT; Middle-right: HGEN; Bottom-left: FAC; Bottom-right: CBMC(SE)

2.58 times slower than `VW` default in terms of PAR-10 with a cutoff of 600 CPU seconds). This was caused by the short cutoff time used during the configuration process, as motivated in Section 5.2; had we used the same 5 CPU second cutoff time for computing PAR-10 score, we expected that the configured challengers would have always outperformed the default versions.

Examining benchmark distributions individually and ranging over our 8 challengers, we observed average and median speedups over default configurations of

396 and 3.58 (for QCP), 15,900 and 3,240 (for SWGCP), 5.84 and 2.74 (for R3SAT), 1.23 and 1.01 (HGEN), 15.4 and 1.61 (FAC), 6.61 and 2.00(CBMC(SE)). We were surprised to observe only small speedups for all challengers on HGEN. Considering challengers individually and ranging over our 6 benchmark distributions, average and median PAR-10 improvement was 15.0 and 1.85 (for ANOV), 13,200 and 3.84 (for G2), 1.74 and 1.05 (for GNOV), 1,070 and 0.98 (for PAWS), 1.33 and 1.03 (for RANOV), 4,870 and 6.85 (for RSAPS), 2,080 and 12.3 (for SAPS), 539 and 16.6 (for VW). RANOV showed the smallest performance improvement as a result of automated configuration across all benchmarks; this is likely due to RANOV's small parameter space (it has only one parameter).

Table 9.12 shows the performance of our SATenstein-LS solvers, the best default challengers, and the best automatically configured challengers. For QCP, HGEN and CBMC(SE), the SATenstein-LS solvers still significantly outperformed the best configured challengers. For R3SAT and SWGCP, the performance difference was small, but still above 10%. The only benchmark where the best configured challenger outperformed SATenstein-LS was FAC. As we will see later (in Figure 9.3), SATenstein-LS[FAC] turns out to be very similar to the best configured challenger, SAPS[FAC].

Overall, these experimental results provide evidence in favour of our first hypothesis: the good performance of SATenstein-LS solvers is due to combining components gleaned from existing high-performance algorithms in novel ways. Later in Section 9.4.3, we provide detailed analysis for demonstrating the design difference between configured SATenstein-LS solvers and component solvers.

### 9.3.3 Comparison with Complete Solvers

Table 9.13 compares the performance of SATenstein-LS solvers and four prominent complete SAT solvers (two for each distribution). For four out of our six benchmark distributions, SATenstein-LS solvers comprehensively outperformed the complete solvers. For the other two industrial distributions (FAC and CBMC(SE)), the performance of the selected complete solvers was much better than that of either the SATenstein-LS solvers and any of our other local search solvers. The success of DPLL-based complete solvers on industrial instances is not surprising;

| Distribution | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| Best Challenger(default) | ANOV | RANOV | PAWS | RSAPS | SAPS | VW |
| | 25.42 | 0.15 | 1.77 | 33.28 | 16.41 | 385.12 |
| Performance | 0.02 | 0.12 | **0.08** | 2.33 | 10.56 | 0.23 |
| | 99.6% | **100%** | **100%** | 99.7% | **100%** | 93.4% |
| Best Challenger(tuned) | VW[D] | G2[D] | VW[D] | SAPS[D] | SAPS[D] | VW[D] |
| | 0.33 | 0.05 | 1.26 | 31.77 | **10.68** | 16.45 |
| Performance | 0.02 | 0.05 | 0.15 | 0.75 | **7.00** | **0.02** |
| | **100%** | **100%** | **100%** | 99.6% | **100%** | **100%** |
| | **0.08** | **0.03** | **1.11** | **0.02** | 10.89 | **4.75** |
| SATenstein-LS[D] | **0.01** | **0.02** | 0.14 | **0.01** | 7.90 | **0.02** |
| Performance | **100%** | **100%** | **100%** | **100%** | **100%** | **100%** |

**Table 9.12:** Performance of SATenstein-LS solvers, the best challengers with default configurations and the best automatically configured challengers. Every algorithm was run 25 times on each instance with a cutoff of 600 CPU seconds per run. Each table entry $\langle i, j \rangle$ indicates the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the the penalized average runtime; $b$ (middle) is the median of the median runtimes over all instances; $c$ (bottom) is the percentage of instances solved (i.e., those with median runtime $<$ cutoff).

it is widely believed to be due to their ability to take advantage of instance structure (by means of unit propagation and clause learning). Our results confirm that state-of-the-art local search solvers cannot compete with state-of-the-art DPLL solvers on industrial instances. However, SATenstein-LS solvers have made significant progress in closing the gap. For example, for CBMC(SE), state-of-the-art complete solvers were five orders of magnitude better than the next-best SLS challenger, VW. SATenstein-LS reduced the performance gap to three orders of magnitude. We also obtained some modest improvements (a factor of 1.51) for FAC.

### 9.3.4 Configurations Found

To better understand the automatically-constructed SATenstein-LS solvers, we compared their automatically selected design choices to the design of the existing SLS solvers for SAT. The full parameter configurations of the six SATenstein-LS solvers are shown in Table 9.14.

| Distribution | QCP | SW-GCP | R3SAT | HGEN | FAC | CBMC(SE) |
|---|---|---|---|---|---|---|
| Complete Solver | Minisat2.0 | Minisat2.0 | Kcnf_04 | Kcnf_04 | Minisat2.0 | Minisat2.0 |
| | 35.05 | 2.17 | 4905.6 | 3108.77 | 0.03 | 0.23 |
| Performance | 0.02 | 0.9 | N/A | N/A | **0.02** | 0.03 |
| | 99.5% | **100%** | 18.8% | 49.5% | **100%** | **100%** |
| Complete Solver | March_ pl | March_ pl | March_ pl | March_ pl | Picosat | Picosat |
| | 120.29 | 253.99 | 3543.01 | 2763.41 | **0.02** | **0.03** |
| Performance | 0.2 | 1.12 | N/A | 400.78 | **0.02** | **0.01** |
| | 98.1% | 95.8% | 42.0% | 55.2% | **100%** | **100%** |
| | **0.08** | **0.03** | **1.11** | **0.02** | 10.89 | 4.75 |
| SATenstein-LS[D] | **0.01** | **0.02** | 0.14 | **0.01** | 7.90 | 0.02 |
| Performance | **100%** | **100%** | **100%** | **100%** | **100%** | **100%** |

**Table 9.13:** Performance summary of SATenstein-LS and the complete solvers. Every complete solver was run once (SATenstein-LS was run 25 times) on each instance with a per-run cutoff of 600 CPU seconds. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm $i$ on distribution $j$ as $a/b/c$, where $a$ (top) is the the penalized average runtime; $b$ (middle) is the median of the median runtimes over all instances (for SATenstein-LS, it is the median of the median runtimes over all instances. the median runtimes are not defined if fewer than half of the median runs failed to find a solution within the cutoff time); $c$ (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold.

SATenstein-LS[QCP] uses building blocks 1, 3, and 5. Recall that block 1 is used for performing search diversification, and block 5 is used to update data structures, tabu attributes and clause penalties. In block 3, which is used to instantiate a solver belonging to the WalkSAT architecture, the *heuristic* is based on Novelty$^{++'}$, and in block 1, *diversification* flips the least-frequently-flipped variable from an UNSAT clause. SATenstein-LS[SW-GCP] is similar to SATenstein-LS[QCP] but does not use block 1. In block 3, the *heuristic* is based on Novelty++ as used within G2. SATenstein-LS[R3SAT] uses blocks 1, 4 and 5; it is closest to SAPS, but performs search diversification. A tabu list with length 3 is used to exclude some variables from the search neighborhood. Recall that block 4 is used to instantiate dynamic local search algorithms. SATenstein-LS[HGEN] uses blocks 1, 3, and 5. It is similar to SATenstein-LS[QCP] but uses a *heuristic* based on VW1 as well as a tabu list of length 3. SATenstein-LS[FAC] uses blocks 4 and 5; its instantiation closely resembles that of SAPS, but differs in

| Distribution | Parameter Configuration |
|---|---|
| QCP | -useAdaptiveMechanism 0 -performSearchDiversification 1 -usePromisingList 0 -singleClauseAsNeighbor 1 -adaptWalkProb 0 -selectClause 1 -useClausePenalty 0 -performTabuSearch 0 -heuristic 4 -performAlternateNovelty 0 -searchDiversificationStrategy 3 -dp 0.07 -c 0.0001 -novNoise 0.5 -rfp 0.1 -s 0.1 |
| SW-GCP | -useAdaptiveMechanism 0 -performSearchDiversification 0 -usePromisingList 0 -singleClauseAsNeighbor 1 -adaptWalkProb 0 -selectClause 1 -useClausePenalty 0 -performTabuSearch 0 -heuristic 3 -performAlternateNovelty 0 -dp 0.01 -c 0.01 -novNoise 0.1 -s 0.1 |
| R3SAT | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 0 -scoringMeasure 3 -tieBreaking 2 -useClausePenalty 1 -searchDiversificationStrategy 1 -smoothingScheme 1 -tabuLength 3 -performTabuSearch 1 -alpha 1.189 -ps 0.1 -rho 0.8 -sapsthresh -0.1 -rwp 0.05 -wp 0.01 |
| HGEN | -useAdaptiveMechanism 0 -performSearchDiversification 1 -usePromisingList 0 -singleClauseAsNeighbor 1 -tabuLength 3 -performTabuSearch 1 -useClausePenalty 0 -searchDiversificationStrategy 4 -adaptWalkProb 0 -selectClause 1 -heuristic 7 -c 0.001 -rfp 0.15 -s 0.1 -wpWalk 0.1 |
| FAC | -useAdaptiveMechanism 0 -performSearchDiversification 0 -singleClauseAsNeighbor 0 -scoringMeasure 3 -tieBreaking 1 -useClausePenalty 1 -smoothingScheme 1 -tabuSearch 0 -alpha 1.189 -ps 0.066 -rho 0.83 -sapsthresh -0.3 -wp 0.03 |
| CBMC(SE) | -useAdaptiveMechanism 0 -performSearchDiversification 1 -singleClauseAsNeighbor 0 -useClausePenalty 1 -smoothingScheme 1 -performTabuSearch 0 -searchDiversificationStrategy 4 -scoringMeasure 3 -tieBreaking 2 -alpha 1.066 -ps 0 -rho 0.83 -sapsthresh -0.3 -wp 0.01 -rfp 0.1 |

**Table 9.14:** `SATenstein-LS` parameter configuration found for each distribution.

the way in which variable scores are computed. `SATenstein-LS[CBMC(SE)]` uses blocks 1, 4, and 5; it computes variable scores using `-BreakCount` and employs a search diversification strategy similar to that of `VW`.

Interestingly, none of the six `SATenstein-LS` configurations we found uses a *promising list* (block 2), a technique integrated into many recent SAT Competition winners. This indicates that many interesting designs that could compete with existing high-performance solvers still remain unexplored in SLS design space. In addition, we found that all `SATenstein-LS` configurations differ from existing SLS algorithms (except for `SATenstein[FAC]`, whose configuration and performance is similar to `SAPS`). This underscores the importance of an automated approach, since manually finding such good configurations from a huge design space is very difficult.

## 9.4 Quantitative Comparison of Algorithm Configurations

So far, we have examined the parameter configurations identified for each of our six benchmark distributions, quantitatively assessed the performance they achieved, and qualitatively observed that most were substantially different from existing solver designs. We would like to be able to dig deeper, saying something about how *similar* each of these configurations is to existing designs. More broadly, we would like automatic and quantitative techniques for comparing different solver designs in terms of their similarities and differences. In the case of highly configurable algorithms like `SATenstein-LS`, this requires some sophistication, because parameters share conditional dependencies. The approach presented in the following can deal with arbitrary levels of conditional parameter dependence and be applied to arbitrary parametric algorithms. Unlike previous work on this problem, which only considered the edit distance between configurations [153], the metric we introduce takes into account not only differences between the parameters that are active in two given configurations, but also the importance of each parameter.

### 9.4.1 Concept DAGs

To preserve the hierarchical structure of parameter dependencies, we use a novel data structure called a *concept DAG* to represent algorithm configurations. Our notion of a concept DAG is based on that of a concept tree [217]. We then define four operators whose repeated application that can be used to map between arbitrary concept DAGs, and assign each operator a cost. To compare two parameter configurations, we first represent them using concept DAGs and then define their similarity as the minimal total cost of transforming one DAG into the other.

A *concept DAG* is a six-tuple $G = (V, E, L^V, R, D, M)$, where $V$ is a set of nodes, $E$ is a set of directed edges between the nodes in $V$ such that they form an acyclic graph, $L^V$ is a set of lexicons (terms) for concepts used as node labels, $R$ is a distinguished node called the root, $D$ is the domain of the nodes, and $M$ is an injective mapping from $V$ to $L^V$. A parameter configuration can be expressed as a concept DAG[4] in which each node in $V$ represents a parameter and each directed edge in

[4]It was necessary for us to base this data structure around DAGs rather than trees because pa-

$E$ represents the conditional dependence relationship between two parameters. $D$ is the domain of all parameters and $M$ specifies which values any given parameter $v \in V$ can take. We add an artificial root node $R$, which connects to all parameter nodes that do not have any parent, and refer to these parameters as *level 1 parameters*.

We can transform one concept DAG into another by a series of delete, insert, relabel and move operations, each of which has an associated cost. For measuring the degree of similarity between two algorithm configurations, we first express them as concept DAGs, $DAG_1$ and $DAG_2$. We define the distance between these DAGs as the minimal total cost required for transforming $DAG_1$ into $DAG_2$. Obviously, the distance between two identical configurations is 0.

The parameters with the biggest impact on an algorithm's execution path are likely to have low level (i.e., to be conditional upon few or no other parameters) and/or to turn on a complex mechanism (i.e., to have many parameters conditional upon them). Therefore, we say that the importance of a parameter $v$ is a function of its depth (the length of the longest path from the root $R$ of the given concept DAG to $v$) and the total number of other parameters conditional on it. To capture this definition of importance, we define the cost of each of the four DAG-transforming operations as follows.

**Deletion cost** $C(\text{delete}(v)) = \frac{1}{|V|} \cdot (height(DAG) - depth(v) + 1 + |DE(v)|)$, where $height(DAG)$ is the height of the $DAG$, $depth(v)$ is the depth of node $v$ and $DE(v)$ is the set of descendants of node $v$. This captures the idea that it is more costly to delete low-level parameters and parameters that turn on complex mechanisms.

**Insertion cost** $C(\text{insert}(u,v)) = \frac{1}{|V|} \cdot (height(DAG) - depth(u) + 1 + |DE(v)|)$, where $DE(v)$ is the set of descendants of $v$ after the insertion.

**Moving cost** $C(\text{move}(u,v)) = \frac{|V|-2}{2 \cdot |V|} \cdot [C(\text{delete}(v)) + C(\text{insert}(u,v))]$, where $|V| > 2$.

rameters may have more than one parent parameter, where the child is only active if the parents take certain values. For example, the noise parameter *phi* is only activated when both *useAdaptiveMechanism* and *singleClauseAsNeighbor* are turned on.

**Relabelling cost** $C(\text{relabel}(v, l^v, l^{v^*}) = [C(\text{delete}(v)) + C(\text{insert}(u, v))] \cdot s(l^v, l^{v^*})$, where $s(l^v, l^{v^*})$ is a measure of the distance between two labels $l^v$ and $l^{v^*}$. For parameters with continuous domains, $s(l^v, l^{v^*}) = |l^v - l^{v^*}|$. For parameters whose domains are some finite, ordinal and discrete set $\{l^{v_1}, l^{v_2}, \ldots, l^{v_k}\}$, $s(l^v, l^{v^*}) = \text{abs}(v - v^*)/(k-1)$, where $\text{abs}(v - v^*)$ measures the number of intermediate values between $v$ and $v^*$. For categorical parameters, $s(l^v, l^{v^*}) = 0$ if $l^v = l^{v^*}$ and 1 otherwise.

### 9.4.2 Comparison of `SATenstein-LS` Configurations

We are now able to compare our automatically identified `SATenstein-LS` solver designs to all of the challengers, without having to resort to expert knowledge. As previously mentioned, 3 of our 11 challengers (`AG2p`, `AG2plus`, and `AG20`) are parameter free. Furthermore, `RANOV` only differs from `ANOV` by the addition of a preprocessing step, and so can be understood as a variant of the same algorithm. This leaves us with 7 parameterized challengers to consider. For each, we sampled 50 configurations (consisting of the default configuration, one configuration optimized for each of our 6 benchmark distributions, and 43 random configurations). We then computed the pairwise transformation cost between the resulting 359 configurations ($7 \times 50$ challengers' configurations + 6 `SATenstein-LS` solvers + `AG2p` + `AG2plus` + `AG20`). The result can be understood as a graph with 359 nodes and 128 522 edges, with nodes corresponding to concept DAGs, and edges labeled by the minimum transformation cost between them. To visualize this graph, we used a dimensionality reduction method to map it onto a plane, with the aim of positioning points so that the Euclidean distance between every pair of points approximated their transformation cost as accurately as possible. In particular, we used the `Isomap` algorithm [194], which builds on a multidimensional scaling technique but has the ability to preserve the intrinsic geometry of the data, as captured in the geodesic manifold distances between all pairs of data points. It is capable of discovering the nonlinear degrees of freedom that underlie complex natural observations. We implemented the transformation cost computation code using Matlab, and performed all computations using the same computer cluster as described in Section 9.2.

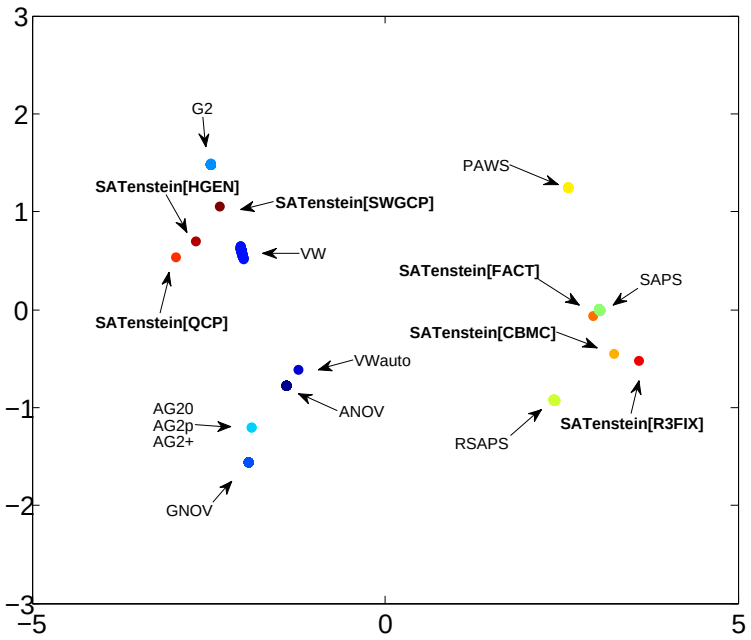



**Figure 9.3:** Visualization of the transformation costs in the design of 16 high-performance solvers (359 configurations) obtained via `Isomap`.

The final layout of similarities among 359 configurations (16 algorithms) is shown in Figure 9.3. Observe that in most cases the 50 different configurations for a given challenger solver were so similar that they mapped to virtually the same point in the graph.

As noted earlier, the distance between any two configurations shown in Figure 9.3 only approximates their true distance. In addition, the result of the visualization also depends on the number of configurations considered: adding an additional configuration may affect the position of many or all other configurations. Thus, before drawing further conclusions about the results illustrated in Figure 9.3, we validated the fidelity of the visualization to the original distance data. As can be seen from Figure 9.4, although `Isomap` tended to underestimate the true distances between configurations, there was a strong correlation between the computed and mapped distances (Pearson correlation coefficient: 0.93). Also, the mapping did a good job of preserving the relative ordering of the true distances between configurations (Spearman correlation coefficient 0.91)—in other words, distances that

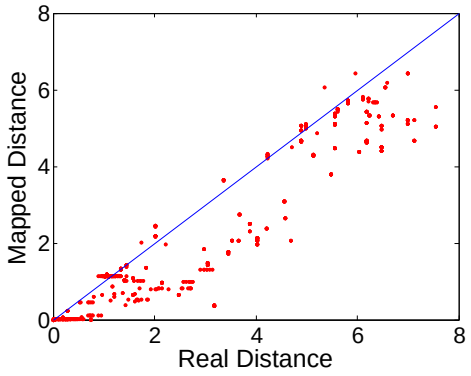



**Figure 9.4:** True *vs* mapped distances in Figure 9.3. The data points correspond to the complete set of SATenstein-LS[D] for all domains and all challengers with their default and domain-specific, optimized configurations.

appear similar in the 2D plot tend to correspond to similar true distances (and vice versa). Digging deeper, we confirmed that the challenger closest in Figure 9.3 to each given SATenstein-LS solver was indeed the one having the lowest true transformation cost. This was not true for the most distant challengers; however, we find this acceptable, since in the following, we are mainly interested in examining which configurations are similar to each other.

Having confirmed that our dimensionality reduction method is performing reliably, let us examine Figure 9.3 in more detail. Overall, and unsurprisingly, we first note that the transformation cost between two configurations in the design space is very weakly related to their performance difference (quantitatively, the Spearman correlation coefficient between performance difference (PAR-10 ratio) and configuration difference (transformation cost) was 0.25). As we suspected based on our manual examination of parameter configurations, each SATenstein-LS solver except SATenstein-LS[FAC] was quite different from every challenger. This provides further evidence that the superior performance of SATenstein-LS solvers is due to combining components from existing SLS algorithms in novel ways. Examining algorithms by type, we note that all dynamic local search algorithms are grouped together, on the right side of Figure 9.3; likewise, the al-

gorithms using adaptive mechanisms are grouped together at the bottom of Figure 9.3. SATenstein-LS solvers were typically more similar to each other than to challengers, and fell into two broad clusters. The first cluster also includes the SAPS variants (SAPS, RSAPS), while the second also includes G2 and VW. None of the SATenstein-LS solvers uses an adaptive mechanism to automatically adjust other parameters; in fact, as shown in Table 9.12, the same is true of the best performance-optimized challengers. This suggests that in many cases, contrary to common belief (see, e.g., [80, 135]) it may be preferable to expose parameters so they can be instantiated by sophisticated configurators rather than automatically adjusting them at running time using a simple adaptive mechanism.

We now consider benchmarks individually. For the FAC benchmark, SATenstein-LS[FAC] had similar performance to SAPS[FAC]; as seen in Figure 9.3, both solvers are structurally very similar as well. Overall, for the 'industrial' distributions, CBMC(SE) and FAC, dynamic local search algorithms often yielded the best performance amongst all challengers. Our automatically-constructed SATenstein-LS solvers for these two distributions are also dynamic local search algorithms. Due to the larger search neighbourhood and the use of clause penalties, dynamic local search algorithms are more suitable for solving industrial SAT instances, which often have some special global structure.

For R3SAT, a well-studied distribution, many challengers showed good performance (the top three challengers were VW, RSAPS, and SAPS). The performance of SATenstein-LS[R3SAT] is only slightly better than that of VW[R3SAT]. Figure 9.3 shows that SATenstein-LS[R3SAT] is a dynamic local search algorithm similar to RSAPS and SAPS.

For HGEN, even the best performance-optimized challengers, RSAPS[HGEN] and SAPS[HGEN], performed poorly. SATenstein-LS[HGEN] achieves more than 1,000-fold speedups against all challengers. Its configuration is far away from any dynamic local search algorithm (the best challengers), and closest to VW and G2.

For QCP, VW[QCP] does not reach the performance of SATenstein-LS[QCP], but significantly outperforms all other challengers. Our transformation cost analysis shows that VW is the closest neighbour to SATenstein-LS[QCP]. For SWGCP, many challengers achieve similar performance to SATenstein-LS[SWGCP].

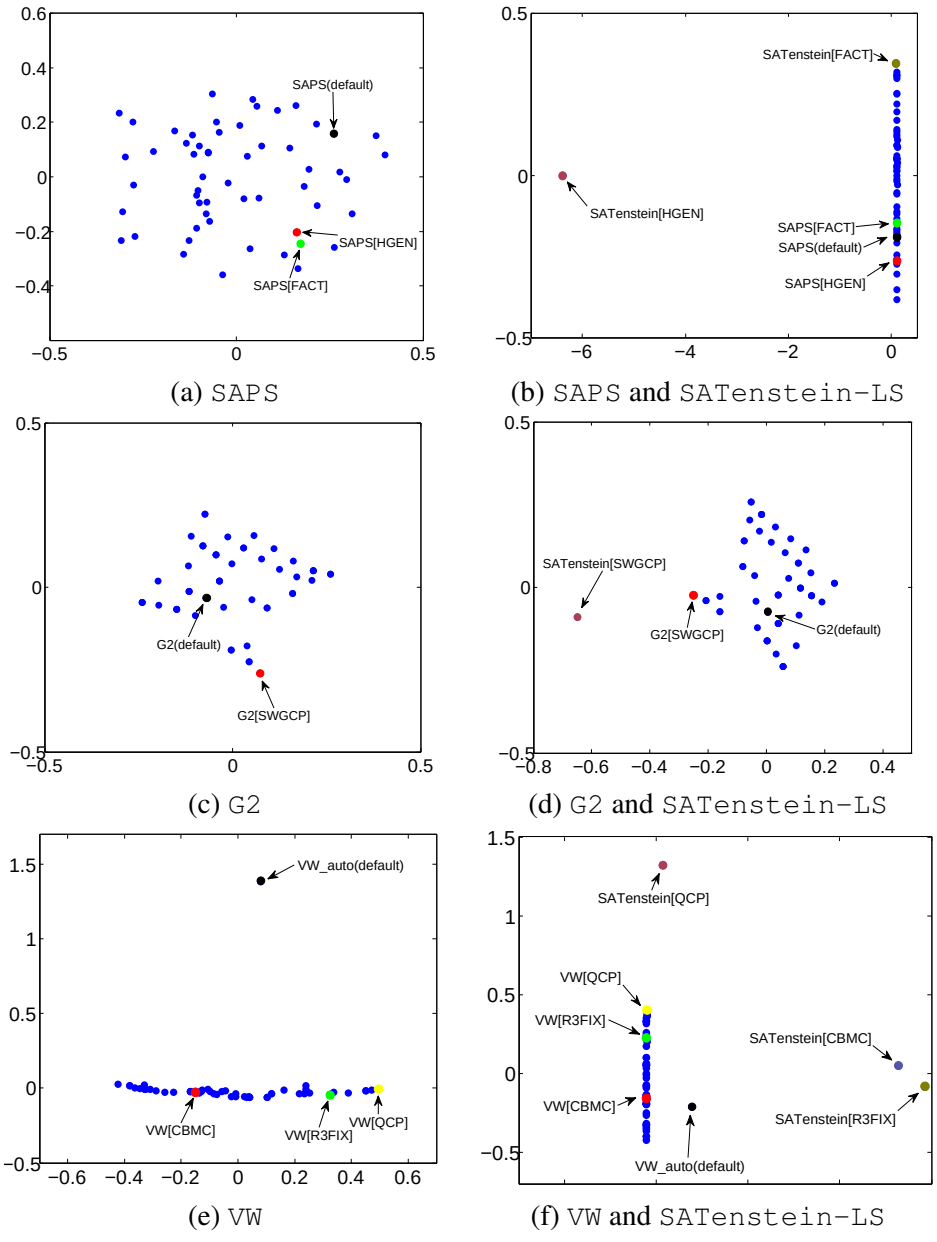
Figure 9.3 shows that SATenstein-LS[SWGCP] is close to G2[SWGCP], which is the best performing challenger on SWGCP.

### 9.4.3 Comparison to Configured Challengers

Since there are large performance gaps between default and configured challengers (as seen in Figure 9.2), we were also interested in the transformation cost between the configurations of individual challenger solvers. Recall that after configuring each challenger for each distribution, we found that SAPS was best on HGEN and FAC; G2 was best on SWGCP, and VW was best on CBMC(SE), QCP, and R3FIX. Figure 9.5:left visualizes the parameter spaces for each of these three solvers (43 random configurations + default configuration + 6 optimized configurations). Figure 9.5:right shows the same thing, but also adds the best SATenstein-LS configurations for each benchmark on which the challenger had top performance.

Examining these figures in the left column of Figure 9.5, we first note that the SAPS configurations optimized for FAC and HGEN are very similar but differ substantially from SAPS's default configuration. On SWGCP, the optimized configuration of G2 not only performed much better than the default but, as seen in Figure 9.5:middle-left, is also quite different. All three top-performing VW configurations are rather different from VW's default, and none of them uses the adaptive mechanism for choosing parameter *wpWalk*, *s*, and *c*. Since the parameter useAdaptiveMechanism is a level-1 parameter and many other parameters are conditionally dependent on it, the transformation costs between VW default and optimized configurations of VW are very large, due to the high relabelling cost for these nodes in our concept DAGs.

The right column of Figure 9.5 illustrates the similarity between optimized SATenstein-LS solvers and the best performing challenger for each benchmark. As previously noted, SATenstein[FAC] and SAPS[FACT] are not only very similar in performance, but also structurally similar. Likewise, SATenstein[SWGCP] is similar to G2SWGCP. On R3SAT, many challengers have similar performance. SATenstein[R3SAT] (PAR-10=1.11) is quite different from the best challenger VW[R3SAT] (PAR-10=1.26), but resembles SAPS[R3SAT] (PAR-10=1.53). For the three remaining benchmarks, SATenstein-LS solvers exhibited much better




(a) SAPS (b) SAPS and SATenstein-LS

(c) G2 (d) G2 and SATenstein-LS

(e) VW (f) VW and SATenstein-LS

**Figure 9.5:** The transformation costs of configuration of individual challengers and selected SATenstein-LS solvers. (a): SAPS (best on HGEN and FACT); (b): SAPS and SATenstein[HGEN, FACT]; (c): G2 (best on SWGCP); (d): G2 and SATenstein[SWGCP]; (e): VW (best on CBMC(SE), QCP, and R3FIX); (f): VW and SATenstein[CBMC, QCP, R3FIX].

performance than the best optimized challengers, and their configurations likewise differ substantially from the challengers' configurations.

As an aside, it might initially be surprising that qualitative features of the visualizations in Figures 9.5 appear to be absent from Figure 9.3. In particular, the sets of randomly sampled challenger configurations that are quite well-separated in Figure 9.5 are nearly collapsed into single points in Figure 9.3, although the scales are not vastly different. The reason for this lies in the fact that the 2D-mapping of the highly non-planar pairwise distance data performed by `Isomap` focuses on minimal overall distortion. For example, when visualizing the differences within a set of randomly sampled `SAPS` configurations (Figure 9.5 (a)), `Isomap` spreads these out into a cloud of points to represent their differences. However, the presence of a single `SATenstein-LS` configuration that has large transformation costs from all of these `SAPS` configurations forces `Isomap` to use one dimension to capture those differences, leaving essentially only one dimension to represent the much smaller differences between the `SAPS` configurations (Figure 9.5 (b)). Adding further very different configurations (as present in Figure 9.3) leads to mappings in which the smaller differences between configurations of the same challenger become insignificant.

## 9.5 Conclusions

In this chapter, we have proposed a new approach for designing heuristic algorithms based on (1) a framework that can flexibly combine components drawn from existing high-performance solvers, and (2) a powerful algorithm configuration tool for finding instantiations that perform well on given sets of instances. We have demonstrated the effectiveness of our approach by automatically constructing high-performance stochastic local search solvers for SAT. We have shown empirically that these automatically constructed SAT solvers outperform existing state-of-the-art solvers with manually and automatically optimized configurations on a range of widely studied distributions of SAT instances.

We have also proposed a new metric for quantitatively assessing the similarity between configurations for highly parametric solvers. We first introduce a data structure, concept DAGs, that preserves the internal hierarchical structure of pa-

rameters. We then estimate the similarity of two configurations as the transformation cost from one configuration to another. We have demonstrated that visualizations based on transformation costs can provide useful insights into similarities and differences between solver configurations. In addition, we believe that this metric could be useful for suggesting potential links between algorithm structure and algorithm performance. While this chapter only applied our metric to compare `SATenstein-LS` and several local search algorithms on SAT, we expect the same technique will be useful more broadly for comparing different algorithm designs that can be expressed within the same configuration space.

Our original inspiration comes from Mary Shelley's classic novel, Frankenstein. One important methodological difference is that we use automated methods for selecting components for our monster instead of picking them by hand. The outcomes are quite different. Unlike the tragic figure of Dr. Frankenstein, whose monstrous creature haunted him enough to quench forever his ambitions to create a 'perfect' human, we feel encouraged to unleash not only our new solvers, but also the full power of our automated solver-building process onto other classes of SAT benchmarks. Like Dr. Frankenstein, we find our creations somewhat monstrous, recognizing that the SATenstein solvers do not always represent the most elegant designs. Thus, desirable lines of future work include techniques for understanding importance of different parameters to achieving strong performance on a given benchmark; the extension of our solver framework with preprocessors; and the investigation of algorithm configuration procedures other than ParamILS in the context of our approach. Encouraged by the results achieved on SLS algorithms for SAT, we believe that the general approach behind `SATenstein-LS` is equally applicable to non-SLS-based solvers and to other combinatorial problems. Finally, we encourage participators from the SAT community to apply `SATenstein-LS` on their own problem distributions, and to extend `SATenstein-LS` with their own heuristics. Source code and documentation for our `SATenstein-LS` framework are freely available at http://www.cs.ubc.ca/labs/beta/Projects/SATenstein.

# Chapter 10

# Hydra: Automatic Configuration of Algorithms for Portfolio-Based Selection

This chapter introduces Hydra, a novel technique for combining automated algorithm configuration and portfolio-based algorithm selection, thereby realizing the benefits of both. Hydra automatically builds a set of solvers with complementary strengths by iteratively configuring new algorithms. It is primarily intended for use in problem domains for which an adequate set of candidate solvers does not already exist. Nevertheless, we tested Hydra on a widely studied domain, stochastic local search algorithms for SAT, in order to characterize its performance against a well-established and highly competitive baseline. We found that Hydra consistently achieves major improvements over the best existing individual algorithms, and always at least roughly matches—and indeed often exceeds—the performance of the best portfolios of these algorithms. [1]

For mixed integer programming problems, there are very few strong solvers, and state-of-the-art solvers are highly parameterized. Such observations render MIP a perfect case for applying techniques such as Hydra. We demonstrate how to improve Hydra to achieve strong performance for MIP based on single MIP

[1] This work is based on the joint work with Holger Hoos, and Kevin Leyton-Brown [212].

solver, IBM ILOG's CPLEX. By applying advanced algorithm selection technique and modifying Hydra's method for selecting candidate configurations, we show that Hydra dramatically improves CPLEX's performance for a variety of MIP benchmarks, as compared to ISAC [111], algorithm configuration alone, and CPLEX's default configuration. [2]

## 10.1 Hydra

Once state-of-the-art algorithm portfolios exist for a problem, such as the SATzilla portfolios for various categories of SAT instances, the question arises: how should new research aim to improve upon it? Inspired by the idea of "boosting as a metaphor for algorithm design" [128], we believe that algorithm design should focus on problems for which the existing portfolio performs poorly. In particular, [128] suggested to use sampling (with replacement) to generate a new benchmark distribution that will be harder for an existing portfolio, and for new algorithms to attempt to minimize average runtime on this benchmark. While we agree with the core idea of aiming explicitly to build algorithms that will complement a portfolio, we have come to disagree with its concrete realization as described most thoroughly by Leyton-Brown et al. (2009), realizing that average performance on a new benchmark distribution is not always an adequate proxy for the extent to which a new algorithm would complement a portfolio.

We note that a region of the original distribution that is exceedingly hard for all candidate algorithms can dominate the new distribution, leading to stagnation. A further problem is illustrated in the following, more complex example (due to Frank Hutter). Consider a uniform distribution over instance types $A$, $B$, and $C$. The current portfolio solves $C$ instances in 0.01 seconds, and $A$ and $B$ instances in 20 seconds each. The new distribution $S$ thus emphasizes instance types $A$ and $B$. There are three kinds of algorithms. $X$ algorithms solve $A$ instances in $0.1\pm\varepsilon$ seconds and $B$ instances in $100\pm\varepsilon$ seconds each, where $\varepsilon$ is a number between 0 and 0.01; the actual runtime varies randomly within this range across given algorithm–instance pairs. $Y$ algorithms solve $B$ instances in $0.1\pm\varepsilon$ seconds and $A$ instances

[2] This work is based on the joint work with Frank Hutter, Holger Hoos, and Kevin Leyton-Brown [213].

in $100\pm\varepsilon$ seconds each. $Z$ algorithms solve both $A$ and $B$ instances in $25\pm\varepsilon$ seconds each. All three algorithm types solve $C$ instances in $10\pm\varepsilon$ seconds each. The best average performance on $S$ will be achieved by some $Z$ algorithm, which we thus add to the portfolio. However, observe that this new $Z$ algorithm is dominated by the current portfolio. Thus our new distribution $S'$ will be the same as $S$. The process thus stagnates (endless algorithms of type $Z$ exist), and we never add any $X$ or $Y$ algorithm to the portfolio, although adding any pair of these would lead to improved overall performance.

To overcome these limitations, we introduce `Hydra`, a new method for automatically designing algorithms to complement a portfolio. This name is inspired by the Lernaean Hydra, a mythological, multi-headed beast that grew new heads for those cut off during its struggle with the Greek hero Heracles. `Hydra`, given only a highly parameterized algorithm and a set of instance features, automatically generates a set of configurations that form an effective portfolio. It thus does not require any domain knowledge in the form of existing algorithms or algorithm components that are expected to work well, and can be applied to any problem. `Hydra` is an *anytime algorithm*: it begins by identifying a single configuration with the best overall performance, and then iteratively adds algorithms to the portfolio. It is also able to drop previously added algorithms when they are no longer helpful.

The critical idea behind `Hydra` is that a new candidate algorithm should be preferred exactly to the extent that it improves upon the performance of a (slightly idealized) portfolio. `Hydra` is thus implemented by changing the performance measure given to the algorithm configuration. A candidate algorithm is scored with its actual performance in cases where it is better than the existing portfolio, but with *the portfolio's performance* in cases where it is worse. Thus an algorithm is not penalized for bad performance on instances for which it should not be selected, but is only rewarded to the extent that it outperforms the portfolio. The examples given earlier would be handled properly by this approach: the presence of intractable instances does not lead one to ignore performance gains elsewhere, while $X$ and $Y$ algorithms would be chosen in the first two iterations.

As shown in pseudocode, `Hydra` takes five inputs: a parameterized solver $s$, a set of training problem instances $I$, an algorithm configuration procedure $AC$ with a

performance metric $m$ to be optimized, and a procedure *PB* for building portfolio-based algorithm selectors.

In its first iteration, `Hydra` uses configurator *AC* to produce a configuration of $s$, dubbed $s_1$, that is optimized on training set $I$ according to performance metric $m$. Solver $s_1$ is then run on all instances of $I$ in order to collect data that can eventually be input to *PB*; runs performed during the earlier configuration process can be cached and reused as appropriate. We define portfolio $P_1$ as the portfolio that always selects $s_1$, and solver set $S_1$ as $\{s_1\}$.

Then, in each subsequent iteration $k \geq 2$, `Hydra` defines a modified performance metric $m_k$ as the better of the performance of the solver being assessed and the performance of the current portfolio, both measured according to performance metric $m$. The configurator *AC* is run to find a configuration $s_k$ of $s$ that optimizes $m_k$ on the entire training set $I$. As previously, the resulting solver is evaluated on the entire set $I$ and then added to the solver set $S$. We then use *PB* to construct a new portfolio $P_k$ from the given set of solvers. In each iteration of `Hydra`, the size of the candidate solver set $S_k$ grows by one; however, *PB* may drop solvers that do not contribute to the performance of portfolio $P_k$ (as in `SATzilla` [210]). Slightly modifying the second example provided earlier, if $Z$ algorithms have slightly *better* performance on $A$ and $B$ instances than the current portfolio, some $Z$ algorithm *will* be chosen in the first iteration. However, $X$ and $Y$ algorithms are chosen in the next two iterations, at which point the $Z$ algorithm will be dropped, because it is dominated by the pair of $X$ and $Y$ algorithms.

`Hydra` can be terminated using various criteria, such as a user-specified bound on the number of iterations and/or a total computation-time budget.

The algorithm configuration procedure *AC* used within `Hydra` must be able to deal efficiently with configurations having equal performance on some or all instances, because such configurations can be expected to be encountered frequently. (For example, all configurations dominated by portfolio $P_{k-1}$ will have equal performance under performance metric $m_k$.) It is also possible to exploit $m_k$ for computational gain when optimizing runtime (as we do in our experimental study below). Specifically, a run of $s$ on some instance $i \in I$ can be terminated during configuration once its runtime reaches portfolio $P_{k-1}$'s runtime on $i$. (Refer to the analogous discussion of capping in algorithm configuration by Hutter et al.

**Procedure** Hydra($s$, $I$, $AC$, $m$, $PB$)

**Input**: Parametric solver $s$; Instance set $I$;
Algorithm configurator $AC$;
Performance metric $m$;
Portfolio builder $PB$
**Output**: Portfolio $P$

$k := 1$; $m_1 = m$ ;

obtain a solver $s_1$ by running configurator $AC$ on parametric solver $s$ and instance set $I$ with performance metric $m_1$;

measure performance of $s_1$ on all instances in $I$, using performance metric $m$;

let $P_1$ by a portfolio that always selects $s_1$;

let $S_1 := \{s_1\}$;

**while** *termination condition not satisfied* **do**

  $k := k+1$;

  define performance metric $m_k$ as the better of the performance of the solver being assessed and the performance of portfolio $P_{k-1}$, both measured using performance metric $m$;

  obtain a new solver $s_k$ by running configurator $AC$ on parametric solver $s$ and instance set $I$ with performance metric $m_k$;

  measure performance of $s_k$ on all instances in $I$, using performance metric $m$;

  $S_k = S_{k-1} \cup \{s_k\}$;

  obtain new portfolio $P_k$ by running portfolio builder $PB$ on $S$;

**return** $P$

(2009).)

It should be noted that `Hydra` need not be started from an empty set of algorithms, or only consider one parameterized algorithm. For example, it is straightforward to initialize $S$ with existing state-of-the-art algorithms before running `Hydra`, or to optimize across multiple parameterized algorithms.

## 10.2 Hydra for SAT

`Hydra` offers the greatest potential benefit in domains where only one highly parameterized algorithm is competitive (e.g., certain distributions of mixed-integer programming problems), and the least potential benefit in domains where a wide variety of strong, uncorrelated solvers already exist. Nevertheless, we chose to evaluate `Hydra` on SAT—possibly the most extreme example of the latter category—effectively building a `SATzilla` of `SATensteins`. We did so for several rea-

sons. Most of all, to demonstrate the usefulness of the approach, we considered it important to work on a problem for which the state of the art is known to be very strong. SLS-based SAT algorithms have been the subject of a large and sustained research effort over the past two decades, and the success of SATzilla demonstrates that existing SAT algorithms can be combined together to form very strong portfolios. The criteria for success is thus set extremely high in this domain. Further, studying SLS for SAT also offered several pragmatic benefits: a wide variety of datasets exist and are agreed to be interesting; effective instance-based features are available; and SATenstein is a suitable configuration target. Finally, because SAT is an important problem, even small improvements are significant.

### 10.2.1 Experimental Setup

We chose inputs for Hydra to facilitate comparisons with past work, setting *s*, *I*, *AC*, and *m* as in Chapter 9, and taking *PB* from Chapter 7. Inputs *s*, *I* and *m* define the application context in which Hydra is run. In contrast, *AC* and *PB* are generic components; we chose these "off the shelf" and made no attempt to modify them to achieve domain-specific performance improvements. We do not expect that an end user would have to vary them either.

*Parametric Solver: SATenstein-LS* . As our parametric solver *s*, we chose SATenstein-LS, a generalized, highly parameterized stochastic local search (SLS) framework (Chapter 9).

*Instances* We investigated the effectiveness of Hydra on four distributions, drawing on well-known families of SAT instances. As no state-of-the-art SLS algorithms are able to prove unsatisfiability, we considered only satisfiable instances. We identified these by running all complete algorithms that had won a SAT competition category between 2002 and 2007 for one hour. First, the BM data set is constructed from 500 instances taken from each of the six distributions used by Chapter 9 (QCP, SWGCP, FACT, CBMC, R3FIX, and HGEN), split evenly into training and test sets. Second, the INDULIKE data set is a mixture of 500 instances from each of the CBMC and FACT distributions, again split evenly into training and

test sets. Third and fourth, the HAND and RAND data sets include all satisfiable instances from the *Random* and *Handmade* categories of the SAT Competitions held between 2002 and 2007; we split the data 1141:571 and 342:171 into training and test sets, respectively.

*Algorithm Configurator: FocusedILS* As our algorithm configurator *AC*, we chose the FocusedILS procedure from the ParamILS framework, version 2.3 [96]. ParamILS is able to deal with extremely large configuration spaces such as SATenstein-LS's, and indeed was the method used to identify the high-performing SATenstein-LS configurations mentioned previously. FocusedILS compares a new configuration with an incumbent by running on instances one at a time, and rejects the new configuration as soon as it yields weakly worse overall performance on the set of instances than the incumbent. As we expect many ties in Hydra's modified performance measures $m_k$, particularly in later iterations, we changed this mechanism in order that new configurations are rejected only once they yield strictly worse overall performance. We also modified FocusedILS to cap all runs at the corresponding runtime for the portfolio $P_{k-1}$, as discussed previously.

*Performance Metric: PAR* We followed the approach of Chapter 9, capping runs at 5 seconds and setting our performance metric $m$ to be Penalized Average Runtime–10 (PAR-10). We performed 10 independent FocusedILS runs on training data with different instance orderings and with a one-day time bound. We retained the parameter configuration that yielded the best PAR score on training data.

*Portfolio Builder: SATzilla* As our portfolio builder *PB* we used the SATzilla framework [210] based empirical hardness models.

We computed the same set of features as in Figure 3.1. For BM and INDULIKE, we only used 40 very efficiently computable features (with an average feature computation time of 0.04 seconds in both cases) since initial, exploratory experiments revealed that Hydra could achieve performance on the order of seconds on these data sets. For the same reason, we also reduced the time allowed for subset se-

lection on these distributions by a factor of 10, allowing time budgets taken from $\{0s, 0.2s, 0.5s, 1s\}$. For RAND and HAND, we used all features except the most expensive ones (LP-based and clause-graph-based features); the average feature computation times were 4.2 seconds and 4.9 seconds, respectively.

*Challengers* As previously explained, one reason that we studied SLS for SAT is that a wide variety of strong solvers exist for this domain. In particular, we identified 17 such algorithms, which we dubbed "challengers." Following Chapter 9, we included all 7 SLS algorithms that won a medal in any of the SAT Competitions between 2002 and 2007, and also 5 additional prominent high-performance algorithms. We also included the 6 SATenstein-LS configurations introduced in Chapter 9. While in some sense this set a high standard for Hydra (it had to compete against strong configurations of its own parametric solver) we included these configurations because they were shown to outperform the previous state of the art.

*Experimental Environment* We collected training data and performed ParamILS runs on two different compute clusters. The first had 55 dual 3.2GHz Intel Xeon machines with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1; the second cluster from Westgrid had 384 dual 3.0GHz Intel Xeon E5450 quad-core machines with 16GB of RAM running Red Hat Linux 4.1.2. Although the use of different machines added noise to the runtime observations in our training data, it had to be undertaken to leverage additional computational resources. To ensure that our results were meaningful, we gathered all test data using only the first cluster; all results reported in this chapter were collected using this data, and the data was used for no other purpose. Runtime was always measured as CPU time.

### 10.2.2 Experimental Results

To establish a baseline for our empirical evaluation, we first ran all 17 challenger algorithms on each of our test sets. The best-performing challengers are identified in the third column of Table 10.1, and their PAR-10 scores are shown in the fourth column. We also report the percentages of instances that each algorithm solved.

| Dataset | Metric | Best Challenger | Chall. Perf | Portf. 11-Chall. | Portf. 17-Chall. | Hydra[D,1] | Hydra[D,7] |
|---|---|---|---|---|---|---|---|
| BM | PAR Score | SATenstein-LS | 224.53 | 54.04 | 3.06 | 249.44 | 3.06 |
| | Solved (%) | [FACT] | 96.4 | 99.3 | 100 | 96.0 | 100 |
| INDULIKE | PAR Score | SATenstein-LS | 11.89 | 135.84 | 7.74 | 33.49 | 7.77 |
| | Solved (%) | [CBMC] | 100 | 98.1 | 100 | 100 | 100 |
| RAND | PAR Score | gNovelty$^+$ | 1128.63 | 897.37 | 813.72 | 1166.66 | 631.35 |
| | Solved (%) | | 81.6 | 85.5 | 86.9 | 80.8 | 89.8 |
| HAND | PAR Score | adaptG$^2$WSAT$_+$ | 2960.39 | 2670.22 | 2597.71 | 2915.22 | 2495.06 |
| | Solved (%) | | 50.9 | 55.8 | 56.9 | 51.7 | 58.7 |

**Table 10.1:** *Performance comparison between Hydra, SATenstein-LS, challengers, and portfolios based on 11 (without 6 SATenstein-LS solvers) and 17 (with 6 SATenstein-LS solvers) challengers. All results are based on 3 runs per algorithm and instance; an algorithm solves an instance if its median runtime on that instance is below the given cutoff time.*

We then used SATzilla to automatically construct portfolios, first from the 11 manually crafted challenger algorithms, and then from the full set of 17 challengers that also included the 6 SATenstein-LS solvers. As can be seen from column 6 of Table 10.1, the latter portfolios perform much better than the best individual challenger, and the same holds for the former, more limited portfolios (column 5) as compared to the best of their 11 handcrafted component solvers. As one would expect, the performance gain was particularly marked for instance set BM, which is highly heterogeneous. In all cases, the inclusion of the 6 SATenstein-LS solvers, which were derived by automatic configuration on the six instance distributions considered by [115], led to improved performance. While this was expected for BM and INDULIKE, which are combinations of the instance distributions for which the 6 SATenstein-LS solvers were built, we were more surprised to observe the same qualitative effect for RAND and HAND.

Column 7 (Table 10.1) shows the performance of the single SATenstein-LS configuration that was obtained in the initial phase of Hydra. Comparing these results to those the portfolio obtained after 7 iterations (column 8), we confirm that Hydra is able to automatically configure solvers to work well as components of a portfolio. Furthermore, in all cases the Hydra portfolio outperformed the portfolio of 11 challengers. The Hydra portfolio outperformed the portfolio of 17 challengers in RAND and HAND, and effectively tied with it in BM and INDULIKE. Note that these latter distributions are those for which SATenstein-LS solvers

were specifically built; indeed, we found that the 17-challenger portfolios relied very heavily on these solvers. Furthermore, we note that Chapter 9 devoted about 240 CPU days to the construction of the 6 SATenstein-LS solvers, while the construction of the entire Hydra[D,7] portfolio required only about 70 CPU days.

Overall, recall that the success of the challenger-based portfolios depends critically upon the availability of domain knowledge in the form of very strong solvers (some handcrafted, such as 11 of the challengers, and some constructed automatically based on clearly-delineated instance distributions, such as the 6 SATenstein-LS solvers). In contrast, Hydra always achieved equivalent or significantly better performance *without* relying on such domain knowledge.

Figure 10.1 shows the PAR-10 performance improvements achieved in each Hydra iteration, considering both training and test data for BM and INDULIKE. In all cases, test performance closely resembled training performance. Hydra's test performance improved monotonically from one iteration to the next. Furthermore, on BM, HAND and RAND, Hydra achieved better performance than the best challenger after at most two iterations. On INDULIKE, Hydra took five iterations to outperform the best challenger, SATenstein-LS[CBMC]. While this may appear surprising considering that the latter is a configuration of SATenstein-LS, it is attributed to much less CPU time for each Hydra iteration than the construction of SATenstein-LS[CBMC].

Figure 10.2 compares the test-set performance of Hydra[D,1] and Hydra[D,7] for BM and INDULIKE. (The plots for HAND and RAND are not shown here, but resemble the BM plot.) Note that Hydra[D,7] is substantially stronger than Hydra[D,1], particularly on hard instances. The fact that Hydra[D,1] on occasion outperforms Hydra[D,7] is due to the feature-based selection not always choosing the best solver from the given portfolio, and that the algorithms are randomized.

Table 10.2 shows, over each of the 7 iterations, the fraction of training instances solved by each Hydra portfolio component. Obviously, a total of $k$ solvers are available in each stage $k$. Note that solver subset selection does lead Hydra to exclude solvers from the portfolio; this transpires on RAND for example, where the third solver was dropped in iteration 7. Another interesting effect can be observed in iteration 3 on INDULIKE, where the second solver was effectively replaced by the third, whose instance share is marginally higher. Had we allowed the algorithm

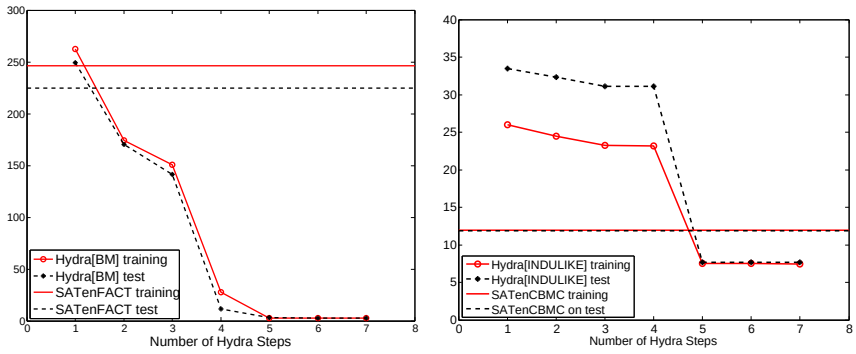



**Figure 10.1:** *`Hydra`'s performance progress after each iteration, for `BM` (left) and `INDULIKE` (right). Performance is shown in terms of PAR-10 score; the vertical lines represent the best challenger's performance for each data set.*

configurator to run longer in iteration 2, it would eventually have found this latter solver. The fact that it was found in the subsequent iteration illustrates `Hydra`'s ability to recover from insufficient allocation of runtime to the algorithm configurator. A similar phenomenon occurred in iterations 6 and 7 on `INDULIKE`. The solver found in iteration 6 turned out not to be useful at all, and was therefore dropped immediately; in the next round of algorithm configuration a useful solver was found. (However, we see in Figure 10.1 that the overall benefit derived from using this latter solver turned out to be quite small.) Finally, we note that for all four distributions, the `Hydra`[D,7] portfolios consisted of at least 5 solvers, each of which were executed on between 6.8 and 41.8% of the instances. This indicates that the individual solvers constructed by `Hydra` indeed worked well on sizeable subsets of our instance sets, without the explicit use of problem-dependent knowledge (such as instance features) for partitioning these sets.

## 10.3 Hydra for MIP

It is difficult to directly apply the original `Hydra` to the MIP domain, for two reasons. First, the data sets we are dealt in MIP tend to be highly heterogeneous; preliminary prediction experiments showed that `Hydra`'s linear regression mod-

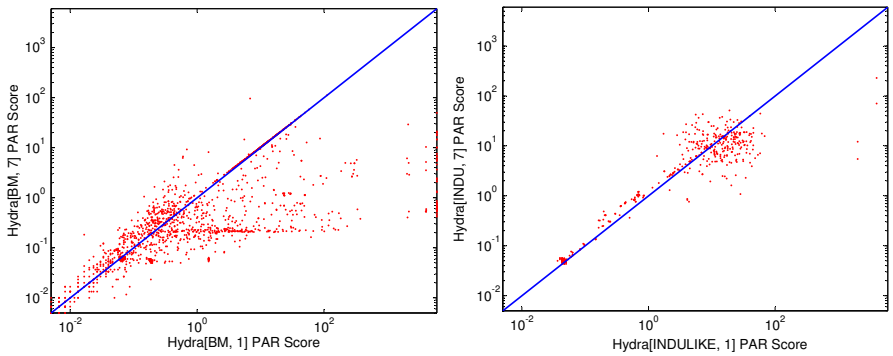



**Figure 10.2:** *Performance comparison between* `Hydra[D,7]` *and* `Hydra[D,1]` *on the test sets, for* `BM` *(left) and* `INDULIKE` *(right). Performance is shown in terms of PAR-10 score.*

| | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ |
|---|---|---|---|---|---|---|---|
| $P_1$ | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| $P_2$ | 45.1 | 54.9 | 0 | 0 | 0 | 0 | 0 |
| $P_3$ | 27.4 | 44.4 | 28.2 | 0 | 0 | 0 | 0 |
| $P_4$ | 18.1 | 31.0 | 21.6 | 29.4 | 0 | 0 | 0 |
| $P_5$ | 13.9 | 25.9 | 19.8 | 26.1 | 14.3 | 0 | 0 |
| $P_6$ | 12.5 | 22.9 | 16.8 | 23.2 | 13.2 | 11.5 | 0 |
| $P_7$ | 12.5 | 23.9 | 0 | 22.8 | 13.2 | 13.2 | 14.4 |

| | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ |
|---|---|---|---|---|---|---|---|
| $P_1$ | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| $P_2$ | 50.0 | 50.0 | 0 | 0 | 0 | 0 | 0 |
| $P_3$ | 49.0 | 0 | 51.0 | 0 | 0 | 0 | 0 |
| $P_4$ | 47.8 | 0 | 42.8 | 9.4 | 0 | 0 | 0 |
| $P_5$ | 35.8 | 0 | 42.6 | 9.4 | 12.2 | 0 | 0 |
| $P_6$ | 35.8 | 0 | 42.6 | 9.4 | 12.2 | 0 | 0 |
| $P_7$ | 31.2 | 0 | 41.8 | 9.2 | 11.0 | 0 | 6.8 |

**Table 10.2:** *The percentage of instances for each solver chosen by algorithm selection at each iteration for* `RAND` *(left) and* `INDULIKE` *(right).* $P_k$ *and* $s_k$ *are respectively the portfolio and algorithm obtained in iteration* $k$*.*

els were not robust for such heterogeneous inputs, sometimes yielding extreme mispredictions of more than ten orders of magnitude. Second, individual `Hydra` iterations can take days to run—even on a large computer cluster—making it difficult for the method to converge within a reasonable amount of time. (We say that `Hydra` has converged when substantial increases in running time cease to lead to significant performance gains.)

For MIP, we proposed two major improvements to `Hydra` that address both of these issues. First, we modify the model-building method used by the algorithm selector, using a classification procedure based on decision forests with a non-uniform loss function. Second, we modify `Hydra` to add multiple solvers in each iteration and to reduce the cost of evaluating these candidate solvers, speeding up convergence. We denote the original method as $\texttt{Hydra}_{\texttt{LR},1}$ ("LR" stands for linear

regression and "1" indicates the number of configurations added to the portfolio per iteration), the new method including only our first improvement as $\texttt{Hydra}_{\texttt{DF},1}$ ("DF" stands for decision forests), and the full new method as $\texttt{Hydra}_{\texttt{DF},k}$.

- Cost-sensitive Decision Forests for Algorithm Selection: Since 2011, the new versions of SATzilla are based on cost-sensitive classification models, in particular cost-sensitive decision forest (DF). DFs offer the promise of effectively partitioning the feature space into qualitatively different parts, particularly for heterogeneous benchmark sets. In contrast to clustering methods, DFs take runtime into account when determining that partitioning. Therefore, we adopt this new approach into Hydra and construct a cost-sensitive DF for every pair of configurations $(i, j)$ based on training data. The cost for a given instance $n$ for configuration pair $(i, j)$ is defined as the performance difference between $i$ and $j$. For a test instances, we apply each DF to vote for the stronger configuration and select the configuration with the most votes as the best algorithm for that instance.

- Speeding Up Convergence: Hydra uses an automated algorithm configurator as a subroutine, which is called in every iteration to find a configuration that augments the current portfolio as well as possible. Since algorithm configuration is a hard problem, configuration procedures are incomplete and typically randomized. As a single run of a randomized configuration procedure may not yield a high-performing parameter configuration, it is common practice to perform multiple runs in parallel and to use the configuration that performs best on the training set [93, 96, 97, 212].

  Here, we make two modifications to Hydra to speed up its convergence. First, in each iteration, we add $k$ promising configurations to the portfolio, rather than just the single best. If algorithm configuration runs were inexpensive, this modification to Hydra would not help: additional configurations could always be found in later iterations, if they indeed complemented the portfolio at that point. However, when each iteration must repeatedly solve many difficult MIP instances, it may be impossible to perform more than a small number of Hydra iterations within any reasonable amount of time, even when using a computer cluster. In such a case, when many good (and

rather different) configurations are found in an iteration, it can be wasteful to retain only one of these.

Our second change to Hydra concerns the way that the 'best' configurations returned by different algorithm configuration runs are identified. $\texttt{Hydra}_{\texttt{DF},1}$ determines the 'best' of the configurations found in a number of independent configurator runs by evaluating each configuration on the full training set and selecting the one with best performance. This evaluation phase can be very costly: for example, if we use a cutoff time of 300 seconds per run during training and have 1 000 instances, then computing the training performance of each candidate configuration can take nearly four CPU days. Therefore, in $\texttt{Hydra}_{\texttt{DF},k}$, we select the configuration for which the configuration procedure's internal estimate of the average performance improvement over the existing portfolio is largest. This alternative is computationally cheap: it does not require any evaluations of configurations beyond those already performed by the configurator. However, it is also potentially risky as different configurator runs typically use the training instances in a different order and evaluate configurations using different numbers of instances. It is thus possible that the configurator's internal estimate of improvement for a parameter configuration is high, but that it turns out to not help for instances the configurator has not yet used. Fortunately, adding $k$ parameter configurations to the portfolio in each iteration mitigates this problem: if each of the $k$ selected configurations has independent probability $p$ of yielding a poor configuration, the probability of all $k$ configurations being poor is only $p^k$.

### 10.3.1 Experimental Setup

While the improvements to Hydra presented were motivated by MIP, they can nevertheless be applied to any domain. The parameterized solver, instance benchmark and performance metric define the application context in which Hydra is run. In contrast, the automated algorithm configuration tool and portfolio builder are generic components.

*CPLEX Parameters.* Out of CPLEX 12.1's 135 parameters, we selected a subset of 74 parameters to be optimized. These are the same parameters considered in [97], minus two parameters governing the time spent for probing and solution polishing. (These led to problems when the captime used during parameter optimization was different from that used at test time.) We were careful to keep all parameters fixed that change the problem formulation (e.g., parameters such as the optimality gap below which a solution is considered optimal). The 74 parameters we selected affect all aspects of CPLEX. They include 12 preprocessing parameters; 17 MIP strategy parameters; 11 parameters controlling how aggressively to use which types of cuts; 8 MIP "limits" parameters; 10 simplex parameters; 6 barrier optimization parameters; and 10 further parameters. Most parameters have an "automatic" option as one of their values. We allowed this value, but also included other values (all other values for categorical parameters, and a range of values for numerical parameters). Exploiting the fact that 4 parameters were conditional on others taking certain values, they gave rise to $4.75 \times 10^{45}$ distinct parameter configurations.

*MIP Benchmark Sets.* Our goal was to obtain a MIP solver that works well on heterogenous data. Thus, we selected four heterogeneous sets of MIP benchmark instances, composed of many well studied MIP instances. They range from a relatively simple combination of two homogenous subsets (CL∪REG) to heterogenous sets using instances from many sources (e.g., , MIX). While previous work in automated portfolio construction for MIP [111] has only considered very easy instances (ISAC(new) with a mean CPLEX default runtime below 4 seconds), our three new benchmarks sets are much more realistic, with CPLEX default runtimes ranging from seconds to hours. We split these instances 50:50 into training and test sets except for ISAC(new), where we divided the 276 instances into a new training set of 184 and a test set of 92 instances. Due to the small size of the data set, we performed this in a stratified fashion, first ordering the instances based on CPLEX default runtime and then picking every third instance for the test set.

We used all 143 MIP features from Figure 3.2 including 101 features from [90, 111, 130] and 42 new probing features. In our feature computation, we used a 5-second cutoff for computing probing features. We omitted these probing features

(only) for the very easy ISAC(new) benchmark set.

*Algorithm Configurator: FocusedILS.* For algorithm configuration we used ParamILS version 2.3.4 with its default instantiation of FocusedILS with adaptive capping [96]. We always executed 25 parallel configuration runs with different random seeds with a 2-day cutoff. (Running times were always measured using CPU time.) During configuration, the captime for each CPLEX run was set to 300 seconds, and the performance metric was penalized average runtime (PAR-10). For testing, we used a cutoff time of 3 600 seconds.

*Portfolio Builder: SATzilla based on cost-sensitive decision forests.* We used the Matlab version R2010a implementation of cost-sensitive decision trees as described in Chapter 7. For any pair of algorithms $(i, j)$, a cost-sensitive decision forest was built with 99 trees. Therefore, $i$ receives a vote from $j$ if more than 44 trees predict it being "better". The algorithm with the most votes is selected as the best algorithm for a given instance. Ties are broken by only counting the votes from those decision forests that involve algorithms which received equal votes; further ties are broken randomly.

*Experimental Environment.* All of our experiments were performed on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1.

*Computational Cost.* The total running time for the various Hydra procedures was often dominated by the time required for running the configurator and therefore turned out to be approximately proportional to the number of Hydra iterations performed. Each iteration required 50 CPU days for algorithm configuration, as well as validation time to (1) select the best configuration in each iteration (only for $\texttt{Hydra}_{\texttt{LR},1}$ and $\texttt{Hydra}_{\texttt{DF},1}$); and (2) gather performance data for the selected configurations. Since $\texttt{Hydra}_{\texttt{DF},4}$ selects 4 solvers in each iteration, it has to gather performance data for 3 additional solvers per iteration (using the same captime of 3 600 seconds), which roughly offsets its savings due to ignoring the

validation step. Using the format ($\texttt{Hydra}_{\texttt{DF},1}$, $\texttt{Hydra}_{\texttt{DF},4}$), the overall runtime requirements in CPU days were as follows: (366,356) for CLUREG; (485, 422) for CLUREGURCW; (256,263) for ISAC(new); and (274,269) for MIX. Thus, the computational cost for each iteration of $\texttt{Hydra}_{\texttt{LR},1}$ and $\texttt{Hydra}_{\texttt{DF},1}$ was similar.

### 10.3.2 Experimental Results

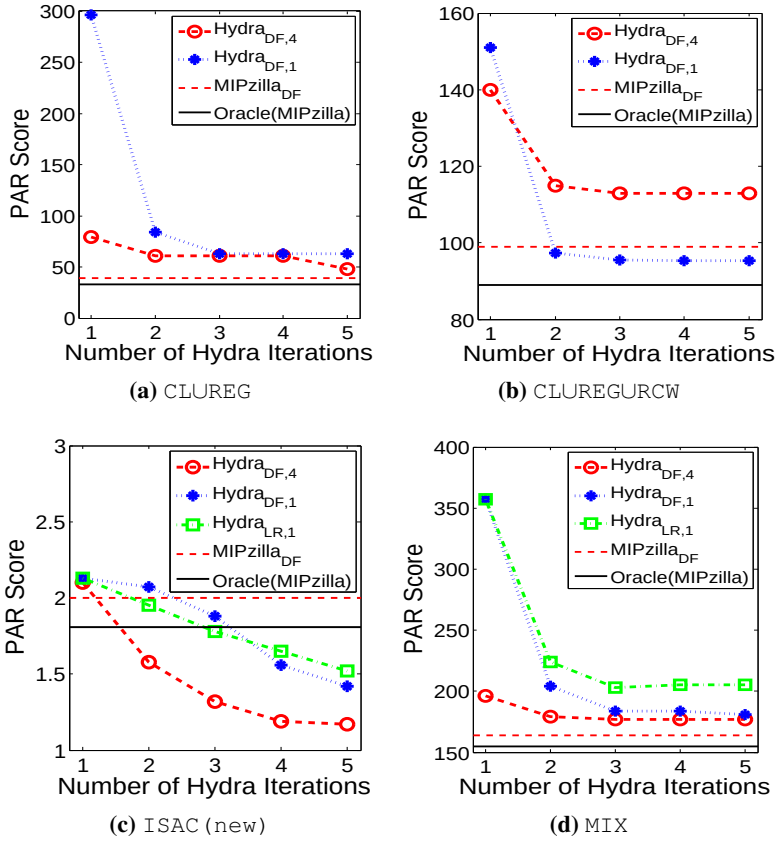
We evaluated our full $\texttt{Hydra}_{\texttt{DF},4}$ approach for MIP; on all four MIP benchmarks, we compared it to $\texttt{Hydra}_{\texttt{DF},1}$, to the best configuration found by ParamILS, and to the CPLEX default. For ISAC(new) and MIX we also assessed $\texttt{Hydra}_{\texttt{LR},1}$. We did not do so for CLUREG and CLUREGURCW because they are relatively simpler and we expected the DF and LR models to perform almost identically.

Table 10.3 presents these results. First, comparing $\texttt{Hydra}_{\texttt{DF},4}$ to ParamILS alone and to the CPLEX default, we observed that $\texttt{Hydra}_{\texttt{DF},4}$ achieved dramatically better performance, yielding between 2.52-fold and 8.83-fold speedups over the CPLEX default and between 1.35-fold and 2.79-fold speedups over the configuration optimized with ParamILS in terms of average runtime. Note that (likely due to the heterogeneity of the data sets) the built-in CPLEX self-tuning tool was unable to find any configurations better than the default for any of our four data sets. Compared to $\texttt{Hydra}_{\texttt{LR},1}$, $\texttt{Hydra}_{\texttt{DF},4}$ yielded a 1.3-fold speedup for ISAC(new) and a 1.5-fold speedup for MIX. $\texttt{Hydra}_{\texttt{DF},4}$ also typically performed better than our intermediate procedure $\texttt{Hydra}_{\texttt{DF},1}$, with speedup factors up to 1.21 (ISAC(new)). However, somewhat surprisingly, it actually performed worse for one distribution, CLUREGURCW. We analyzed this case further and found that in $\texttt{Hydra}_{\texttt{DF},4}$, after iteration three ParamILS did not find any configurations that would further improve the portfolio, even with a perfect algorithm selector. This poor ParamILS performance could be explained by the fact that Hydra's dynamic performance metric only rewarded configurations that made progress on solving some instances better; almost certainly starting in a poor region of configuration space, ParamILS did not find configurations that made progress on *any* instances over the already strong portfolio, and thus lacked guidance towards better regions of configuration space. We believed that this problem could be addressed by means of better configuration procedures in the future.

| DataSet | Solver | Train (cross valid.) | | Test | |
|---|---|---|---|---|---|
| | | Time | PAR (Solved) | Time | PAR (Solved) |
| CL UREG | Default | 424 | 1687 (96.7%) | 424 | 1493 (96.7%) |
| | ParamILS | 145 | 339 (99.4%) | 134 | 296 (99.5%) |
| | $\texttt{Hydra}_{\texttt{DF},1}$ | 64 | 97 (99.9%) | 63 | 63 (**100%**) |
| | $\texttt{Hydra}_{\texttt{DF},4}$ | **42** | **42** (**100%**) | **48** | **48** (**100%**) |
| | MIPzilla | **40** | **40** (**100%**) | **39** | **39** (**100%**) |
| | Oracle (MIPzilla) | **33** | **33** (**100%**) | **33** | **33** (**100%**) |
| CL UREG URCW | Default | 405 | 1532 (96.5%) | 406 | 1424 (96.9%) |
| | ParamILS | 148 | 148 (**100%**) | 151 | 151 (**100%**) |
| | $\texttt{Hydra}_{\texttt{DF},1}$ | **89** | **89** (**100%**) | **95** | **95** (**100%**) |
| | $\texttt{Hydra}_{\texttt{DF},4}$ | 106 | 106 (**100%**) | 112 | 112 (**100%**) |
| | MIPzilla | **99** | **99** (**100%**) | **99** | **99** (**100%**) |
| | Oracle (MIPzilla) | **89** | **89** (**100%**) | **89** | **89** (**100%**) |
| ISAC (new) | Default | 3.98 | 3.98 (**100%**) | 3.77 | 3.77 (**100%**) |
| | ParamILS | 2.06 | 2.06 (**100%**) | 2.13 | 2.13 (**100%**) |
| | $\texttt{Hydra}_{\texttt{LR},1}$ | 1.67 | 1.67 (**100%**) | 1.52 | 1.52 (**100%**) |
| | $\texttt{Hydra}_{\texttt{DF},1}$ | 1.2 | 1.2 (**100%**) | 1.42 | 1.42 (**100%**) |
| | $\texttt{Hydra}_{\texttt{DF},4}$ | **1.05** | **1.05** (**100%**) | **1.17** | **1.17** (**100%**) |
| | MIPzilla | 2.19 | 2.19 (**100%**) | 2.00 | 2.00 (**100%**) |
| | Oracle (MIPzilla) | 1.83 | 1.83 (**100%**) | 1.81 | 1.81 (**100%**) |
| MIX | Default | 182 | 992 (97.5%) | 156 | 387 (99.3%) |
| | ParamILS | 139 | 717 (98.2%) | 126 | 357 (99.3%) |
| | $\texttt{Hydra}_{\texttt{LR},1}$ | 74 | 74 (**100%**) | 90 | 205 (**99.6%**) |
| | $\texttt{Hydra}_{\texttt{DF},1}$ | 60 | 60 (**100%**) | 65 | 181 (**99.6%**) |
| | $\texttt{Hydra}_{\texttt{DF},4}$ | **53** | **53** (**100%**) | **62** | **177** (**99.6%**) |
| | MIPzilla | **48** | **48** (**100%**) | **48** | **164** (**99.6%**) |
| | Oracle (MIPzilla) | **34** | **34** (**100%**) | **39** | **155** (**99.6%**) |

**Table 10.3:** *Performance (average runtime and PAR in seconds, and percentage solved) of* $\texttt{Hydra}_{\texttt{DF},4}$, $\texttt{Hydra}_{\texttt{DF},1}$ *and* $\texttt{Hydra}_{\texttt{LR},1}$ *after 5 iterations.*

Figure 10.3 shows the test performance the different Hydra versions achieved as a function of their number of iterations, as well as the performance of the MIPzilla portfolios we built manually. When building these MIPzilla portfolios for CLUREG, CLUREGURCW, and MIX, we exploited ground truth knowledge about the constituent subsets of instances, using a configuration optimized specifically for each of these subsets. As a result, these portfolios yielded very strong performance. Although our various Hydra versions did not have access to this ground truth knowledge, they still roughly matched MIPzilla's performance (indeed, $\texttt{Hydra}_{\texttt{DF},1}$ outperformed MIPzilla on CLUREG). For ISAC(new), our base-




**(a)** CLUREG **(b)** CLUREGURCW

**(c)** ISAC(new) **(d)** MIX

**Figure 10.3:** *Performance per iteration for* $Hydra_{DF,4}$, $Hydra_{DF,1}$ *and* $Hydra_{LR,1}$, *evaluated on test data.*

line MIPzilla portfolio used CPLEX configurations obtained by ISAC [111]; all Hydra versions clearly outperformed MIPzilla in this case, which suggests that its constituent configurations are suboptimal. For ISAC(new), we observed that for (only) the first three iterations, $Hydra_{LR,1}$ outperformed $Hydra_{DF,1}$. We believed that this occurred because in later iterations the portfolio had stronger solvers, making the predictive models more important. We also observed that $Hydra_{DF,4}$ consistently converged more quickly than $Hydra_{DF,1}$ and $Hydra_{LR,1}$. While $Hydra_{DF,4}$ stagnated after three iterations for data set CLUREGURCW (refer to our prior discussion), it achieved the best performance at every given point in time for the three other data sets. For ISAC(new), $Hydra_{DF,1}$ did not converge after 5 iterations, while $Hydra_{DF,4}$ converged after 4 iterations and achieved bet-

ter performance. For the other three data sets, $\texttt{Hydra}_{\texttt{DF},4}$ converged after two iterations. The performance of $\texttt{Hydra}_{\texttt{DF},4}$ after the first iteration (i.e., with 4 candidate solvers available to the portfolio) was already very close to the performance of the best portfolios for MIX and CL∪REG.

We spent a tremendous amount of effort attempting to compare $\texttt{Hydra}_{\texttt{DF},4}$ with ISAC [111], since ISAC is also a method for automatic portfolio construction and was previously applied to a distribution of MIP instances. ISAC's authors supplied us with their training instances and the CPLEX configurations their method identified, but are generally unable to make their code available to other researchers and, as mentioned previously, were unable to recover their test data. We therefore compared $\texttt{Hydra}_{\texttt{DF},4}$'s and ISAC's relative speedups over the CPLEX default (thereby controlling for different machine architectures) on their training data. We note that $\texttt{Hydra}_{\texttt{DF},4}$ was given only 2/3 as much training data as ISAC (due to the need to recover a test set from [111]'s original training set); the methods were evaluated using only the original ISAC training set; the data set is very small, and hence high-variance; and all instances were quite easy even for the CPLEX default. In the end, $\texttt{Hydra}_{\texttt{DF},4}$ achieved a 3.6-fold speedup over the CPLEX default, as compared to the 2.1-fold speedup reported in [111].

As shown in Figure 10.3, all versions of Hydra performed much better than a MIPzilla portfolio built from the configurations obtained from ISAC's authors for the ISAC(new) dataset. In fact, even a perfect oracle of these configurations only achieved an average runtime of 1.82 seconds, which is a factor of 1.67 slower than $\texttt{Hydra}_{\texttt{DF},4}$.

## 10.4 Conclusions

In this chapter, we introduced Hydra, a new automatic algorithm design approach that combines portfolio-based algorithm selection with automatic algorithm configuration. We applied Hydra to SAT, a particularly well-studied and challenging problem domain, producing high-performance portfolios based only on a single highly parameterized SLS algorithm, SATenstein-LS. Our experimental results on widely-studied SAT instances showed that Hydra significantly outperformed 17 state-of-the-art SLS algorithms. Hydra reached, and in two of four cases ex-

ceeded, the performance of portfolios that used all 17 challengers as candidate solvers, 6 of which had been configured automatically using domain knowledge about specific types of SAT instances. At the same time, the total CPU time used by Hydra to reach this performance level for each distribution was less than a third of that used for configuring the 6 automatically-configured challengers.

We also showed how to extend Hydra to achieve strong performance for heterogeneous MIP distributions, outperforming CPLEX's default, ParamILS alone, ISAC and the original Hydra approach. This was accomplished by using a cost-sensitive classification model for algorithm selection (which also led to performance improvements in SATzilla), along with improvements to Hydra's convergence speed. We expect that $\texttt{Hydra}_{\texttt{DF},k}$ can be further strengthened by using improved algorithm configurators, such as model-based procedures (e.g., SMAC [99]). Overall, the availability of effective procedures for constructing portfolio-based algorithm selectors, such as our new Hydra, should encourage the development of highly parametrized algorithms for other prominent NP-hard problems in AI, such as planning and CSP.

# Chapter 11

# Conclusion and Future Work

Computationally hard problems play a key role in many practical applications, including formal verification, planning and scheduling, resource allocation and management. Even though theoretically, no worst-case polynomial time algorithm exists, heuristic methods are able to solve large problems effectively in practice. However, designing a high-performance heuristic solver is not an easy task. Traditionally, algorithm developers manually explore the combinations of algorithmic components and empirically evaluate them on small benchmarks. Due to the nature of NP-completeness, it is often the case that the developed solver is only good on certain types of benchmarks. Given a new benchmark, the whole difficult, tedious and time-consuming process needs to be repeated again.

Motivated by an increasing demand for high-performance solvers for difficult combinatorial problems in practical applications, by the desire to reduce the human effort required for building such algorithms, and by an ever-increasing availability of cheap computing power that can be harnessed for automating parts of the algorithm design process, this thesis leveraged rigorous statistical methods to study such computationally hard problems. My work has already displayed great impact in many areas of research and development. The sets of instance characteristics we proposed were widely used for obtaining insights into understanding the hardness of problem instances and designing algorithms for solving them (see, e.g., [111, 112, 140]). The constructed statistical methods can characterize algorithm runtime (even satisfiability status for NP-complete decision problems) with high

levels of confidence. My automated algorithm design approaches have led to substantial improvements for solving a range of hard computational problems.

## 11.1 Statistical Models of Instance Hardness and Algorithm Performance

Traditional notions of complexity cannot adequately explain the strong performance of heuristic algorithms; empirical methods are often the only practical means for assessing and comparing algorithms' performance. Through adapting supervised machine learning techniques, we constructed statistical models that predict solvers' performance without actually running them.

Inspired by the success of Leyton-Brown et al. [127] that predicted an algorithm's runtime using so-called empirical hardness models (EHMs), my work made significant advances on improving prediction accuracy. Firstly, we extended the feature set that characterizes propositional satisfiability (SAT) instances. In addition, we introduced features for other type of NP-complete problems such as the travelling salesman problem and the mixed integer programming problem (Chapter 3). Such features proved to be informative and have been widely appropriated by other research groups [111]. Secondly, we used new statistical models (e.g., Gaussian processes, random forests [101], hierarchical hardness models [208]), substantially improving prediction accuracies over previous linear regression models (Chapter 5, Chapter 6). Thirdly, we showed that EHMs were not limited to predicting the runtime of an algorithm, and instead can predict arbitrary user-defined measures of algorithm performance (e.g., penalized runtime or some arbitrary performance score). Finally, we showed that other statistical models (classification) could be constructed to predict the solution of NP-complete decision problems (Chapter 4). My classifiers achieved high accuracy in predicting the satisfiability status of SAT instances ranging from randomly generated to industrial; the classifications were sufficiently accurate to help make a better prediction of an algorithm's performance [208].

Overall, the extensive experimental study showed that the regression models achieved good, robust performance in predicting algorithms' performance. These models are useful for algorithm analysis, scheduling, algorithm portfolio construc-

tion, automated algorithm configuration, and other applications. The classification models were able to predict the satisfiability status of SAT instances with high accuracy even for uniform random 3-SAT instances from the solubility phase transition (Chapter 4). Further study on this benchmark showed that one could build a three-leaf decision tree trained with very small instances and only two features that achieved good classification accuracies across a wide range of instance sizes.

## 11.2 Portfolio-based Algorithm Selection

Heuristic algorithms are capable of handling very large instances, but they often only perform well on certain types of instances. Therefore, practitioners confront a potentially difficult algorithm selection problem: given a new instance, which algorithm should be used in order to optimize some performance objective. We developed a portfolio-based algorithm selector, `SATzilla`, which solved the algorithm selection problem automatically based on statistical models introduced in the previous section. By exploiting the complementary strengths of different SAT solvers, `SATzilla` won more than 10 medals in 2007, 2009 SAT competitions [124], and the 2012 SAT challenge [13]. It encouraged the development of many other portfolio-based solvers, and represented state-of-the-art in SAT solving for many years.

The goal of algorithm selection is to find a mapping from instances to algorithms that optimizes some performance metric. `SATzilla` approached this problem by using EHMs to predict the solvers' performance. For a new instance, it first predicts the performance of each solver based on instance features, then picks the one that is predicted to be the best. Several new techniques were introduced in Chapter 7 to improve the robustness of `SATzilla`, such as pre-solvers, backup solvers, and solver subset selection [209, 210]. These components played important roles on the success of `SATzilla` and became standard techniques widely used in constructing portfolio-based algorithm selectors. Recently, we improved `SATzilla`'s performance even further by adapting a new cost-sensitive classification technique that punishes misclassification in direct proportion to impact on portfolio performance (Chapter 7). The new `SATzilla` won the 2012 SAT challenge.

The general framework of SATzilla can be applied to any problem domain. In Chapter 10.3, we showed that MIPzilla (SATzilla for MIP) also achieved state-of-the-art performance on solving mixed integer programming problem.

In addition to its state-of-the-art performance, portfolio-based algorithm selection is useful for evaluating solver contributions. By omitting a solver from the portfolio, one can measure the contribution of this solver by computing SATzilla's performance difference with and without it. In Chapter 8, we showed that solvers that exploit novel strategies proved more valuable than those that exhibited the best overall performance.

## 11.3 Automatically Building High-performance Algorithms from Components

Designing high-performance heuristic algorithms for solving NP-complete problems is a time-consuming task even for domain experts. The resulting algorithm may not meet users' requirements due to 1) absence of the optimal combination of heuristics and 2) lack knowledge of user's benchmarks or performance metric. We solved the first problem by designing a generalized, highly parameterized algorithm with many different promising heuristics. Furthermore, the choices and behaviors of heuristics are often controlled by a large set of parameters. Therefore, this approach removes the burden of making early design choices without knowing the interaction of multiple heuristics and encourages the designer to consider many alternative designs from existing algorithms in addition to novel mechanisms. To better meet the requirements of end users, we used automated algorithm configuration tools [96] that optimized solver parameters given any benchmark and performance metric.

This general approach could be applied to many domains with its effectiveness demonstrated on the domain of SAT (Chapter 9). By taking components from 25 local search algorithms, we built a highly parameterized local search algorithm, SATenstein, that could be instantiated as $2.01 \times 10^{14}$ different solvers. Most of these instantiations had never been previously studied. Given a benchmark and a performance metric, we applied a black-box automated algorithm configurator to optimize SATenstein's parameters. Empirical evidence demonstrates that

the automatically constructed SATenstein outperformed existing state-of-the-art solvers with manually and automatically tuned configurations in several widely studied SAT benchmarks. In addition, we have proposed a new data structure, concept DAG, to represent the parameter configurations, and defined a novel similarity metric for comparing different configurations based the transformation cost between concept DAGs. The visualization of these similarity measure provided useful insights into algorithm design. For example, contrary to common belief (see, e.g., [80, 135]), it is preferable to expose parameters so they can be instantiated by automated configurators rather than adjusting them at running time using a simple adaptive mechanism.

## 11.4 Automatically Configuring Algorithms for Portfolio-Based Selection

In a problem domain with only one or a few high-performance parameterized algorithms, it would be difficult to construct a portfolio-based algorithm selector due to the small number of candidates. An automated algorithm configuration produces a single algorithm, which could achieve high performance overall, but may perform badly on many individual instances. To overcome these problems, we developed a new automated algorithm design approach, Hydra, that combines the strength of algorithm selection and algorithm configuration, and achieved state-of-the-art performance on SAT (Chapter 10.2) and MIP (Chapter 10.3) with a single parameterized algorithm.

Once a portfolio exists for a domain, how should new research aim to improve up it? Hydra approaches this question by adapting the concept of boosting which focuses on problems that are handled poorly with the current portfolio. Hydra is implemented by iteratively changing the performance metric given to the algorithm configurator. Such a metric emphasizes the potential marginal contribution of a new configuration to the existing portfolio. In each iteration, one algorithm (configuration) is added into the candidate algorithm set to improve the performance of the current portfolio. With a single parameterized solver, Hydra reached, and often exceeded, the performance of portfolios that used the state-of-the-art solvers as candidate solvers (including solvers configured using domain knowledge of spe-

cific types of instances). Later in Chapter 10.3, we further improved Hydra using a more advanced algorithm selection technique, along with improvements for speedup convergence. The resulting $\texttt{Hydra}_{\texttt{DF},k}$ achieved strong performance for heterogeneous MIP distributions, outperforming CPLEX's default, configurations from ParamILS alone , and ISAC.

Since Hydra requires only very little domain knowledge (one parameterized algorithm and an instance feature extractor), it is extremely attractive in cases of new problem domains. It also encourages the development of highly parameterized algorithms for other prominent NP-complete problems in AI, such as planning and CSP.

## 11.5 Future Research Directions

I am rather interested in studying and solving problems that have a substantial impact on society and industry, such as computational sustainability, bioinformatics, hardware and software verification, and other problems pertinent to industrial applications. I firmly believe that meta-algorithmic techniques are the future in solving hard computational problems in a broad range of areas. Therefore, I plan to continue developing new techniques and am looking forward to collaboration with domain experts in a variety of areas. Evidenced by the successes on SAT and MIP, I believe that my research on automated algorithm design will generate state-of-the-art algorithms for problems in real world applications.

**Informative Features**. One challenge in applying meta-algorithmic techniques on practical applications is to discover a set of features that is both easy to compute and yet correlate well with an algorithm's performance. Although our feature sets (for SAT, MIP, and TSP) have proven to be informative and efficient, more powerful features can be obtained by using heuristic information from new algorithms or intuition from domain experts. For example, the distribution of flip counts over variables (based on local search probing) could be used to indicate the number of local minima.

**More Applications**. Although my proposed automated algorithm design approaches have been applied to SAT and MIP, they can be applied to any computational hard problems. I continually seek more applications, for example, building

portfolio-based algorithm selectors to solve protein folding problems, constructing highly parameterized complete solvers for SAT and optimizing them by automated algorithm configuration tools.

**Integrating Strong Presolving Techniques**. In `SATzilla`'s presolving phase, two algorithms are run for a short duration of time with the goal of solving easy instances. Recently there were many advances in optimizing solver schedule. `3S` [112] introduced a new approach for presolving by constructing a fixed solver schedule using mixed integer programming. `aspeed` [77] used ASP to solve timeout-optimal scheduling. I believe that `SATzilla`'s performance could be further improved by adapting such advanced techniques for optimizing presolving.

**Explaining instance hardness**. In real applications, it would be important to know "what property makes some type of instances hard?" or "Which algorithm components are most important to achieve good performance on certain types of instances?" I plan to extend my work on EHMs and develop an automated procedure for analysis of instances and algorithm components. This research will permit domain experts to gain insights into the functionality and interaction between multiple heuristics. Our group has been actively working on this topic: Hutter et al. (2014) proposed efficient methods for gaining insight into the relative importance of different hyperparameters and their interaction for machine learning algorithms; Fawcett et al. (2014) studied the relative importance of features for domain-independent planners.

**Parallel Algorithm Portfolios**: Depending on which heuristics are used, different algorithms' performance can vary significantly on the same instance. Running multiple solvers in parallel can, on average, be beneficial. Therefore, I plan to extend sequential portfolio techniques into a parallel environment. In the case of great uncertainty in algorithm selection, running more uncorrelated solvers in parallel can reduce the cost of picking a solver with a runtime much larger than the best solver. One recent approach [78] used an automatic algorithm configurator to produce a set of configurations to be executed in parallel.

# Bibliography


[1] D. W. Aha. Generalizing from case studies: A case study. In *Proceedings of the 9th International Conference on Machine Learning (ICML'92)*, pages 1–10. Morgan Kaufmann, 1992. → pages 14

[2] K. Ahmadizadeh, C. Dilkina, B.and Gomes, and A. Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*, volume 6308 of *LNCS*, pages 514–521. Springer, 2010. → pages 34

[3] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 of *LNCS*, pages 142–157. Springer, 2009. → pages 17

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding tours in the TSP. Technical Report TR-99885, University of Bonn, 1999. → pages 35, 36

[5] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006. → pages 1, 35, 83

[6] B. Arinze, S.-L. Kim, and M. Anandarajan. Combining and selecting forecasting models using rule based induction. *Computers and operations research*, 24:423–433, 1997. → pages 14

[7] D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 366–378. Springer, 2007. → pages 30

[8] D. Babić and F. Hutter. Spear theorem prover. Solver description, SAT competition 2007, http://www.domagoj-babic.com/uploads/Pubs/SAT08/sat08.pdf, Version last visited on April 29, 2014, 2007. → pages 82

[9] F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02)*, pages 613–619. AAAI Press, 2002. → pages 21

[10] F. Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Proceedings of the 5th International Conference on Theory and Applications of Satisfiability Testing (SAT'02)*, pages 7–16, 2002. → pages

[11] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *LNCS*, pages 341–355. Springer, 2003. → pages 21

[12] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In *Proceedings of the 4th International Conference on Hybrid Metaheuristics (HM'07)*, pages 108–122, 2007. → pages 17

[13] A. Balint, A. Belov, M. Järvisalo, and C. Sinz. The international SAT Challenge. http://baldur.iti.kit.edu/SAT-Challenge-2012/, 2012. Version last visited on August 6, 2012. → pages 2, 195

[14] T. Bartz-Beielstein. *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer, 2006. → pages 75

[15] T. Bartz-Beielstein and S. Markon. Tuning search algorithms for real-world applications: A regression tree based approach. In *Proceedings of the 2004 Congress on Evolutionary Computation (CEC'04)*, pages 1111–1118. IEEE Press, 2004. → pages 72, 77

[16] M. Berkelaar, J. Dirks, K. Eikland, P. Notebaert, and J. Ebert. lp_solve 5.5. http://lpsolve.sourceforge.net/5.5/index.htm, 2012. Version last visited on August 6, 2012. → pages 83

[17] T. Berthold, G. Gamrath, S. Heinz, M. Pfetsch, S. Vigerske, and K. Wolter. SCIP 1.2.1.4. http://scip.zib.de/doc/html/index.shtml, 2012. Version last visited on August 6, 2012. → pages 83

[18] A. Biere. Picosat version 535. Solver description, SAT competition 2007, http://www.satcompetition.org/2007/picosat.pdf, Version last visited on April 29, 2014, 2007. → pages 150

[19] A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008. → pages 102, 150

[20] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC'99)*, pages 317–320. ACM Press, 1999. → pages 1, 21

[21] M. Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, 2004. → pages 17

[22] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pages 11–18. Springer-Verlag, 2002. → pages 6, 7

[23] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stüzle. *Empirical Methods for the Analysis of Optimization Algorithms*, chapter F-race and iterated F-race: an overview, pages 311–336. Springer-Verlag, 2010. → pages 17

[24] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. → pages 58, 72

[25] D. R. Bregman and D. G. Mitchell. The SAT solver MXC, version 0.5. Solver description, SAT competition 2007, http://www.cs.sfu.ca/~mitchell/papers/MXC-Sat07-competition.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[26] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. → pages 43, 79, 80

[27] L. Breiman, J. H. Friedman, R. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. → pages 76, 77

[28] E. A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, September 1994. → pages 72

[29] E. A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP-95)*, pages 80–91, 1995. → pages 12, 72

[30] W. W. C. M. Li and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition 2007, ttp://www.satcompetition.org/2007/adaptG2WSAT.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[31] T. Carchrae and J. C. Beck. Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence*, 21(4): 373–387, 2005. → pages 13, 16

[32] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 331–337. AAAI Press, 1991. → pages 29, 41

[33] M. Chiarandini, C. Fawcett, and H. Hoos. A modular multiphase heuristic solver for post enrolment course timetabling. In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT'08)*, 2008. → pages 6, 7

[34] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 168–176. Springer, 2004. → pages 30, 146

[35] Concorde. Concorde TSP Solver, 2012. http://www.math.uwaterloo.ca/tsp/concorde/index.html. Version last visited on October 24, 2012. → pages 36

[36] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd annual ACM symposium on Theory of computing*, pages 151–158, 1971. → pages 1

[37] W. Cook. Concorde downloads page, 2012. http://www.tsp.gatech.edu/concorde/downloads/downloads.htm. Version last visited on October 24, 2012. → pages 38

[38] C. J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3SAT. *Artificial Intelligence*, 81:31–35, 1996. → pages 41

[39] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097. AAAI Press, 1994. → pages 21

[40] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(1):201–215, 1960. → pages 21

[41] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962. → pages 21

[42] R. Dechter and I. Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 134–145. Morgan Kaufman, 1994. → pages 21

[43] G. Dequen and O. Dubois. kcnfs. Solver description, SAT competition 2007, http://home.mis.u-picardie.fr/~dequen/sat/kcnfs-description--07.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[44] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 248–253. Morgan Kaufmann, 2001. → pages 21, 22, 42, 59, 101, 150

[45] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005. → pages 22, 26, 30, 59, 146

[46] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004. → pages 21, 59, 82, 126

[47] N. Eén and N. Sörensson. Minisat v2.0 (beta). Solver description, SAT Race 2006, http://fmv.jku.at/sat-race-2006/descriptions/27-minisat2.pdf, Version last visited on April 29, 2014, 2006. → pages 101

[48] C. Fawcett, M. Vallati, F. Hutter, J. Hoffmann, H. Hoos, and K. Leyton-Brown. Improved features for runtime prediction of domain-independent planners. In *International Conference on Automated Planning and Scheduling (ICAPS'14)*, page To appear, 2014. → pages 199

[49] A. S. Fraenkel. Complexity of protein folding. *Bulletin of Mathematical Biology*, 55:1199–1210, 1993. → pages 1

[50] A. S. Fukunaga. Automated discovery of composite sat variable-selection heuristics. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02)*, pages 641–648. AAAI Press, 2002. → pages 16, 17

[51] M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2006. → pages 14, 16

[52] M. Gagliolo and J. Schmidhuber. Impact of censored sampling on the performance of restart strategies. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *LNCS*, pages 167–181. Springer, 2006. → pages 95

[53] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. → pages 1

[54] L. D. Gaspero and A. Schaerf. EasySyn++: A tool for automatic synthesis of stochastic local search algorithms. In *Proceedings of the international conference on Engineering stochastic local search algorithms: designing, implementing and analyzing effective heuristics (SLS'07)*, volume 4638 of *LNCS*, pages 177–181. Springer, 2007. → pages 16

[55] C. Gebruers, A. Guerri, B. Hnich, and M. Milano. Making choices using structure at the instance level within a case based reasoning framework. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, volume 3011 of *LNCS*, pages 380–386. Springer, 2004. → pages 14

[56] C. Gebruers, B. Hnich, D. Bridge, and E. Freuder. Using CBR to select solution strategies in constraint programming. In *Proceedings of the 6th International Conference on Case Based Reasoning (ICCBR'05)*, volume 3620 of *LNCS*, pages 222–236. Springer, 2005. → pages 14

[57] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 28–33. AAAI Press, 1993. → pages 24

[58] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 654–660. AAAI Press, 1999. → pages 29, 146

[59] A. Gilpin and T. Sandholm. Information-theoretic approaches to branching in search. *Discrete Optimization*, 8(2):147–159, 2010. → pages 30

[60] C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 221–226. AAAI Press, 1997. → pages 29, 146

[61] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001. → pages 13

[62] C. P. Gomes, W. van Hoeve, and A. Sabharwal. Connections in networks: A hybrid approach. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'08)*, volume 5015 of *LNCS*, pages 303–307. Springer, 2008. → pages 30, 33

[63] J. Gratch and S. A. Chien. Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 4:365–396, 1996. → pages 17

[64] J. Gratch and G. Dejong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 235–240. AAAI Press, 1992. → pages 16, 17

[65] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 475–479. IOS Press, 2004. → pages 14

[66] H. Guo and W. H. Hsu. A learning-based algorithm selection meta-reasoner for the real-time MPE problem. In *Proceedings of the 17th Australian Conference on Artificial Intelligence (AI'04)*, volume 3339 of *LNCS*, pages 307–318. Springer, 2004. → pages 14

[67] Gurobi Optimization Inc. Gurobi 2.0. http://www.gurobi.com/, 2012. Version last visited on August 6, 2012. → pages 83

[68] I. Guyon, S. Gunn, M. Nikravesh, and L. Zadeh. *Feature Extraction, Foundations and Applications*. Springer, 2006. → pages 72

[69] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, 2nd edition, 2009. → pages 78

[70] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1): 106–130, 2000. → pages 36, 83

[71] P. Herwig. Using graphs to get a better insight into satisfiability problems. Master's thesis, Delft University of Technology, Department of Electrical Engineering, Mathematics and Computer Science, 2006. → pages 28

[72] M. Heule and H. V. Maaren. `march_dl`: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006. → pages 59

[73] M. Heule and H. V. Maaren. Improved version of march_ks. http://www.st.ewi.tudelft.nl/sat/Sources/stable/march_pl, 2007. Version last visited on June 16, 2009. → pages 21, 150

[74] M. Heule and H. v. Maaren. march_ks. Solver description, SAT competition 2007, http://www.satcompetition.org/2007/march_ks.pdf, Version last visited on April 29, 2014, 2007. → pages 102, 150

[75] M. Heule, J. Zwieten, M. Dufour, and H. Maaren. March_eq: implementing additional reasoning into an efficient lookahead SAT solver. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 4642 of *LNCS*, pages 345–359. Springer, 2005. → pages 101

[76] E. A. Hirsch. Random generator hgen2 of satisfiable formulas in 3-CNF. http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2-1.01.tar.gz, 2002. Version last visited on June 16, 2009. → pages 29, 146

[77] H. Hoos, R. Kaminski, T. Schaub, and M. Schneider. aspeed: ASP-based solver scheduling. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, pages 176–187. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012. → pages 199

[78] H. Hoos, K. Leyton-Brown, T. Schaub, and M. Schneider. Algorithm configuration for portfolio-based parallel SAT-solving. In *Workshop on Combining Constraint Solving with Mining and Learning (CoCoMile) at the European Conference on Artificial Intelligence (ECAI)*, 2012. → pages 199

[79] H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 661–666. AAAI Press, 1999. → pages 24, 26, 141

[80] H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02)*, pages 655–660. AAAI Press, 2002. → pages 21, 24, 139, 142, 150, 152, 156, 167, 197

[81] H. H. Hoos. Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Department of Computer Science, 2008. → pages 6, 16

[82] H. H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012. → pages 138

[83] H. H. Hoos and T. Stützle. *Stochastic Local Search – Foundations & Applications*. Morgan Kaufmann, 2005. → pages 12, 20, 22, 23, 36, 38

[84] E. Horvitz, Y. Ruan, C. P. Gomes, H. Kautz, B. Selman, and D. M. Chickering. A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI'01)*, pages 235–244. Morgan Kaufmann, 2001. → pages 14

[85] E. I. Hsu and S. A. McIlraith. Characterizing propagation methods for Boolean satisfiability. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *LNCS*, pages 325–338. Springer, 2006. → pages 22

[86] E. I. Hsu, C. Muise, J. C. Beck, and S. A. McIlraith. Probabilistically estimating backbones and variable bias: Experimental overview. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08)*, volume 5202 of *LNCS*, pages 613–617. Springer, 2008. → pages 28

[87] J. Huang. TINISAT in SAT competition 2007. Solver description, SAT competition 2007, http://www.satcompetition.org/2007/tinisat.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[88] L. Huang, J. Jia, B. Yu, B. Chun, P.Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of the 23rd Conference on Advances in Neural Information Processing Systems (NIPS'10)*, pages 883–891. MIT Press, 2010. → pages 12, 72, 73

[89] B. Huberman, R. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 265:51–54, 1997. → pages 13

[90] F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University Of British Columbia, Computer Science, 2009. → pages 31, 186

[91] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 233–248. Springer, 2002. → pages 21, 23, 25, 26, 83, 142, 149, 150, 152, 156

[92] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *LNCS*, pages 213–228. Springer, 2006. → pages 12, 13, 72, 94

[93] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 27–34. IEEE Press, 2007. → pages 17, 82, 184

[94] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07)*, pages 1152–1157. AAAI Press, 2007. → pages 6, 7, 17, 147

[95] F. Hutter, H. H. Hoos, T. Stützle, and K. Leyton-Brown. ParamILS version 2.3. http://www.cs.ubc.ca/labs/beta/Projects/ParamILS. Version last visited on Sept. 16, 2013., 2008. → pages 147

[96] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009. → pages 17, 18, 147, 175, 176, 178, 184, 187, 196

[97] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'10)*, volume 6140 of *LNCS*, pages 186–202. Springer, 2010. → pages 17, 18, 30, 34, 147, 184, 186

[98] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. Time-bounded sequential parameter optimization. In *Proceedings of the 4th Learning and Intelligent Optimization Conference (LION'10)*, volume 6073 of *LNCS*, pages 281–298. Springer, 2010. → pages 79

[99] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th Learning and Intelligent Optimization Conference (LION'11)*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011. → pages 13, 79, 192

[100] F. Hutter, J. Hoffmann, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31th International Conference on Machine Learning (ICML'14)*, page To appear. Omnipress, 2014. → pages 199

[101] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: methods & evaluation. *Artificial Intelligence*, 206(0):79–111, 2014. → pages iv, 20, 71, 194

[102] C. M. Institute. P vs NP Problem. http://www.claymath.org/millenium-problems/p-vs-np-problem, 2014. Version last visited on March 6, 2014. → pages 1

[103] International Business Machines Corp. IBM ILOG CPLEX Optimizer – Data Sheet, 2011. ftp://public.dhe.ibm.com/common/ssi/ecm/en/wsd14044usen/WSD14044USEN.PDF. Version last visited on August 6, 2012. → pages 31

[104] International Business Machines Corp. CPLEX 12.1. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer, 2012. Version last visited on August 6, 2012. → pages 83

[105] A. Ishtaiwi, J. Thornton, Anbulagan, A. Sattar, and D. N. Pham. Adaptive clause weight redistribution. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *LNCS*, pages 229–243. Springer, 2006. → pages 21

[106] D. S. Johnson. Random TSP generators for the DIMACS TSP Challenge. http://www2.research.att.com/~dsj/chtsp/codes.tar, 2011. Version last visited on May 16, 2011. → pages 39

[107] D. S. Johnson and L. A. McGeoch. *Local Search in Combinatorial Optimization*, chapter The Traveling Salesman Problem: a Case Study in Local Optimizaiton. Wiley and Sons, 1997. → pages 35, 36

[108] D. S. Johnson and L. A. McGeoch. Experimental analysis of heuristics for the STSP. *The Traveling Salesman Problem and its Variations*, pages 369–443, 2002. → pages 36

[109] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13: 455–492, 1998. → pages 75

[110] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufmann, 1995. → pages 12

[111] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - instance specific algorithm configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI'10)*, pages 751–756. IOS Press, 2010. → pages 9, 18, 19, 31, 34, 173, 186, 190, 191, 193, 194

[112] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, number 6876 in LNCS, pages 454–469. Springer, 2011. → pages 16, 124, 127, 193, 199

[113] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201. AAAI Press, 1996. → pages 21

[114] H. A. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 318–325. Morgan Kaufmann, 1999. → pages 1, 21

[115] A. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 517–524. Morgan Kaufmann, 2009. → pages iv, 20, 26, 138, 152, 180

[116] P. Kilby, J. Slaney, S. Thiebaux, and T. Walsh. Estimating search tree size. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*, pages 1014–1019. AAAI Press, 2006. → pages 12

[117] S. Kirkpatrick and B. Selman. Cirtical behavior in the satisfiability of random Boolean formulae. *Science*, 264:1297–1301, 1994. → pages 41

[118] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975. → pages 12, 14

[119] D. G. Krige. A statistical approach to some basic mine valuation problems on the Witwatersrand. *Journal of the Chemical, Metallurgical and Mining Society of South Africa*, 52(6):119–139, 1951. → pages 75

[120] B. Krishnapuram, L. Carin, M. Figueiredo, and A. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6): 957–968, 2005. → pages 60

[121] O. Kullmann. Investigating the behaviour of a SAT solver on random formulas. http://www.cs.swan.ac.uk/~csoliver/Artikel/OKsolverAnalyse.html, 2002. Version last visited on June 22, 2008. → pages 21, 59

[122] M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *Electronic Notes in Discrete Mathematics (ENDM)*, 2001. → pages 15

[123] N. D. Lawrence, M. Seeger, and R. Herbrich. Fast Sparse Gaussian Process Methods: The Informative Vector Machine. In *Proceedings of the 15th Conference on Advances in Neural Information Processing Systems (NIPS'02)*, pages 609–616. MIT Press, 2003. → pages 78

[124] D. Le Berre, O. Roussel, and L. Simon. The international SAT Competitions web page. www.satcompetition.org, 2012. Version last visited on Jan 29, 2012. → pages 4, 41, 122, 127, 195

[125] D. Lehmann, R. Müller, and T. Sandholm. *The Winner Determination Problem*. MIT Press, 2006. → pages 30

[126] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce (EC'00)*, pages 66–76. ACM, 2000. → pages 34

[127] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 556–572. Springer, 2002. → pages 3, 11, 12, 13, 56, 72, 94, 194

[128] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*, pages 899–903. Springer, 2003. → pages 7, 12, 173

[129] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1542–1543. Morgan Kaufmann, 2003. → pages 12, 14, 92

[130] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009. → pages 3, 8, 31, 34, 72, 173, 186

[131] C. Li and W. Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 158–172. Springer, 2005. → pages 21, 23, 25, 26, 139, 141, 142, 150, 152, 156

[132] C. Li, W. Wei, and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition 2007, ttp://www.satcompetition.org/2007/adaptG2WSAT.pdf, Version last visited on April 29, 2014, 2007. → pages 141, 142, 150, 152

[133] C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330 of *LNCS*, pages 341–355. Springer, 1997. → pages 59

[134] C. M. Li and W. Wei. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition 2009, https://www.laria.u-picardie.fr/~cli/adaptg2wsat2009++.tar, Version last visited on April 29, 2014, 2009. → pages 42

[135] C. M. Li, W. X. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *LNCS*, pages 121–133. Springer, 2007. → pages 24, 26, 140, 141, 150, 152, 167, 197

[136] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–51, 1973. → pages 36, 38

[137] G. Lindner and R. Studer. AST: Support for algorithm selection with a CBR approach. In *Principles of Data Mining and Knowledge Discovery*, volume 1704 of *LNCS*, pages 418–423. Springer, 1999. → pages 14

[138] L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 353–358. AAAI Press, 1998. → pages 12, 14

[139] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: an efficient SAT solver. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3542 of *LNCS*, pages 360–375. Springer, 2005. → pages 21, 22, 28, 101

[140] Y. Malitsky and M. Sellmann. Stochastic offline programming. In *Proceedings of the 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 784–791. IEEE, 2009. → pages 18, 193

[141] P. J. Manning and M. E. McDill. Optimal parameter settings for solving harvest scheduling models with adjacency constraints. *Mathematical and Computational Forestry & Natural-Resource Sciences*, 4(1):16–26, 2012. → pages 35

[142] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326. AAAI Press, 1997. → pages 24, 141, 142

[143] N. Meinshausen. Quantile regression forests. *Journal of Machine Learning Research*, 7:983–999, 2006. → pages 80

[144] O. Mersmann, B. Bischl, H. Trautmann, M. Wagner, J. Bossek, and F. Neumann. A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem. *Annals of Mathematics and Artificial Intelligence (AMAI)*, pages 151–182, 2013. → pages 38

[145] S. Minton. An analytic learning system for specializing heuristics. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 922–929. Morgan Kaufmann, 1993. → pages 16

[146] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 459–465. AAAI Press, 1992. → pages 3, 41

[147] H. D. Mittelmann. Performance of optimization software - an update. http://plato.asu.edu/talks/mittelmann_bench.pdf, 2011. Version last visited on March 16, 2014. → pages 31

[148] R. Monasson and R. Zecchina. Entropy of the k-satisfibility problem. *Physical Review Letters*, 76:3881–3885, 1996. → pages 42

[149] R. Monasson and R. Zecchina. The statistical mechanics of the random k-satisfibility model. *Physical Review E*, 56:1357–1370, 1997. → pages 42

[150] K. Murphy. The bayes net toolbox for matlab. *Computing Science and Statistics: Proceedings of Interface*, 33(2):1024–1034, 2001. → pages 59

[151] I. T. Nabney. *NETLAB: algorithms for pattern recognition*. Springer-Verlag, 2002. → pages 74

[152] A. Nadel, M. Gordon, A. Palti, and Z. Hanna. Eureka-2006 SAT solver. Solver description, SAT Race 2006, http://www.cs.tau.ac.il/research/alexander.nadel/Eureka.pdf, Version last visited on April 29, 2014, 2006. → pages 101

[153] M. Nikolić, F. Marić, and P. Janičić. Instance-based selection of policies for sat solvers. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, volume 5584 of *LNCS*, pages 326–340. Springer, 2009. → pages 162

[154] J. Nocedal and S. J. Wright. *Numerical Optimization (Second Edition)*. Springer, 2006. → pages 76

[155] E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. Hoos. Satzilla: An algorithm portfolio for SAT. Solver description, SAT competition 2004, http://www.researchgate.net/publication/2925723_SATzilla_An_Algorithm_Portfolio_for_SAT, Version last visited on April 29, 2014, 2004. → pages 5, 14

[156] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: beyond the clauses-to-variables ratio. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 438–452. Springer, 2004. → pages 3, 9, 12, 13, 26, 40, 44, 55, 56, 57, 60, 69, 72, 94

[157] M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005. → pages 16

[158] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, 2008. → pages 16

[159] B. Pfahringer, H. Bensusan, and C. Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*, pages 743–750. Morgan Kaufmann, 2000. → pages 14

[160] D. Pham and Anbulagan. Resolution enhanced SLS solver: R+AdaptNovelty+. Solver description, SAT competition 2007, http://www.satcompetition.org/2007/ranov.pdf, Version last visited on April 29, 2014, 2007. → pages 102, 150, 152, 156

[161] D. N. Pham, J. Thornton, C. Gretton, and A. Sattar. Combining adaptive and dynamic local search for satisfiability. *Journal on Satisfiability,*

*Boolean Modeling and Computation*, 4:149–172, 2008. → pages 26, 102, 141, 149, 150, 152, 156

[162] K. Pipatsrisawat and A. Darwiche. Rsat 1.03: SAT solver description. Technical Report D-152, Automated Reasoning Group, UCLA, 2006. → pages 101

[163] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: SAT solver description. Solver description, SAT competition 2007, http://reasoning.cs.ucla.edu/rsat/papers/rsat_2.0.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[164] M. Pop, S. L. Salzberg, and M. Shumway. Genome sequence assembly:algorithms and issues. *Computer*, 35(7):47–54, 2002. → pages 1

[165] P. C. Pop and S. Iordache. A hybrid heuristic approach for solving the generalized traveling salesman problem. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 481–488. ACM, 2011. → pages 147

[166] S. Prestwich. Random walk with continuously smoothed variable weights. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 203–215. Springer, 2005. → pages 24, 26, 139, 141, 142, 144, 148, 150, 152, 156

[167] J. Quinonero-Candela, C. E. Rasmussen, and C. K. Williams. Approximation methods for gaussian process regression. In *Large-Scale Kernel Machines*, Neural Information Processing, pages 203–223. MIT Press, 2007. → pages 78

[168] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006. → pages 75, 76, 78, 79

[169] G. Reinelt. *The Travelling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, 1994. → pages 36

[170] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15: 65–118, 1976. → pages 5, 14

[171] O. Roussel. Description of ppfolio. Solver description, SAT competition 2011, http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf, Version last visited on April 29, 2014, 2011. → pages 16, 124

[172] J. Sacks, W. J. Welch, T. J. Welch, and H. P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4(4):409–423, 1989. → pages 75

[173] H. Samulowitz and R. Memisevic. Learning to solve QBF. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07)*, pages 255–260. AAAI Press, 2007. → pages 15

[174] T. J. Santner, B. J. Williams, and W. I. Notz. *The Design and Analysis of Computer Experiments*. Springer-Verlag, 2003. → pages 75

[175] J. Schmee and G. J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432, 1979. → pages 95, 105

[176] M. Schmidt. minfunc. http://www.di.ens.fr/~mschmidt/Software/minFunc.html, 2012. Version last visited on August 5, 2012. → pages 76

[177] B. Selman and H. A. Kautz. Domain-independant extensions to GSAT : Solving large structured variables. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 290–295. Morgan Kaufmann, 1993. → pages 23

[178] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446. AAAI Press, 1992. → pages 21, 23

[179] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343. AAAI Press, 1994. → pages 21, 23, 24, 141, 142

[180] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81:17–29, 1996. → pages 29, 146

[181] J. Sherman and W. Morrison. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix (abstract). *Annals of Mathematical Statistics*, 20:621, 1949. → pages 73

[182] L. Simon. SAT competition random 3CNF generator. www.satcompetition.org/2003/TOOLBOX/genAlea.c, 2002. Version last visited on May 16, 2012. → pages 42

[183] S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998. → pages 36

[184] K. Smith-Miles and J. van Hemert. Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, 61:87–104, 2011. → pages 12, 39, 72, 74, 75

[185] K. Smith-Miles, J. van Hemert, and X. Y. Lim. Understanding TSP difficulty by learning from evolved instances. In *Proceedings of the 4th Learning and Intelligent Optimization Conference (LION'10)*, volume 6073 of *LNCS*, pages 266–280. Springer, 2010. → pages 37

[186] M. Soos. CryptoMiniSat 2.5.0. Solver description, SAT Race 2010, http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_13.pdf, Version last visited on April 29, 2014, 2010. → pages 82

[187] N. Sörensson and N. Eén. Minisat2007. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/, 2007. Version last visited on May 20, 2010. → pages 102, 150

[188] I. Spence. Ternary tree solver (tts-4-0). Solver description, SAT competition 2007, http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_13.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[189] P. Stephan, R. Brayton, and A. Sangiovanni-Vencentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:1167–1176, 1996. → pages 21

[190] M. Streeter, D. Golovin, and S. F. Smith. Combining multiple heuristics online. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07)*, pages 1197–1203. AAAI Press, 2007. → pages 14

[191] T. Stützle and H. Hoos. MAX-MIN ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000. → pages 35

[192] S. Subbarayan and D. Pradhan. Niver: Non-increasing variable elimination resolution for preprocessing SAT instances. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *LNCS*, pages 276–291. Springer, 2005. → pages 21

[193] G. Sutcliffe and C. B. Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39–54, 2001. → pages 126

[194] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500): 2319–2323, 2000. → pages 164

[195] J. Thornton, D. N. Pham, S. Bain, and V. Ferreira. Additive versus multiplicative clause weighting for SAT. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 191–196. AAAI Press, 2004. → pages 23, 25, 26, 141, 142, 144, 150, 152, 156

[196] K. M. Ting. An instance-weighting method to induce cost-sensitive trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(3):659–665, 2002. → pages 43

[197] D. A. Tompkins, A. Balint, and H. H. Hoos. Captain jack - new variable selection heuristics in local search for SAT. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, volume 6695 of *LNCS*, pages 302–316. Springer, 2011. → pages 147

[198] D. A. D. Tompkins and H. H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *LNCS*, pages 306–320. Springer, 2004. → pages 26, 139, 143

[199] V. Tresp. A Bayesian committee machine. *Neural Computation*, 12(11): 2719–2741, 2000. → pages 78

[200] T. Uchida and O. Watanabe. Hard SAT instance generation based on the factorization problem. http://www.is.titech.ac.jp/~watanabe/gensat/a2/GenAll.tar.gz, 1999. Version last visited on June 6, 2009. → pages 30, 146

[201] D. Vallstrom. Vallst documentation. http://vallst.satcompetition.org/index.html, 2005. Version last visited on May 20, 2010. → pages 101

[202] A. van Gelder. Another look at graph coloring via propositional satisfiability. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations (COLOR'02)*, pages 48–54, 2002. → pages 21

[203] W. Wei and C. M. Li. Switching between two adaptive noise mechanisms in local search for SAT. Solver description, SAT competition 2009, https://www.laria.u-picardie.fr/~cli/TNM.tar, Version last visited on April 29, 2014, 2009. → pages 82

[204] W. Wei, C. M. Li, and H. Zhang. Deterministic and random selection of variables in local search for SAT. Solver description, SAT competition 2007, http://www.satcompetition.org/2007/adaptg2wsat+.pdf, Version last visited on April 29, 2014, 2007. → pages 102

[205] E. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, 63:325–336, 1990. → pages 12

[206] N. A. Weiss. *A Course in Probability*. Addison-Wesley, 2005. → pages 81

[207] S. J. Westfold and D. R. Smith. Synthesis of efficient constraint-satisfaction programs. *Knowledge Engineering Review*, 16(1): 69–84, 2001. → pages 16

[208] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *LNCS*, pages 696–711. Springer, 2007. → pages iv, 55, 72, 107, 194

[209] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *LNCS*, pages 712–727. Springer, 2007. → pages 12, 20, 72, 95, 195

[210] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008. → pages iv, 6, 7, 9, 15, 16, 20, 26, 72, 88, 92, 175, 178, 195

[211] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla2009: An automatic algorithm portfolio for SAT. Solver description, 2009 SAT Competition, http://www.cs.ubc.ca/~xulin730/mypaper/satzilla2009.pdf, Version last visited on April 29, 2014, 2009. → pages iv, 28, 92

[212] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI'10)*, pages 210–216. AAAI Press, 2010. → pages iv, 172, 184

[213] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011. → pages iv, 173

[214] L. Xu, H. Hoos, and K. Leyton-Brown. Predicting satisfiability at the phase transition. In *Proceedings of the 26th National Conference on Artificial Intelligence (AAAI'12)*, pages 584 – 590. AAAI Press, 2012. → pages iv, 41

[215] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions in portfolio-based algorithm selectors. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *LNCS*, pages 228–241. Springer, 2012. → pages iv, 125

[216] L. Xu, F. Hutter, J. Shen, H. Hoos, and K. Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. Solver description, 2012 SAT Challenge, http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/SATzilla2012final.pdf, Version last visited on April 29, 2014, 2012. → pages iv, 92, 121

[217] Y. Xue, C. Wang, H. Ghenniwa, and W. Shen. A tree similarity measuring method and its application to ontology. *Journal of Universal Computer Science*, 15(9):1766–1781, 2001. → pages 162

[218] M. Yokoo. Why adding more constraints makes a problem easier for hill-climbing algorithms: Analyzing landscapes of CSPs. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330 of *LNCS*, pages 356–370. Springer, 1997. → pages 41

[219] E. Zarpas. Benchmarking SAT Solvers for Bounded Model Checking. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 340–354. Springer, 2005. → pages 30

[220] H. Zhang. SATO: an efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction (CADE'97)*, volume 1249 of *LNCS*, pages 272–275. Springer, 1997. → pages 59

[221] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Procedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. IEEE, 2001. → pages 59