Instance-Specific Algorithm Configuration

by
Yuri Malitsky

B.Sc., Cornell University, Ithaca, NY, 2007
M.Sc., Brown University, Providence, RI, 2009

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in The Department of Computer Science at Brown University

PROVIDENCE, RHODE ISLAND

May 2012

This dissertation by Yuri Malitsky is accepted in its present form
by The Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date_____ _____

Meinolf Sellmann, Ph.D., Advisor

Recommended to the Graduate Council

Date_____ _____

Barry O'Sullivan, Ph.D., Reader

Date_____ _____

Erik Sudderth, Ph.D., Reader

Date_____ _____

Anna Lysyanskaya, Ph.D., Reader

Approved by the Graduate Council

Date_____ _____

Peter M. Weber, Dean of the Graduate School

Vitae

2009-2012 Ph.D., Brown University, Providence, RI, United States
2007-2009 M.Sc., Brown University, Providence, RI, United States
2003-2007 B.Sc., Cornell University, Ithaca, NY, United States

Acknowledgements

When it comes to a PhD thesis, even though there is only one name on the cover page, this work would not have been possible without the support of the people around me. First and foremost of these is of course my advisor and mentor, Meinolf Sellmann. With his brilliant guidance, help, and motivation, he always pushed me ever onwards and thus led me from a wide eyed undergrad with too much energy and no idea what to do with it, into a researcher that is now defending his PhD.

I would also like to thank my coauthors: Ashish Sabharwal, Horst Samulowitz, Carla Pedro Gomes, Willem Jan Jan Hoeve, Serdar Kadioglu, Kevin Tierney, Bistra N Dilkina, and Christian Kroer. Working with you has been a pure delight, and I hope to continue our collaborations in the future.

Besides the coauthors, I really need to thank all the people you have been so helpful in proof reading this thesis, some of them more than once. I know that reading someone else's manuscript must have been a daunting task, but every suggestion you made helped make this thesis that much better. And for that I thank you: Serdar Kadioglu, Kevin Tierney, Stu Black, Carleton Coffrin, and Christian Kroer.

Finally, I must thank all of the CS Grad students at Brown whose presence has helped keep me sane through the all nighters and the constant deadlines. Thanks to you all, and especially (in no particular order) to Serdar Kadioglu, Olya Ohrimenko, Justin Yip, Carleton Coffrin, Irina Calciu, Stu Black, Rebecca Mason, Alexandra Papoutsaki, Aggeliki Tsoli, Eric Sodomka, Steve Gomez, Andy Pavlo, Micha Elsner, Dae Il Kim, and Genevieve Patterson.

Abstract of " Instance-Specific Algorithm Configuration " by Yuri Malitsky, Ph.D., Brown University, May 2012

When developing a new heuristic or complete algorithm for a constraint satisfaction or constrained optimization problem, we frequently face the problem of choice. There may be multiple branching heuristics that we can employ, different types of inference mechanisms, various restart strategies, or a multitude of neighborhoods from which to choose. Furthermore, the way in which the choices we make affect one another is not readily perceptible. The task of making these choices is known as algorithm configuration.

Developers often make many of these algorithmic choices during the prototyping stage. Based on a few preliminary manual tests, certain algorithmic components are discarded, even before all the remaining components have been implemented. However, by making the algorithmic choices beforehand developers may unknowingly discard components that are used in the optimal configuration. In addition, the developer of an algorithm has limited knowledge about the instances that a user will typically employ the solver for. That is the very reason why solvers have parameters: to enable users to fine-tune a solver for their specific needs.

On the flip side, manually tuning a parameterized solver can require significant resources, effort, and expert knowledge. Before even trying the numerous possible parameter settings, the user must learn about the inner workings of the solver to understand what each parameter does. Furthermore, it has been shown that manual tuning often leads to highly inferior performance.

This dissertation shows how to train a multi-scale, multi-task approach for enhanced performance based on machine learning techniques automatically. In particular this work presents a new methodology for Instance-Specific Algorithm Configuration (ISAC). ISAC is a general configurator that focusses on tuning different categories of parameterized solvers according to the instances they will be applied to. Specifically, this dissertation shows that the instances of many problems can be decomposed into a representative vector of features. It further shows that instances with similar features often cause similar behavior in the applied algorithm. ISAC exploits this observation by automatically detecting the different sub-types of a problem and then training a solver for each variety. This technique is explored on a number of problem domains, including set covering, mixed integer, satisfiability, and set partitioning. ISAC is then further expanded to demonstrate its application to traditional algorithm portfolios and adaptive search methodologies. In all cases, marked improvements are shown over the existing state-of-the-art solvers. These improvements were particularly evident during the 2011 SAT Competition, where a solver based on ISAC won 7 medals, including a gold in the Handcrafted instance category, and another gold in the Randomly generated instance category.

Contents

List of Tables

List of Algorithms

List of Figures

Chapter One

Introduction

1.1 Quest for an Efficient Problem-Cognizant Solver

In computer science it is often the case that programs are designed to solve many instances of the same problem. In the stock market for example, there are programs that must continuously evaluate the value of a portfolio, deciding the most opportune time to buy or sell stocks. In container stowage, each time a container ship comes into port, a program needs to find a way to load and unload containers as quickly as possible while not compromising the ship's integrity and making sure that at the next port all the containers that need to be unloaded are closer to the top of the stacks. In databases, there are programs that need to schedule jobs and store information across multiple machines continually so that the average time to completion is minimized. A robot relying on a camera needs to process images detailing the state of its current environment continually. Whenever dealing with uncertainty, like in the case of hurricane land fall, an algorithm needs to evaluate numerous scenarios to choose the best evacuation routes. Furthermore, these applications need not only be online tasks, but can be offline as well. Scheduling airplane flights and crews for maximum profit needs done every so often to adjust to changes, delays and mechanical issues, but they do not need to be computed instantly.

In all the above-mentioned cases, and many more similar ones, the task of the program is to solve different instances of the same problem continually. In such applications, it is not enough just to solve the problem, but it is also necessary that this is done with increasing accuracy and/or efficiency. One possible way to achieve these improvements, is for developers and researchers to keep designing progressively better algorithms. While this is essential for continual progress, it is obvious that there is no singular universally best algorithm. Tangential to this research, there is still a lot of potential that can be gained through a better utilization and understanding of existing techniques.

In practice, developers often make decisive choices about the internal parameters of a solver when creating it. But because a solver can be used to address many different problems, certain settings or heuristics are beneficial for one group of instances while a different setting could be better for another problem. It is therefore important to develop configurable solvers, whose internal behavior can be adjusted to suit the application at hand.

Let us take the very simple example of a simulated annealing (SA) search. This probabilistic local search strategy was inspired by a phenomenon in metallurgy where repeated controlled heating and cooling would result in the formation of larger crystals with fewer defects. Analogously, the search strategy tries to replace its current solution with a randomly selected neighboring solution. If this neighbor is better than the current solution, it is accepted as the new current solution. However, if this random solution is worse, it is selected with some probability depending on the current temperature parameter and how much worse it is than the current solution. Therefore, the higher the temperature, the more likely the search is to accept the new solution, thus exploring more of the search space. Alternatively, as the temperature is continually lowered as the search proceeds, SA focuses more on improving solutions and thus exploiting a particular portion of the search space. In practice, SA has been shown to be highly effective on a multitude of problems, but the key to its success lies in the initial setting of the temperature parameter and the speed with which it is lowered. Setting the temperature very high can be equivalent to random sampling but works well in a vary jagged search space with many local optima.

Alternatively, a low temperature is much better for quickly finding a solution in a relatively smooth search area but is unlikely ever to leave a local optima. The developer, however, often does not know the type of problem the user will be solving, so fixing these parameters beforehand can be highly counterproductive. Yet in many solvers, constants like the rate of decay, the frequency of restarts, the learning rate, etc. are all parameters that are deeply embedded within the solver and manually set by the developer.

Generalizing further from individual parameters, it is clear that even the deterministic choice of the employed algorithms must be left open to change. In optimization, there are several seminal papers advocating the idea of exploiting statistics and machine learning technology to increase the efficiency of combinatorial solvers. For example, it has been suggested to protocol during the solution process of a constraint satisfaction problem those variable assignments that cause a lot of filtering and to base future branching decisions on this data. This technique, called impact-based search, is one of the most successful in constraint programming and has become part of the IBM Ilog CP Solver. The developer might know about the success of this approach and choose it as the only available heuristic in the solver. Yet while the method works really well in most cases, there are scenarios where just randomly switching between multiple alternate heuristics performs just as well, if not better. In other words, there is no single heuristic or approach that has been shown to be best over all scenarios.

Another way to explain this phenomenon is to point out that the problems being solved are in fact NP-hard, and there is yet no existing deterministic win-all algorithm. Therefore two things can be claimed. First, it is possible to construct worst-case scenarios where any statistical inference method fails completely. For instance, consider impact-based search for solving SAT. Take any two SAT formulae α, β, both over variables $x_1, \ldots, x_n$. Let us introduce a new variable x_0 and add $\vee x_0$ to all clauses in α and $\vee \bar{x_0}$ to all clauses in β. The SAT problem we want to solve is the conjunction of all modified clauses in α and β. Say we branch on variable x_0 and set it to false first. Impact-based search gathers statistics in the resulting left subtree to guide the search in the right subtree. However, after setting x_0 to false for the left subtree, the resulting problem is to find a satisfying assignment for α. In the right subtree, we set x_0 to true, and the task is to find a satisfying assignment for β. Since α and β were chosen independently from one another, it is not reasonable to assume that the statistics gathered when solving α are in any way meaningful for the solution of β. And obviously, α and β can be chosen in such a way that the statistics gathered when solving α are completely misleading when solving β.

The second aspect is that stochastic algorithms have usually led to impressive improvements in practice, despite the above worst-case argument. That is to say, there is substantial practical evidence that exploiting (online or offline) statistical knowledge can boost the average-case performance of combinatorial solvers. In some sense one may argue that the very fact that statistical inference does not work in the worst-case is what makes it statistical inference. If we could draw any hard conclusions, we would revert to deterministic inference and filter variable domains or derive new redundant constraints. However, statistical inference only kicks in when our ability to reason about the given problem deterministically is exhausted.

As the bottom line however, since these approaches tend to aim to improve the average-case performance, it means that they are gaining improvements in some cases at the expense of decreased performance on others. Therefore, there is no single solver or algorithm that

works best over all scenarios. In order to make the solvers as accessible as possible, the developers should make all choices available to the user, thus letting the users choose the methodologies that are best for their particular datasets.

One of the success stories of such an approach from the boolean satisfiability domain is SATenstein[55]. Observing the ideas and differences behind some of the most successful local search SAT solvers, the creators of SATenstein noticed that all solvers followed the same general structure. The solver selected a variable in the SAT formula and then assigned it to be either true or false. The differences in the solvers were mainly due to how the decision was made to select the variable and which value was assigned. Upon this observation, SATenstein was developed such that all existing solvers could be replicated by simply modifying a few parameters. This not only created a single base for any existing local search SAT solver, but also allowed users easily to try new combinations of components to experiment with previously unknown solvers. It is therefore imperative to make solvers and algorithms configurable allowing for them to be used to maximum performance for the application at hand.

Yet while solvers like SATenstein provide the user with a lot of power to fine tune a solver to their exact specifications, the problem of choice arises. SATenstein has over 40 parameters that can be defined. A mathematical programming solver like IBM Cplex[52] has over 100. Without expert knowledge of exactly how these solvers work internally, setting these parameters becomes a guessing game rather than research. Things are further complicated if a new version of the solver becomes available with new parameters or the non-linear relation between some parameters changes. This also makes switching to a new solver a very expensive endeavor, requiring time and resources to become familiar with the new environment. On top of the sheer expense of manually tuning parameters, it has been consistently shown that even the developers who originally made the solver struggle when setting the parameters manually.

The research into AI algorithms and techniques that can automate the setting of a solver's parameters has resulted in a paradigm shift with advantages beyond improved performance of solvers. For one, research in this direction can potentially improve the quality of comparisons between existing solvers. As things are now, when a new solver needs to be compared to the existing state-of-the-art, it is often the case that the developers find a new dataset and then carefully tweak their proposed approach. However, when running the competing solver, much less time is devoted to making sure it is running optimally. There are many possible reasons for this, but the result is the same. Through automating the configuration of solvers for the problems at hand, a much more fair comparison can be achieved. Furthermore, through this approach towards configuration, it will be possible to claim the benefits of a newly proposed heuristic or method definitively if it is automatically chosen as best for a particular dataset or, even better, if the configuration tool can automatically find the types of instances where the new approach is best.

Through tuning, researchers would also be allowed to focus their efforts in the development of new algorithms better. When improving performance of a solver, it is important to note whether the benefits are coming from small improvements on many easy instances or a handful of hard ones. Through tuning, it is possible to identify the current bounds on performance, effectively honing in on cases where a breakthrough can do the most benefit. Furthermore, by studying benchmarks, it might be possible to discern the structural differences between these hard instances from the easy ones, which can lead to insights on

what makes the problems hard and how these differences can be exploited.

Additionally, what if we can automatically identify the hard problems? What if by studying the structure of hard instances we notice that the structure can be systematically perturbed to make the instance easier. What if we can intelligently create hard instances that have a particular internal structure instead of randomly trying to achieve interesting benchmarks? What if we can create adaptive solvers that detect changes in the problem structure and can completely modify their strategy based on these changes?

This dissertation presents a new methodology that is motivated by these issues, creating a clear infrastructure that can be readily expanded and applied to a variety of domains.

1.2 Thesis Outline

This dissertation shows how to train a multi-scale, multi-task approach for enhanced performance based on machine learning techniques automatically .

Although the idea of automatically tuning algorithms is not new, the field of automatic algorithm configuration has experienced a renaissance in the past decade. There now exist a number of techniques that are designed to select a parameter set automatically that on average works well on all instances in the training set [49, 4]. The research outlined in this paper takes the current approaches a step further by taking into account the specific problem instances that need to be solved. Instead of considering that there is one optimal parameter set that will yield the best performance on all instances, it assumes that there are multiple types of problems, each yielding to different strategies. Furthermore, the dissertation assumes that there exist a finite collection of features for each instance that can be used to correctly identify its structure, and thus used to identify the sub-types of the problems. Taking advantage of these two assumptions we introduce Instance-Specific Algorithm Configuration (ISAC), a new automated procedure to provide instance-specific tuning.

This dissertation, continues with an outline of related work in the field of training solvers in Chapter 2. Chapter 3 then explains the proposed approach, ISAC, and how it is different from prior research. Chapter 4 presents the application of ISAC to set covering, mixed integer programs, and satisfiability problems. Furthermore, Chapter 5 shows how the approach can be used to train algorithm portfolios, improving performance over existing techniques that use regression. Chapter 6 enhances ISAC, showing how feature filtering can be accomplished in cases where evaluating performance can be restrictively expensive. Chapter 7 then shows how ISAC can be modified to handle dynamic training, where a unique algorithm is tuned for each instance. Following is, Chapter 8, which shows how to tune parallel portfolio algorithms. In Chapter 9 the thesis shows how ISAC can be used to create an adaptive solver that changes its behavior based on the current sub-problem observed during search. Each of these sections are supported by numerical evaluation. The dissertation concludes with a discussion of the strengths and weaknesses of the ISAC methodology and potential future work.

Chapter Two

Related Work

Automatic algorithm configuration is a quickly evolving field that aims to overcome the limitations and difficulties associated with manual parameter tuning. Many techniques have been attempted to address this problem, including meta-heuristics, evolutionary computation, local search, etc. Yet despite the variability in the approaches, this flood of proposed work mainly ranges between four ideas: algorithm construction, instance-oblivious tuning, instance-specific regression, and adaptive methods. The four sections of this chapter discuss the major works for each of these respective philosophies and the final section summarizes the chapter.

2.1 Algorithm Construction

Algorithm construction focuses on automatically creating a solver from an assortment of building blocks. These approaches define the structure of the desired solver, declaring how the available algorithms and decisions need to be made. A machine learning technique then evaluates different configurations of the solver trying to find the one that performs best on a collection of training instances.

The MULTI-TAC system [67] is an example of this approach applied to the constraint satisfaction problem (CSP). The backtracking solver is defined as a sequence of rules that determine which branching variable and value selection heuristics to use under what circumstances, as well as how to perform forward checking. Using a beam search to find the best set of rules, the system starts with an empty configuration. The rules or routines are then added one at a time. A small Lisp program corresponding to these rules is created and run on the training instances. The solver that properly completes the most instances proceeds to the next iteration. The strength of this approach is the ability to represent all existing solvers while automatically finding changes that can lead to improved performance. The algorithm, however, suffers from the search techniques used to find the best configurations. Since the CSP solver is greedily built one rule or routine at a time, certain solutions can remain unobserved. Furthermore, as the number of possible routines and rules grows or the underlying schematic becomes more complicated, the number of possible configurations becomes too large for the described methodology.

Another approach from this category is the CLASS system, developed by Fukunaga [29]. This system is based on the observation that many of the existing local search (LS) algorithms used for SAT are seemingly composed of the same building blocks with only minor deviations. The Novelty solver [66], for example, is based on the earlier GWSAT solver [84], except instead of randomly selecting a variable in a broken clause, it chooses the one with the highest net gain. Minor changes like these have continuously improved LS solvers for over a decade. The CLASS system tries to automate this reconfiguration and fine tuning process by developing a concise language that can express any existing LS solver. A genetic algorithm then creates solvers that conform to this language. To avoid overly complex solvers, all cases having more than two nested conditionals are automatically collapsed by replacing the problematic sub-tree with a random function of depth one. The resulting solvers were shown to be competitive with the best existing solvers. The one issue with this approach, however, is that developing such a grammar for other algorithms or problem types can be difficult, if not impossible.

As another example, in [71] Oltean proposed to construct a solver that uses a genetic algorithm (GA) automatically. In this case, the desired solver is modeled as a sequence of

the selection, combination and mutation operations of a GA. For a given problem type and collection of training instances, the objective is to find the sequence of these operations that results in the solver requiring the fewest iterations to train. To find this optimal sequence of operations, Oltean proposes to use a linear genetic program. The resulting algorithms were shown to outperform the standard implementations of genetic algorithms for a variety of tasks. However, while this approach can be applied to a variety of problem types, it ultimately suffers from requiring a long time to train. Just to evaluate an iteration of the potential solvers, each GA needs to be run 500 times on all the training instances to determine the best solver in the population accurately. This is fine for quickly evaluated instances, but once each instance requires more than a couple of seconds to evaluate, the approach becomes too time-consuming.

Algorithm construction has also been applied to create a composite sorting algorithm used by a compiler [62]. The authors observed that there is no single sorting strategy that works perfectly on all possible input instances, with different strategies yielding improved performance on different instances. With this observation, a tree-based encoding was used for a solver that iteratively partitioned the elements of an instance until reaching a single element in the leaf node, and then sorted the elements as the leaves were merged. The primitives defined how the data is partitioned and under what conditions the sorting algorithm should change its approach. For example, the partitioning algorithm employed would depend on the amount of data that needs to be sorted. To make their method instance-specific, the authors use two features encoded as a six-bit string. For training, all instances are split according to the encodings and each encoding is trained separately. To evaluate the instance, the encoding of the test instance is computed and the algorithm of the nearest and closest match is used for evaluation. This approach has been shown to be better than all existing algorithms at the time, providing a factor two speedup. The issue with the approach, however, is that it only uses two highly disaggregated features to identify the instance and that during training it tries to split the data into all possible settings. This becomes intractable as the number of features grows.

2.2 Instance-Oblivious Tuning

Given a collection of sample instances, instance-oblivious tuning attempts to find the parameters resulting in the best average performance of a solver on all the training data. There are three types of solver parameters. First, parameters can be categorical, controlling decisions like what restart strategy to use or which branching heuristic to employ. Alternatively, parameters can be ordinal, controlling decisions like the size of the neighborhood for a local search or the size of the tabu list. Finally, parameters can be continuous, defining an algorithm's learning rate or the probability of making a random decision. Due to these differences, the tuning algorithms used to set the parameters can vary wildly. For example, the values of a categorical parameter have little relation to each other, making it impossible to use regression techniques. Similarly, continuous parameters have much larger domains than ordinal parameters. Here we discuss a few of the proposed methods for tuning parameters.

One example of instance-oblivious tuning focuses on setting continuous parameters. Coy *et.al* [23] suggested that by computing a good parameter set for a few instances, averaging all the parameters will result in parameters that would work well in the general

case. Given a training set, this approach first selected a small diverse set of problem instances. The diversity of the set was determined by a few handpicked criteria specific to the problem type being solved. Then analyzing each of these problems separately, the algorithm tests all possible extreme settings of the parameters. After computing the performance at these points, a response surface is fitted, and greedy descent is used to find a locally optimal parameter set for the current problem instance. The computed parameter sets computed for each instance are finally averaged to return a single parameter set expected to work well on all instances. This technique was empirically shown to improve solvers for set covering and vehicle routing. The approach, however, suffers once more parameters need to be set or if these parameters are not continuous.

For a small set of possible parameter configurations, F-Race [15] employs a racing mechanism. During training, all potential algorithms are raced against each other, whereby a statistical test eliminates inferior algorithms before the remaining algorithms are run on the next training instance. But the problem with this is that it prefers small parameter spaces, as larger ones would require a lot of testing in the primary runs. Careful attention must also be given to how and when certain parameterizations are deemed pruneable, as this greedy selection is likely to end with a sub-optimal configuration.

Alternatively, the CALIBRA system, proposed in [3], starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine starts from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments.

For derivative free optimization of continuous variables, [6] introduced a mesh adaptive direct search (MADS) algorithm. In this approach, the parameter search space is partitioned into grids, and the corner points of each grid are evaluated for best performance. The grids associated with the current lower bound are then further divided and the process is repeated until no improvement can be achieved. One of the additional interesting caveats to the proposed method was to use only short running instances in the training set to speed up the tuning. It was observed that the parameters found for the easy instances tended to generalize to the harder ones, thus leading to significant improvements over classical configurations.

In another example, a highly parameterized solver like SATenstein [55] was developed, where all the choices guiding the stochastic local search SAT solver were left open as parameters. SATenstein can therefore be configured into any of the existing solvers as well as some completely new configurations. Among the methods used to tune such a solver is ParamILS.

In 2007, ParamILS [49] was first introduced as a generic parameter tuner, able to configure arbitrary algorithms with very large numbers of parameters. The approach conducts focused iterated local search, whereby starting with a random assignment of all the parameters, a local search with a one-exchange neighborhood is performed. The local search continues until a local optimum is encountered, at which point the search is repeated from a new starting point. To avoid randomly searching the configuration space, at each iteration the local search gathers statistics on which parameters are important to finding improved settings, and focuses on assigning them first. This black-box parameter tuner has been shown to be successful with a variety of solvers, including Cplex [52], SATenstein [55] and SAPS [51], but suffers due to not being very robust, and depending on the parameters being discretized.

As an alternative to ParamILS, in 2009 the gender-based genetic algorithm [4] (GGA) was introduced. This black box tuner conducts a population-based local search to find the best parameter configuration. This approach presented a novel technique of introducing competitive and non-competive genders to balance exploitation and exploration of the parameter space. Therefore, at each generation, half of the population competes on a collection of training instances. The parameter settings that yield the best overall performance are then mated with the non-competetive population, with the children removing the worst-performing individuals from the competitive population. This approach was shown to be remarkably successful in tuning existing solvers, often outperforming ParamILS.

Most recently, a Sequential Model-based Algorithm Configuration (SMAC) [48] was introduced in 2010. This approach proposes to generate a model over the solver's parameters to predict the likely performance. This model can be anything from a random forest to marginal predictors. This model is used to identify aspects of the parameter space, like what parameters are most important. Possible configurations are then generated according to this model and compete against the current incumbent. The best configuration continues onto the next iteration. While this approach has been shown to work on some problems, it ultimately depends on the accuracy of the model used to capture the interrelations of the parameters.

2.3 Instance-Specific Regression

One of the main drawbacks of instance-oblivious tuning is ignoring the specific instances, striving instead for the best average case performance. However, works like [94], [69] and many others have observed that not all instances yield to the same approaches. This observation supports the no free lunch theorem [104], which states that no single algorithm can be expected to perform optimally over all instances. Instead, in order to gain improvements in performance for one set of instances, it will have to sacrifice performance on another set. The typical instance-specific tuning algorithm computes a set of features for the training instances and uses regression to fit a model that will determine the solver's strategy.

Algorithm portfolios are a prominent example of this methodology. Given a new instance, the approach forecasts the runtime of each solver and runs the one with the best predicted performance. SATzilla [108] is an example of this approach as applied to SAT. In this case the algorithm uses ridge regression to forecast the log of the run times. Interestingly, for the instances that timeout during training, the authors suggest to use the predicted times as the observed truth, a technique they show to be surprisingly effective. In addition, SATzilla uses feedforward selection over the features it uses to classify a SAT instance. It was found that certain features are more effective at predicting the runtimes of randomly generated instances as opposed to industrial instances and vice-versa. Overall, since its initial introduction in 2007, SATzilla has won medals at the 2007 and 2009 SAT Competitions [2].

In algorithm selection the solver does not necessarily have to stick to the same algorithm once it is chosen. For example, [35] proposed to run in parallel (or interleaved on a single processor) multiple stochastic solvers that tackle the same problem. These "algorithm portfolios" were shown to work much more robustly than any of the individual stochastic

solvers. This insight has since led to the technique of randomization with restarts, which is commonly used in all state-of-the-art complete SAT solvers. Algorithm selection can also be done dynamically. As was shown in [30], instead of choosing the single best solver from a portfolio, all the solvers are run in parallel. However, rather than allotting equal time to everything, each solver is biased depending on how quickly the algorithm thinks it will complete. Therefore, a larger time share is given to the algorithm that is assumed to be first to finish. The advantage of this technique is that it is less susceptible to an early error in the performance prediction.

In [73], a self-tuning approach is presented that chooses parameters based on the input instance for the local search SAT solver WalkSAT. This approach computes an estimate of the invariant ratio of a provided SAT instance, and uses this value to set the noise of the WalkSAT solver, or how frequently a random decision is made. This was shown to be effective on four DIMACS benchmarks, but failed for those problems where the invariant ratio did not relate to the optimal noise parameter.

In another approach, [46, 47] tackle solvers with continuous and ordinal (but not categorical) parameters. Here, Bayesian linear regression is used to learn mapping from features and parameters into a prediction of runtime. Based on this mapping for given instance features, a parameter set that minimizes predicted runtime is searched for. The approach in [46] led to a twofold speed-up for the local search SAT solver SAPS [51].

Most recently, an alternative example expands on the ideas introduced in SATzilla by presenting Hydra [106]. Instead of using a set of existing solvers, this approach uses a single highly parameterized solver. Given a collection of training instances, a set of different configurations are produced to act as the algorithm portfolio. Instances that are not performing well under the current portfolio are then identified and used as the training set for a new parameter configuration that is to be added to the portfolio. Alternatively, if a configuration is found not to be useful any longer, it is removed from the portfolio. A key ingredient to making this type of system work is the provided performance metric, which uses a candidate's actual performance when it is best and the overall portfolio's performance otherwise. This way, a candidate configuration is not penalized for aggressively tuning for a small subset of instances. Instead, it is rewarded for finding the best configurations and thus improving overall performance.

An alternative to regression based approaches for instance specific tuning, CPHydra [72] attempts to schedule solvers to maximize the probability of solving an instance within the allotted time. Given a set of training instances and a set of available solvers, CPHydra collects information on the performance of every solver on every instance. When a new instance needs to be solved, its features are computed and the k-nearest neighbors are selected from the training set. The problem then is set as a constraint program, that tries to find the sequence and duration in which to invoke the solvers so as to yield the highest probability of solving the instance. The effectiveness of the approach was demonstrated when CPHydra won the CSP Solver Competition in 2008, but also showed the difficulties of the approach since the dynamic scheduling program only used three solvers and a neighborhood of 10 instances.

2.4 Adaptive Methods

All of the works presented so far were trained offline before being applied to a set of test instances. Alternative approaches exist that try to adapt to the problem they are solving in an online fashion. In this scenario, as a solver attempts to solve the given instance, it learns information about the underlying structure of the problem space, trying to exploit this information in order to boost performance.

An example of this technique is STAGE [17], an adaptive local search solver. While searching for a local optima, STAGE learned an evaluation function to predict the performance of a local search algorithm. At each restart, the solver would predict which local search algorithm was likely to find an improving solution. This evaluation function was therefore used to bias the trajectory of the future search. The technique was empirically shown to improve the performance of local search solvers on a variety of large optimization problems.

Impact-based search strategies for constraint programming (CP)[79] are another example of a successful adaptive approach. In this work, the algorithm would keep track of the domain reduction of each variable after the assignment of a variable. Assuming that we want to reduce the domains of the variables quickly and thus shrink the search space, this information about the impact of each variable guides the variable selection heuristic. The empirical results were so successful that this technique is now standard for Ilog CP Solver, and used by many other prominent solvers, like MiniSAT [27].

In 1994, an adaptive technique was proposed for tabu search [8]. By observing the average size of the encountered cycles, and how often the search returned to a previous state, this algorithm dynamically modified the size of its tabu list.

Another interesting result for transferring learned information between restarts was presented in Disco-Novo-GoGo [83]. In this case, the proposed algorithm uses a value-ordering heuristic while performing a complete tree search with restarts. Before a restart takes place, the algorithm observes the last tried assignment and changes the value ordering heuristic to prefer the currently assigned value. In this way, the search is more likely to explore a new and more promising portion of the search space after the restart. When applied to constraint programming and satisfiability problems, orders of magnitude performance gains were observed.

2.5 Chapter Summary

In this chapter, related work for automatic algorithm configuration was discussed. The first approach of automatic algorithm construction focused on how solving strategies and heuristics can be automatically combined to result in functional solver by defining the solver's structure. Alternatively, given that a solver is created where all the controlling parameters are left open to the user, the instance-oblivious methodology finds the parameter settings that result in the best average-case performance. When a solver needs to be created to perform differently depending on the problem instance, instance-specific regression is often employed to find an association between the features of the instance and the desired parameter settings. Finally, to avoid extensive offline training on a set of representative instances, adaptive methods that adapt to the problem dynamically are

also heavily researched.

All these techniques have been shown empirically to provide significant improvements in the quality of the tuned solver. Each approach, however, also has a few general drawbacks. Algorithm construction depends heavily on the development of an accurate model of the desired solver; however, for many cases, a single model that can encompass all possibilities is not available. Instance-oblivious tuning assumes that all problem instances can be solved optimally by the same algorithm, an assumption that has been frequently shown impossible in practice. Instance-specific regression, on the other hand, depends on accurately fitting a model from the features to a parameter, which is intractable and requires a lot of training data when the features and parameters have non-linear interactions. Adaptive methods require a high overhead since they need to spend time exploring and learning about the problem instance while attempting to solve it. The remainder of this paper focuses on how instance-oblivious tuning can be extended to create a modular and configurable framework that is instance-specific.

Chapter Three

Instance-Specific Algorithm Configuration

Instance-Specific Algorithm Configuration, ISAC, the proposed approach, takes advantage of the strengths of two existing techniques, instance-oblivious tuning and instance-specific regression, while mitigating their weaknesses. Specifically, ISAC combines the two techniques to create a portfolio where each solver is tuned to tackle a specific type of problem instance in the training set. This is achieved using the assumption that problem instances can be accurately represented by a finite number of features. Furthermore, it is assumed that instances that have similar features can be solved optimally by the same solver. Therefore, given a training set, the features of each instance are computed and used to cluster these instances into distinct groups. The ultimate goal of the clustering step is to bring instances together that prefer to be solved by the same solver. An automatic parameter tuner then finds the best parameters for the solver of each cluster. Given a new instance, its features are computed and used to assign the instance to the appropriate cluster where it is evaluated with the solver tuned for that particular cluster.

This three step approach is versatile and applicable to a number of problem types. Furthermore, the approach is independent of the precise algorithms employed for each step. This chapter first presents two clustering approaches that can be used, highlighting the strengths and weaknesses of each. The chapter then presents the two methods of tuning the solver. Due to the problem specific nature, the feature computation will be presented in the Chapter 4.

3.1 Clustering the Instances

There are many clustering techniques available in recent research[9]. This section, however, first presents how to define the distance metric which is important regardless of the clustering method employed. The section then presents the two clustering approaches initially tested for ISAC.

3.1.1 Motivation

One of the underlying assumptions behind ISAC is that there are groups of similar instances, all of which can be solved efficiently by the same solver. The dissertation further postulates that these similarities can be identified automatically. Figure 3.1 highlights the validity of these assumptions. The figures are based on the standard 48 SAT features (which will be introduced in detail in Chapter 4) for 5,347 instances from the 2002-2009 SAT Comptetions [2]. The features were then normalized and using PCA, projected into 2 dimensions. We ran 37 solvers available in 2009 with a 5,000 second timeout and recorded the best possible time for each instance. Figure 3.1 shows the performance of two of these solvers (clasp [31] and AshiQCP a parameterization of SATenstein [55]). In the figure, an instance is referred to as "good" if the runtime of the solver on this instance was no worse than 25% more time than the best recorded time for that instance. All other instances are deemed to be "ok" unless the solver timed-out.

First thing that stands out is that there is a clear separation between the industrial instances on the left and the randomly generated instances on the right. Furthermore, what is also interesting to note from this experiment is that clear clusters can be seen where a solver is really good on the instances, and other clusters where it is really bad. To take AshiQCP as an example, the solver seems to have poor performance on most instances

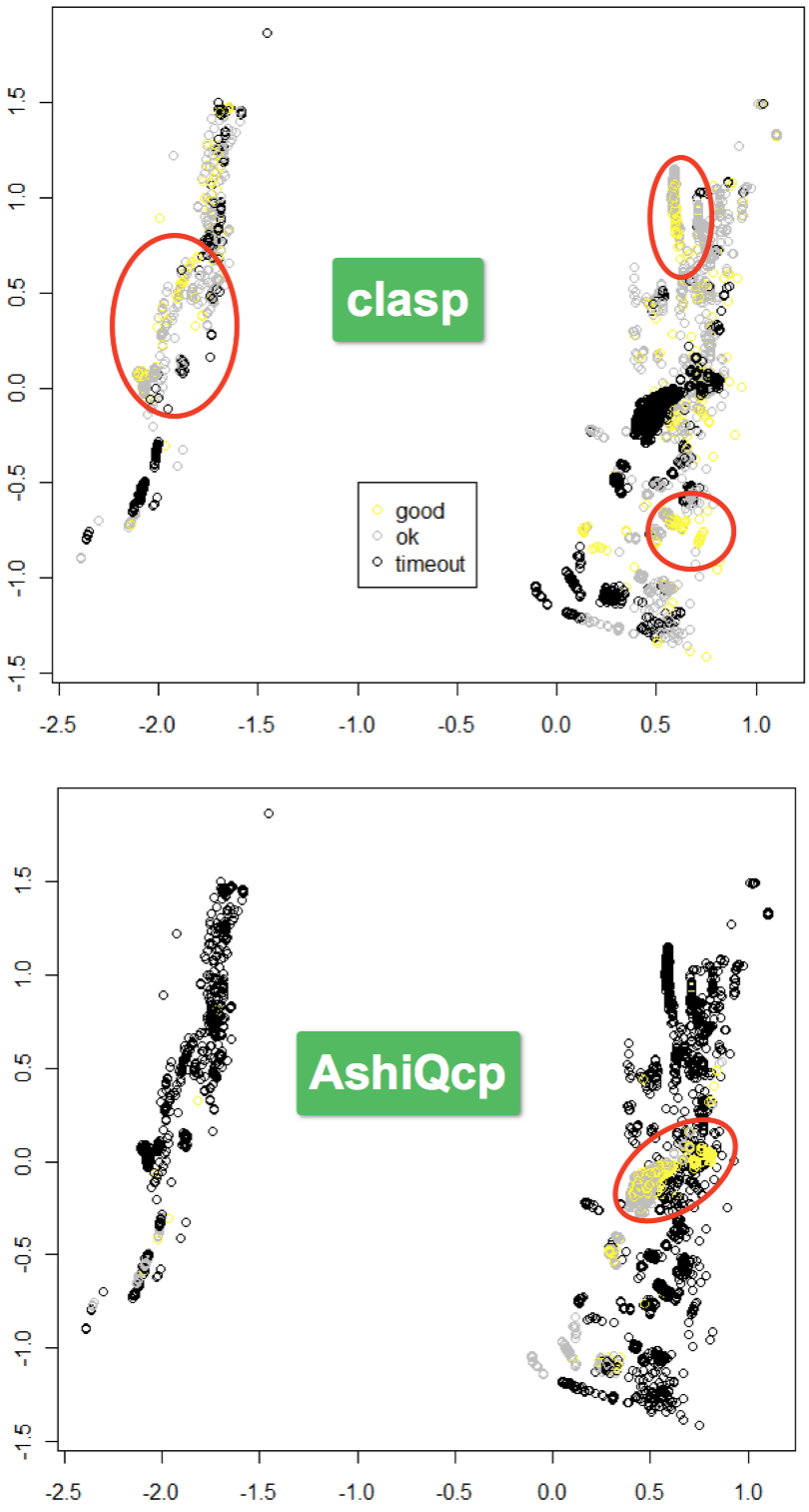

Figure 3.1: Performance of clasp and AshiQCP (a parameterization of SATenstein) on 5,437 SAT instances. A feature vector was computed for each instance and the projected into 2D using PCA. A good instance is one that performs no worse than 25% slower than the best solver on that instance. An ok instance is one that is more than 25% worse than the best solver. An instance that takes more than 5,000 seconds is marked as a timeout.

except for a small group of instances from the Randomly generated categories of the SAT competitions. Clasp on the other hand has completely different sets of regions where it performs very well. This behavior is seen repeated for all of the 37 solvers that were tested and is a strong motivator for pursuing clustering for algorithm configuration.

3.1.2 Distance metric

The quality of a clustering algorithm strongly depends on how the distance metric is defined in the feature space. Features are not necessarily independent. Furthermore, important features can range between small values while features with larger ranges could be less important. Finally, some features can be noisy, or worse, completely useless and misleading. For the current version of ISAC, however, it is assumed that the features are independent and not noisy. Chapter 6 will show how to handle situations where this is not the case.

A weighted Euclidean distance metric can handle the case where not all features are equally important to a proper clustering. This metric also handles the case where the ranges of the features vary wildly. To automatically set the weights for the metric an iterative approach is needed. Here all the weights were first set to one and the training instances were clustered accordingly. Once the solvers have been tuned for each cluster, the quality of the clusters was evaluated. To this end, for each pair of clusters $i \neq j$, the difference was computed between the performance on all instances in cluster i which is achieved by the solver for that cluster and the solver of the other cluster. The distance between an instance a in cluster C_i and the centers of gravity of cluster C_j is then the maximum of this regret and 0. Using these desired distances, the feature metric is adjusted and the process continues to iterate until the feature metric stops changing.

This iterative approach works well when improving a deterministic value like the solution quality, where it is possible to perfectly assess algorithm performance. The situation changes when the objective is to minimize runtime. This is because parameter sets that are not well suited for an instance are likely to run for a very long time, necessitating the need to introduce a timeout. This then implies that the real performance is not always known, and all that can be used is the lower bound. This complicates learning a new metric for the feature space. In the experiments, for example, it was found that most instances from one cluster timed out when run with the parameters of another. This not only leads to poor feature metrics, but also costs a lot of processing time. Furthermore, because runtime is often a noisy measurement, it is possible to encounter a situation where instances oscillate between two equally good clusters. Finally, this approach is very computationally expensive, requiring several retuning iterations which can take CPU days or even weeks for each iteration.

Consequently, for the purpose of tuning the speed of general solvers this dissertation suggests a different approach. Instead of learning a feature metric over several iterations, the features are normalized using translation and scaling so that, over the set of training instances, each feature spans exactly the interval $[-1, 1]$. That is, for each feature there exists at least one instance for which this feature has value 1 and at least one instance where the feature value is -1. For all other instances, the value lies between these two extremes. By normalizing the features in this manner, it was found that features with large and small ranges are given equal consideration during clustering. Furthermore, the

Algorithm 1: k-Means Clustering Algorithm

1: k-**Means(**X**,** k**)**
2: Choose k random points $C_1, \ldots, C_k$ from X.
3: **while** not done **do**
4: **for** $i = 1, \ldots, k$ **do**
5: $S_i \leftarrow \{j \ : \ ||X_j - C_i|| \leq ||X_j - C_l|| \ \forall \ l = 1, \ldots, k\}$
6: $C_i \leftarrow \frac{1}{|S_i|} \sum_{j \in S_i} X_j$
7: **end for**
8: **end while**
9: **return** (C, S)

assumption that there were no noisy or bad features did not result in bad clusterings. However, Chapter 6 shows how filtering can be applied to further improve performance.

3.1.3 k-means

One of the most straightforward clustering algorithms is Lloyd's k-means[63]. As can be seen in Algorithm 1, the algorithm first selects k random points in the feature space. It then alternates between two steps until some termination criterion is reached. The first step assigns each instance to a cluster according to the shortest distance to one of the k points that were chosen. The next step then updates the k points to the centers of the current clusters.

While this clustering approach is very intuitive and easy to implement, the problem with k-means clustering is that it requires the user to specify the number of clusters k explicitly. If k is too low, this means that some of the potential is lost to tune parameters more precisely for different parts of the instance feature space. On the other hand, if there are too many clusters, the robustness and generality of the parameter sets that are optimized for these clusters is sacrificed. Furthermore, for most training sets, it is unreasonable to assume that the value of k is known.

3.1.4 g-means

In 2003, Hamerly and Elkan proposed an extention to k-means that automatically determines the number of clusters[37]. This work proposes that a good cluster exhibits a Gaussian distribution around the cluster center. The algorithm, presented in Algorithm 2, first considers all inputs as forming one large cluster. In each iteration, one of the current clusters is picked and is assessed whether it is already sufficiently Gaussian. To this end, g-means splits the cluster in two by running 2-means clustering. All points in the cluster can then be projected onto the line that runs through the centers of the two sub-clusters, obtaining a one-dimensional distribution of points. g-means now checks whether this distribution is normal using the widely accepted statistical Anderson-Darling test. If the current cluster does not pass the test, it is split into the two previously computed clusters, and the process is continued with the next cluster.

It was found that the g-means algorithm works very well for our purposes, except sometimes clusters can be very small, containing very few instances. To obtain robust pa-

Algorithm 2: g-means Clustering Algorithm

g-Means(X)
$k \leftarrow 1$, $i \leftarrow 1$
$(C, S) \leftarrow k$-Means(X, k)
while $i \leq k$ **do**
  $(\bar{C}, \bar{S}) \leftarrow k$-Means$(S_i, 2)$
  $v \leftarrow \bar{C}_1 - \bar{C}_2$, $w \leftarrow \sum v_i^2$
  $y_i \leftarrow \sum v_i x_i / w$
  if Anderson-Darling-Test(y) failed **then**
    $C_i \leftarrow \bar{C}_1$, $S_i \leftarrow \bar{S}_1$
    $k \leftarrow k + 1$
    $C_k \leftarrow \bar{C}_2$, $S_k \leftarrow \bar{S}_2$
  else
    $i \leftarrow i + 1$
  end if
end while
return (C, S, k)

rameter sets we do not allow clusters that contain fewer than a manually chosen threshold, a value which depends on the size of the data set. Beginning with the smallest cluster, the corresponding instances are redistributed to the nearest clusters, where proximity is measured by the Euclidean distance of each instance to the cluster's center.

3.2 Training Solvers

Once the training instances are separated into clusters, the parameterized solver must be tuned for each cluster. As shown in existing research [49], manual tuning is complex and laborious process that usually results in sub-par performance of the solver. This section introduces two algorithms used to tune the parameters in our experiments.

3.2.1 Local Search

With automatic parameter tuning being a relatively new field, there are not many off-the-shelf tuners available. Furthermore, some problems seem to be outside the scope of existing tuners requiring the development of problem specific tuners. One such scenario is when the parameters of the solvers are a probability distribution; where the parameters are continuous variables between 0 and 1 and sum up to one. For this kind of problem we developed [65] a local search shown in Algorithm 3.

This search strategy is presented with an algorithm A for a combinatorial problem as well as a set S of training instances. Upon termination, the procedure returns a probability distribution for the given algorithm and benchmark set.

The problem of computing this favorable probability distribution can be stated as a continuous optimization problem: Minimize$_{\text{distr}} \sum_{i \in \text{S}}$ Perf(A,distr,i) such that 'distr' is a probability distribution used by A. Each variable of the distribution is initialized randomly

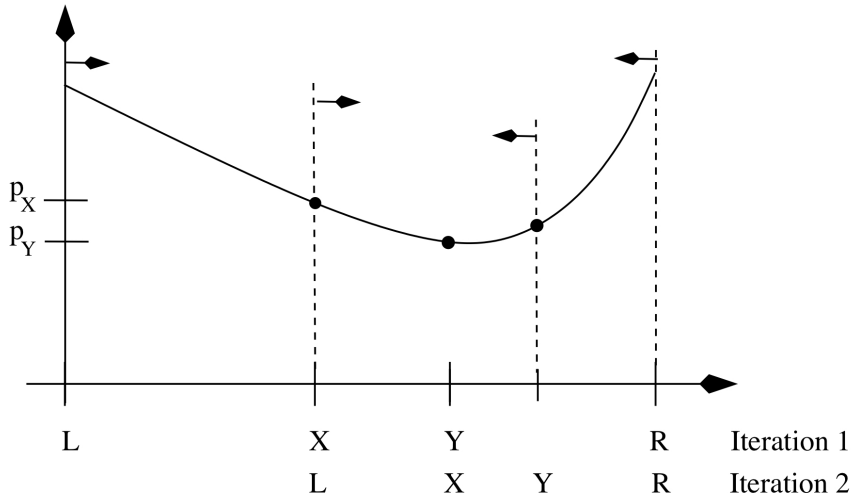

Figure 3.2: Minimizing a One-Dimensional Convex Function by Golden Section.

Algorithm 3: Local Search for tuning variables that are part of a probability distribution.

LSTuner(Algorithm A, BenchmarkSet S)
distr ← RandDistr()
$\lambda_l \leftarrow \frac{\sqrt{5}-1}{\sqrt{5}+1}$, $\lambda_r \leftarrow \frac{2}{\sqrt{5}+1}$
while termination criterion not met **do**
  $(a, b) \leftarrow$ ChooseRandPair(), m ← distr$_a$+ distr$_b$
  $X \leftarrow \lambda_l$, $Y \leftarrow \lambda_r$
  $L \leftarrow 0$, $R \leftarrow 1$, length ← 1
  $p_X \leftarrow \sum_{i \in \mathrm{S}}$ Perf(A,distr[a=m X,b=m $(1-X)$], i)
  $p_Y \leftarrow \sum_{i \in \mathrm{S}}$ Perf(A,distr[a=m Y,b=m $(1-Y)$], i)
  while length $> \varepsilon$ **do**
    if $p_X < p_Y$ **then**
      $p_Y \leftarrow p_X$
      $R \leftarrow Y$, length ← $R - L$
      $Y \leftarrow X$, $X \leftarrow L + \lambda_l$ length
      $p_X \leftarrow \sum_{i \in \mathrm{S}}$ Perf(A,distr[a=m X,b=m $(1-X)$], i)
    else
      $p_X \leftarrow p_Y$
      $L \leftarrow X$, length ← $R - L$
      $X \leftarrow Y$, $Y \leftarrow L + \lambda_r$ length
      $p_Y \leftarrow \sum_{i \in \mathrm{S}}$ Perf(A,distr[a=m Y,b=m $(1-Y)$], i)
    end if
  end while
  distr ← distr[a=m X,b=m $(1-X)$]
end while
return distr

and then normalized so that all variables sum up to one. In each iteration, two variables a, b are picked randomly and their joint probability mass m is redistributed among themselves while keeping the probabilities of all other advisors the same.

It is expected that the one-dimensional problem which optimizes which percentage of m is assigned to advisor a (the remaining percentage is determined to go to advisor b) is convex. The search seeks the best percentage using a method for minimizing one-dimensional convex functions over closed intervals which is based on the golden section (see Figure 3.2): two points are considered $X < Y$ within the interval $[0, 1]$ and their performance is measured as 'p_X' and 'p_Y.' The performance at X is assessed by running the algorithm A on the given benchmark with distribution 'distr [a=m X,b=m $(1 - X)$]', which denotes the distribution resulting from 'distr' when assigning probability mass 'Xm' to variable a and probability mass '$(1-X)$m' to variable 'b'. Now, if the function is indeed convex, if $p_X < p_Y$ ($p_X \geq p_Y$), then the minimum of this one-dimensional function lies in the interval $[0, Y]$ ($[X, 1]$). The search continues splitting the remaining interval (which shrinks geometrically fast) until the interval size 'length' falls below a given threshold 'ε.' By choosing points X and Y based on the golden section, in each iteration only one new point needs to be evaluated rather than two. Moreover, the points considered at each iteration are reasonably far apart from each other to make a comparison meaningful which is important for as our function evaluation may be noisy (due to the randomness of the algorithm invoked) and points very close to each other are likely to produce very similar results.

3.2.2 GGA

One drawback to developing proprietary tuning algorithms is the difficulty of trasferring the technique between problem types. To test a more general procedure, the Gender-Based Genetic Algorithm (GGA[4]), a state-of-the-art automatic parameter tuner, is explored. This tuner uses a genetic algorithm to find the parameters for a specified solver. Representing the parameters in an and-or tree, the tuner randomly generates two populations of possible parameter configurations. These two groups are classified as being *competitive* or $non - competitive$. A random sub-set of the individuals from the competitive population are selected and run against each other over a subset of the training instances. This tournament is repeated several times until all members of the competitive population participated in exactly one tournament. Each member of the non-competitive competition is mated with one of the tournament winners. This process is repeated for one hundred iterations when the best parameter setting is returned as the parameter set to be used by the solver.

In this scenario the parameters of the tuned solver are represented as an and-or tree (Figure 3.3). This representation allows the user to specify the relation between the parameters. For example, parameters that are independent are separated by an *and* parent. On the other hand if a parameter depends on the setting of another parameter it is defined as a child of that parameter. This representation allows GGA to better search the parameter space by maintaining certain parameter settings constant as a group instead of randomly changing different parameters.

Each mating of a couple results in one new individual with a random gender. The genome of the offspring is determined by traversing the variable tree top-down. A node

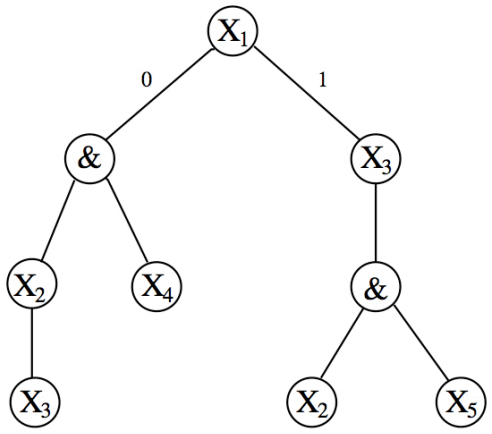

Figure 3.3: And-or tree used by GGA representing the parameters of the tuned algorithm.

can be labelled O ("open"), C ("competitive"), or N ("non-competitive"). If the root is an *and*-node, or if both parents agree on the value of the root-variable, it is labeled O. Otherwise, the node is labeled randomly as C or N. The algorithm continues by looking at the children of the root (and so on for each new node). If the label of the parent node is C (or N) then with high probability P% the child is also labeled with C (N), otherwise the label is switched. By default P is set to 90%.

Finally, the variable assignment associated with the offspring is given by the values from the C (N) parent for all nodes labelled C (N). For variable-nodes labelled O both parents agree on its value, and this value is assigned to the variable. Note that this procedure effectively combines a uniform crossover for child-variables of open *and*-nodes in the variable tree (thus exploiting the independence of different parts of the genome) and a randomized multiple-point crossover for variables that are more tightly connected.

As a final step to determine the offspring's genome, with low probability M% each variable is mutated. By default M is set to 10%. When mutating a categorical variable, the new value in the domain is chosen uniformly at random. For continuous and integer variables, the new value is chosen according to a Gaussian distribution where the current value marks the expected value and the variance is set as 10% of the variable's domain.

3.3 ISAC

Ultimately the ISAC methodology is summarized in Algorithm 4 where the application of all three components is displayed. Particularly, given a parameterized algorithm A, a list of training instances T, and their corresponding feature vectors F. First, the features in the set are normalized and the scaling and translation values are memorized for each feature (s, t).

Then, an algorithm is used to cluster the training instances based on the normalized feature vectors. The final result of the clustering is a number of k clusters S_i, a list of

Algorithm 4: Instance-Specific Algorithm Configuration

ISAC-Learn(A, T, F)
$(\bar{F}, s, t) \leftarrow$ Normalize(F)
$(k, C, S, d) \leftarrow$ Cluster $(T, \bar{F})$
for all $i = 1, \ldots, k$ **do**
  $P_i \leftarrow Train(A, S_i)$
end for
$R \leftarrow Train(A, T)$
return (k, P, C, d, s, t, R)

ISAC-Run$(A, x, k, P, C, d, s, t, R)$
$f \leftarrow$ Features(x)
$\bar{f}_i \leftarrow (f_i - t_i)/s_i \; \forall \; i$
for all $j = 1, \ldots, k$ **do**
  if $||\bar{f} - C_i|| \leq d_i$ **then**
    return $A(x, P_i)$
  end if
end for
return $A(x, R)$

cluster centers C_i, and, for each cluster, a distance threshold d_i which determines when a new instance will be considered as close enough to the cluster center to be solved with the parameters computed for instances in this cluster.

Then, for each cluster of instances S_i favorable parameters P_i are computed using an instance-oblivious tuning algorithm. After this is done, the parameter set R is computed for all the training instances. This serves as the recourse for all future instances that are not near any of the clusters.

When running algorithm A on an input instance x, we first compute the features of the input and normalize them using the previously stored scaling and translation values for each feature. Then, we determine whether there is a cluster such that the normalized feature vector of the input is close enough to the cluster center. If so, A is run on x using the parameters for this cluster. If the input is not near enough to any of our clusters the instance-oblivious parameters R are used, which work well for the entire training set.

3.4 Chapter Summary

This chapter presents the components of the proposed instance specific automatic parameter tuner, ISAC. The approach partitions the problem of automatic algorithm configuration into three distinct pieces. First, the feature values are computed for each instance. Second, the training instances are clustered into groups of instances that have similar features. Finally, an automatic parameter tuner is used to find the best parameters for the solver for each cluster. This chapter shows two basic configurations of the last two steps of ISAC. Being problem specific, the features used for clustering are explained in the numerical section of the subsequent chapters.

Chapter Four

Training Parameterized Solvers

This chapter details the numerical results of applying ISAC on three different types of combinatorial optimization problems. The first section covers the set covering problem, showing that instance-oblivious tuning of the parameters can yield significant performance improvements and that ISAC can perform better than a istance-specific regression approach. The second section presents the mixed integer problem and shows that even a state-of-the-art solver like Cplex can be improved through instance specific tuning. The third section introduces the Satisfiability problem and shows how an algorithm portfolio can be enhanced by the proposed approach. The chapter concludes with a brief summary of the results and benefits of the ISAC approach.

Unless otherwise noted, experiments were run on dual processor dual core Intel Xeon 2.8 GHz computers with 8GB of RAM. SCP solvers Hegel and Nysret were evaluated on quad core dual processor Intel Xeon 2.53 Ghz processors with 24GB of RAM.

4.1 Set Covering Problem

The empirical evaluation begins with one of the most studied combinatorial optimization problems: the set covering problem (SCP). In SCP, given a finite set $S := \{1, \ldots, n\}$ of items, a family $F := \{S_1, \ldots, S_m \subseteq S\}$ of subsets of S, and a cost function $c : F \rightarrow \mathbb{R}^+$, the objective is to find a subset $C \subseteq F$ such that $S \subseteq \bigcup_{S_i \in C} S_i$ and $\sum_{S_i \in C} c(S_i)$ is minimized. In the *unicost* SCP, the cost of each set is set to one. This problem formulation appears in numerous practical applications such as crew scheduling [41, 44, 22], location of emergency facilities [97], and production planning in various industries [99].

In accordance to ISAC, the first step deals with the identification of a set of features that accurately distinguish the problem instances. Following the process introduced in [65], the features are generated by computing the maxima, minima, averages, and standard deviations of the following vectors:

- vector of normalized subset costs $c' \in [1, 100]^m$,
- vector of subset densities $(|S_i|/n)_{i=1\ldots m}$,
- vector of item costs $(\sum_{i \| j \in S_i} c'_i)_{j=1\ldots n}$,
- vector of item coverings $(|\{i \mid j \in S_i\}|/m)_{j=1\ldots n}$,
- vector of costs over density $(c'_i/|S_i|)_{i=1\ldots m}$,
- vector of costs over square density $(c'_i/|S_i|^2)_{i=1\ldots m}$,
- vector of costs over $k \log k$-density $(\frac{c'_i}{(|S_i| \log |S_i|)})_{i=1\ldots m}$, and
- vector of root-costs over square density $(\sqrt{c'_i}/|S_i|^2)_{i=1\ldots m}$.

Computation of these feature values on average takes only 0.01 seconds per instance.

Due to a sparsity of well established benchmarks for set covering problems, a new highly diverse set of instances is generated. Specifically, a large collection of instances is randomly generated, each comprised of 100 items and 10,000 subsets. To generate these instances an

SCP problem is considered as a binary matrix where each column represents an item and each row represents a subset. A one in this matrix corresponds to an item being included in the subset. The instance generator then randomly makes three decisions. One, to fill the matrix by either row or column. Two, if the density (ratio of ones to zeroes) of the row or column is constant, has a mean of 4%, or has a mean of 8%. Three, whether the cells to be set to one are chosen uniformly at random or with a Gaussian bias centered around some cell. The cost of each subset is chosen uniformly at random from $[1, 1000]$. For the unicost experiments, all the subset costs are reset to 1. The final data set comprises 200 training instances and 200 test instances.

4.1.1 Solvers

Due to the popularity of SCP, a rich and diverse collection of algorithms was developed to tackle this problem. To analyze the effectiveness and applicability of the approach three orthogonal approaches are focussed on that cover the spectrum of incomplete algorithms. Relying on local search strategies, these algorithms are not guaranteed to return optimal solutions. The first algorithm is the greedy randomized set covering solver from [65]. This approach repeatedly adds subsets one at a time until reaching a feasible solution. The decision of which subset to add next is determined by one of the following six heuristics, chosen randomly during the construction of the cover:

- The subset that costs the least ($\min c$).
- The subset that covers the most new items ($\max k$).
- The subset that minimizes the ratio of costs over the number of newly covered items ($\min c/k$).
- The subset that minimizes the ratio of costs over newly covered items times the logarithm of newly covered items ($\min \frac{c}{k \log k}$).
- The subset that minimizes the ratio of costs over the square of newly covered items ($\min \frac{c}{k^2}$).
- The subset that minimizes the ratio of square root of costs over the square of newly covered items ($\min \frac{\sqrt{c}}{k^2}$).

The second solver, "Hegel"[54], uses a specialized type of local search called dialectic search. Designed to optimally balance exploration and exploitation, dialectic search begins with two greedily obtained feasible solution called the "thesis" and "antithesis" respectively. Then a greedy walk traverses from the thesis to the antithesis; first removing all subsets from the solution of the thesis that are not in the antithesis and then greedily adding the subsets from the antithesis that minimize the overall cost. The Hegel approach was shown to outperform the fastest algorithms on a range of SCP benchmarks.

The third solver, Nysret [70], uses an alternate type of local search algorithm called tabu search. A greedily obtained feasible solution defines the initial state. For each consequent step, the neighborhood is composed of all states obtained by adding or removing one subset from the current solution. The fitness function is then evaluated as the cumulative cost

Approach	% Optimality Gap Closed	
	Train	Test
Un-tuned GRS solver	25.9 (4.2)	40.0 (4.1)
GRS with Instance-specific regression	32.8 (3.6)	38.1 (3.7)
GRS with Instance-oblivious tuning	**40.0 (3.6)**	**46.1 (3.8)**

Table 4.1: Comparison of the default assignment of the greedy randomized solver (GRS) with parameters found by the instance-specific multinomial regression tuning approach and an instance-oblivious parameter tuning approach. The table shows the percent of the optimality gap closed over using a single best heuristic. The standard deviation is presented in parentheses.

of all the included subsets plus the number of the uncovered items. The neighbor with the lowest cost is chosen as the starting state for the next iteration. During this local search, the subsets that are included or removed are kept track of in the tabu list for a limited number of iterations. To prevent cycles in the local search, neighbors that change the status of a subset in the tabu list are excluded from consideration. In 2006, Nysret was shown empirically to be the best solver for unicost SCP.

4.1.2 Numerical Results

This section presents three results. First it compares the performance of the instance-specific tuning approach of multinomial regression to an instance-oblivious parameter tuning. Showing the strength of parameter tuning, two configurations of ISAC are then presented, compared to the instance-oblivious tuning approach. The section concludes by showing that the ISAC approach can be applied out of the box to two state-of-the-art solvers and results in significant improvements in performance.

Table 4.1 compares the effectiveness of parameter tuning to the classical multinomial regression approach. The experiment is performed on the greedy randomized solver, where the parameters are defined as the probabilities of each heuristic being chosen. It was found that by using only one heuristic during a greedy search leaves, on average, a 7.2% (7.6%) optimality gap on the training (testing) data [65]. A default assignment of equal probabilities to all heuristics can close up to 40% of this gap on the test instances. For the multinomial regression approach, the algorithm learns a function for each parameter that converts the instance feature vector to a single value, called score. These scores are then normalized to sum to 1 to create a valid probability distribution. However, while this approach leads to some improvements on the training set, the learned functions are not able to carry over to the test set, closing only 38.1% of the optimality gap. Training on all the instances simultaneously, using a state-of-the-art parameter tuner like GGA leads to superior performance on both training and test sets. This result emphasized the effectiveness of multi-instance parameter tuners over instance-specific regression models.

As stated in the previous chapter, straight application of a parameter tuner ignores the diversity of instances that might exist in the training and test sets. ISAC addresses this issue by introducing a cluster-based training approach consisting of three steps: computation of features, clustering of training instances, and cluster based parameter tuning.

Approach		% Optimality Gap Closed	
		Train	Test
GRS with Instance-oblivious tuning		40.0 (3.6)	46.1 (3.8)
GRS with ISAC	Configuration 1	**47.7 (2.4)**	50.3 (3.7)
	Configuration 2	44.4 (3.3)	**51.3 (3.8)**

Table 4.2: Comparison of two versions of ISAC to an instance-oblivious parameter tuning approach. The table shows the percent of the optimality gap closed over a greedy solver that only uses the single best heuristic throughout the construction of the solution. The standard deviation is presented in parentheses.

The first configuration of ISAC[65] uses a weighted Euclidean distance for the features, k-means for clustering, and a proprietary local search for parameter tuning. To set the distance metric weights, this configuration iterated the clustering and tuning steps, trying to minimize the number of training instances yielding better performance for solvers tuned on another cluster. The next configuration[53] built off the first attempt, streamlining each part of the ISAC procedure. As a result, this configuration normalized the features, used g-means for clustering, and GGA for parameter tuning.

As can be seen in Table 4.2, both configurations of ISAC improve on the performance of a solver tuned on all instances. This highlights the benefit of clustering the instances before training. Furthermore, while the numerical results of both configurations are relatively similar it is important that the second is a much more efficient and general of the two. The first configuration was designed specifically to tune the greedy SCP solver and required multiple tuning iterations to achieve the observed result. The second configuration on the other hand uses out of the box tools that can be easily adapted to any solver. It also only requires one clustering and tuning iteration which makes it much faster than the first. Because of its versatility, unless otherwise stated all further comparisons to ISAC refer to the second configuration.

To explore the ISAC approach, it is next evaluated on two state-of-the-art local search SCP solvers; Hegel and Nysret. For both solvers the time to find a set covering solution that is within 10% of optimal is measured. Hegel and Nysret had a timeout of 10 seconds during training and testing. Table 4.3 compares the default configuration of the solvers, the instance-oblivious configuration obtained by GGA, and the instance-specific tuned versions found by ISAC. To provide a more holistic view of ISAC's performance, three evaluation metrics are presented: the arithmetic and geometric means of the runtime in seconds and the average slow down (the arithmetic mean of the ratios of the performance of the competing solver over ISAC). For these experiments the size of the smallest cluster is set to be at least 30 instances. This setting resulted in 4 clusters of roughly equal size.

The first experiments show that the default configuration of both solvers can be improved significantly by automatic parameter tuning. For the Nysret solver, an arithmetic mean runtime of 2.18 seconds for ISAC-Nysret, 3.33 seconds for GGA-Nysret, and 3.44 seconds for default are measured. That is, instance-oblivious parameters run 50% slower than instance-specific parameters. For Hegel, it is found that the default version runs more than 60% slower than ISAC-Hegel.

Solver		Avg. Run Time		Geo. Avg.		Avg. Slow Down	
		Train	Test	Train	Test	Train	Test
Nysret	Default	2.79	3.45	2.36	2.60	1.49	1.79
	GGA	2.58	3.40	2.27	2.63	1.35	1.72
	ISAC	**1.99**	**2.04**	**1.96**	**1.97**	**1.00**	**1.00**
Hegel	Default	3.04	3.15	2.52	2.49	2.20	2.03
	GGA	1.58	1.95	**1.23**	**1.33**	1.10	1.15
	ISAC	**1.45**	**1.92**	**1.23**	1.36	**1.00**	**1.00**

Table 4.3: Comparison of default, instance-oblivious parameters provided by GGA, and instance-specific parameters provided by ISAC for Hegel and Nysret. The table presents the arithmetic and geometric mean runtimes in seconds, as well as the average degradation when comparing each solver to ISAC.

It is worth noting the high variance of runtimes from one instance to another, which is caused by the diversity of our benchmark. To get a better understanding, the average slow down of each solver when compared with the corresponding ISAC version is provided. For this measure we find that, for an average test instance, default Nysret requires more than 1.70 times the time of ISAC-Nysret, and GGA-Nysret needs 1.62 times over ISAC-Nysret. For default Hegel, an average test instance takes 2.10 times the time of ISAC-Hegel while GGA-Hegel only runs 10% slower. This confirms the findings in [54] that Hegel runs robustly over different instance classes with one set of parameters.

It is concluded that even advanced, state-of-the-art solvers can greatly benefit from ISAC. Depending on the solver, the proposed method works as well or significantly better than instance-oblivious tuning. Note that this is not self-evident since the instance-specific approach runs the risk of over-tuning by considering fewer instances per cluster. In these experiments, these problems are not observed. Instead it is found that the instance-specific algorithm configurator offers the potential for great performance gains without over-fitting the training data.

4.2 Mixed Integer Programming

An NP-hard problem, mixed integer programming (MIP) involves optimizing a linear objective function while obeying a collection of linear inequalities and variable integrality constraints. Mixed integer programming is an area of great importance in operations research as they can be used to model just about any discrete optimization problem. They are used especially heavily to solve problems in transportation and manufacturing: airline crew scheduling, production planning, vehicle routing, etc.

Even though solving MIPs is an active field, to the best of our knowledge no prior research exists on the type of features that can be used to classify a MIP instance. It is therefore proposed to use the information about the objective vector, the right hand side (RHS) vector, and the constraint matrix formulate the feature vector. The following general statistics on the variables in the problem are computed:

- number of variables and number of constraints,
- percentage of binary (integer or continuous) variables,
- percentage of variables (all, integer, or continuous) with non zero coefficients in the objective function, and
- percentage of $\leq$ ($\geq$ or $=$) constraints.

The mean, min, max, and standard deviation of the following vectors are also used, where $U = Z \cup R$, $R = \{x_i \mid x_i \text{ is real valued}\}$, and $Z = \{x_i \mid x_i \text{ is restricted to be integer}\}$. These vectors focus on the actual coefficient values of each of the variables:

- vector of coefficients of the objective function (of all, integer, or continuous variables): $(c_i | x_i \in X)$ where $X = U \vee X = Z \vee X = R$,
- vector of RHS of the $\leq$ ($\geq$ or $=$) constraints: $(b_j | A_j x \circ b_j)$ where $\circ = (\geq) \vee \circ = (\leq) \vee \circ = (=)$,
- vector of number of variables (all, integer or continuous) per constraint j: $(\#\{A_{(i,j)} \mid A_{(i,j)} \neq 0, x_i \in X\})$ where $X = U \vee X = Z \vee X = R$,
- vector of the coefficients of variables (all, integer, or continuous) per constraint j: $(\sum_i A_{(i,j)} | \forall j, x_i \in X)$ where $X = U \vee X = Z \vee X = R$, and
- vector of the number of constraints each variable i (all, integer, or continuous) belongs to: $(\#\{A_{(i,j)} \mid A_{(i,j)} \neq 0, x_i \in X\}$ where $X = U \vee X = Z \vee X = R$.

Computation of these feature values on average took only 0.02 seconds per instance.

MIPs are used to model a wide variety of problem types. Therefore, in order to capture the spectrum of possible instances we assembled a highly diverse benchmark data set composed of problem instances from six different sources. Network flow instances, capacitated facility location instances, bounded and unbounded mixed knapsack instances and capacitated lot sizing problems, all taken from [82], as well as combinatorial auction instances from [59]. In total there are 588 instances in this set, which was split into 276 training and 312 test instances.

- Given some graph, the *network flow problem* aims to find the maximal flow that can be routed from node s to node t while adhering to the capacity constraints of each edge. The interesting characteristic of these problems is that special-purpose network flow algorithms can be used to solve such problems much faster than general-purpose solvers.
- In the *capacity facility problem*, a collection of demand points and a distance function are defined. The task is then to place n supply nodes that minimize some distance objective function while maintaining that each supply node does not service too many demand points. Problems of this type are generally solved using Lagrangen relaxation and matrix column generation methods.

- The *knapsack problem* is a highly popular problem type that frequently appears in real-world problems. Given a collection of items, each with an associated profit and weight, the task of a solver is to find a collection of items that results in the highest profit while remaining below a specified weight capacity constraint. In the bounded knapsack version there are multiple copies of each item while in the unbounded version there is an unlimited number of copies of each item. Usually these types of problems are solved using a branch-and-bound approach.

- The task of the *capacitated lot sizing problem* is to determine the amount and timing of the production of products to generate a production plan that best satisfies all the customers. Specifically, at each production step, certain items can be produced using a specific resource. Switching the available resource incurs a certain price as does maintaining items in storage. The problem also specifies the number of copies of each item that need to be generated and by what time. This is a very complex problem that in practice usually is defined as a MIP.

- In a *combinatorial auction* participants place bids on combinations of discrete items rather just on a single item. These auctions have been traditionally used to auction estates, but has recently also been applied to truckload transportation and bus routes. Another important recent application was the auction of the radio spectrum for wireless communications. The problem specification is given a collection of bids, find the most profitable allocation of items to bidders. In practice these problems usually are modeled as the set packing problem.

4.2.1 Solver

For these experiments Cplex 12.1 is used. For 15 years, IBM Cplex [52] has represented the state-of-the-art optimization package. Used by many of the worlds leading commercial firms and researchers in over 1,000 universities, Cplex has become a critical part of optimization research. Although the specific techniques and implementations are kept proprietary, the solver provides flexible, high-performance optimizers for solving linear programming, mixed integer programming, quadratic programming, and constraint programming problems. For each of these problem types, Cplex can handle problems with millions of constraints and variables, often setting performance records. The solver also has numerous options for tuning solving strategies for specific problems, which makes it ideal for the purposes of this dissertation.

4.2.2 Numerical Results

Experiments were carried out with a timeout of 30 seconds for training and 300 seconds for evaluation on the training and testing sets. The size of the smallest cluster is set to be 30 instances. This resulted in 5 clusters, where 4 consisted of only one problem type, and 1 cluster combined network flow and capacitated lot sizing instances.

Table 4.4 compares instance-specific ISAC with instance-oblivious GGA and the default settings of Cplex. it is observed again that the default parameters can be significantly improved by tuning the algorithm for a representative benchmark. For the average test instance ISAC-Cplex needs 3.4 seconds, GGA-Cplex needs 5.2 seconds and default Cplex

Solver		Avg. Run Time		Geo. Avg.		Avg. Slow Down	
		Train	Test	Train	Test	Train	Test
Cplex	Default	6.1	7.3	2.5	2.5	2.0	1.9
	GGA	3.6	5.2	1.7	1.8	1.3	1.2
	ISAC	**2.9**	**3.4**	**1.5**	**1.6**	**1.0**	**1.0**

Table 4.4: Comparison of ISAC versus the default and the instance-oblivious parameters provided by GGA when tuning Cplex. The table presents the arithmetic and geometric mean runtimes as well as the average slowdown per instance.

requires 7.3 seconds. Instance-obliviously tuned Cplex is 50% slower, and default Cplex even more than 114% slower than ISAC-Cplex.

The improvements achieved by automatic parameter tuning can be seen when considering the average per-instance slow-downs. According to this measure, for a randomly chosen instance in the test set it is expected that GGA-Cplex needs 1.2 times the time required by ISAC-Cplex. Default Cplex needs 1.9 times the time of ISAC-Cplex.

It is necessary to note that due to license restrictions only a very small training set of 276 instances could be used, which is very few given the high diversity of the considered benchmark. Taking this into account and seeing that Cplex is a highly sophisticated and extremely well-tuned solver, the fact that ISAC boosts performance so significantly is surprising and shows the great potential of instance-specific tuning.

4.3 SAT

Our final evaluation of ISAC is on the propositional satisfiability problem (SAT), the prototypical NP-complete problem that has far reaching effects on many areas of computer science. For SAT, given a propositional logic formula F in conjunctive normal form, the objective is to determine whether there exists a satisfying truth assignment to the variables of F. In recent years, there has been tremendous progress in solving SAT problems, so that modern SAT solvers can now tackle instances with hundreds of thousands of variables and over one million clauses.

The well-established features proposed by [108] is used to classify each problem instance. However, in preliminary experiments it is found that the local search features mentioned in [108] take a considerable amount of time to compute and are not imperative to finding a good clustering of instances. Consequently, these features were excluded, and only the following are used:

- problem size features: number of clauses c, number of variables v, and their ratio c/v,
- variable-clause graph features: degree statistics for variable and clause nodes,
- variable graph features: node degree statistics,

- balance features: ratio of positive to negative literals per clause, ratio of positive to negative occurrences of each variable, fraction of binary and ternary clauses,
- proximity to horn clauses: fraction of horn clauses and statistics on the number of occurrences in a horn clause for each variable,
- unit propagations at depths 1, 4, 16, 64 and 256 on a random path in the DPLL [25] search tree, and
- search space size estimate: mean depth to contradiction and estimate of the log of number of nodes.

Computation of these feature values on average took only 0.01 seconds per instance.

Data set	**Train**	**Test**	**Ref.**
QCP	1000	1000	[34]
SWGCP	1000	1000	[32]
3SAT-random	800	800	[68]
3SAT-structured	1000	1000	[86]

Table 4.5: Data sets used to evaluate ISAC on SAT.

The collection of SAT instances described in Table 4.5 is considered. The objective of the quasigroup completion problem (QCP) is to determine if a partially filled N by N matrix can be filled with numbers $\{1...N\}$ so that the elements of each row are different and the elements of each column are different. As a benchmark we take a collection of QCP instances that have been encoded as a SAT instance. The graph coloring problem is given a graph G and N possible colors, the objective is to assign each node in the graph a color such that no adjacent nodes are assigned the same color. As one type of instances in our benchmarks a set of these graph coloring problems (SWGCP) that have been encoded as SAT instances are used. The third problem type in the benchmark are randomly generated SAT instances (3SAT-random) using the G2 generator [98]. Finally considered are randomly generated SAT instances (3SAT-structured) that more closely resemble industrial SAT instances by introducing structure into the instances.

4.3.1 Solver

ISAC is tested on the highly parameterized stochastic local search solver SAPS [51]. Unlike most existing SAT solvers, SAPS was originally designed with automatic tuning in mind and therefore all of the parameters influencing the solver are accessible readily to users. Furthermore, since it was first released, the default parameters of the solver have been improved drastically by general purpose parameter tuners [4, 49].

In addition to tuning a single solver, a portfolio like solver is created. This solver is comprised of nine competitive SAT solvers, where the job of ISAC is to identify not only which solver is best suited for the instance but also the best parameters for that solver. This is done by making the choice of the solver an additional categorical parameter to be

Solver		Avg. Run Time		Geo. Avg.		Avg. Slow Down	
		Train	Test	Train	Test	Train	Test
SAPS	Default	79.7	77.4	0.9	0.9	292.5	274.1
	GGA	14.6	14.6	0.2	0.2	5.5	4.7
	ISAC	**4.0**	**5.0**	**0.1**	**0.1**	**1.0**	**1.0**

Table 4.6: Comparison of the SAPS solvers with default, GGA tuned, and ISAC. The arithmetic and geometric mean runtimes in seconds are presented as well as the average slow-down per instance.

	clasp	jerusat	kcnfs	march	minisat	mxc	rsat	zchaf	ISAC	Oracle
Train	15.1	42.1	98.5	57.1	22.1	19.2	43.8	64.3	**5.9**	3.4
Test	16.8	39.9	95.6	51.9	22.7	20.1	42.9	64.6	**5.8**	2.9

Table 4.7: Performance of a portfolio style SAT solver tuned using ISAC compared to performance of each of the solvers in the portfolio. The table presents the average runtime. Oracle is a portfolio algorithm that always chooses the best solver for the given instance.

set by ISAC. The used solvers have all been ranked either a first, second or third place in a recent SAT competition: clasp 1.3.2, jerusat 1.3, kcnfs 2006, march pl, minisat 2.0, mxc 09, rsat 2.01 and zchaf.

4.3.2 Numerical Results

Experiments were carried out with a timeout of 30 seconds for training and 300 seconds for evaluation on the training and testing sets. The size of the smallest cluster is set to be at least 100 instances. This resulted in 18 clusters each with roughly 210 instances. Here not only were all of the 4 types of instances correctly separated into distinct clusters, a further partition of instances from the same class was provided.

The performance of SAPS was evaluated using the default parameters, GGA, and ISAC and present the results in Table 4.6.

Even though the default parameters of SAPS have been tuned heavily in the past [49], tuning with GGA solves the benchmark over 5 times faster than default SAPS. Instance-specific tuning allows us to gain another factor of 2.9 over the instance-oblivious parameters, resulting in a total performance improvement of over one order of magnitude. This refutes the conjecture of [47] that SAPS may not be a good solver for instance-specific parameter tuning.

It is worth noting that over 95% of instances in this benchmark can be solved in under 15 seconds. Consequently, some exceptionally hard, long-running instances greatly dilute the average runtime. The average slow-down per instance is therefore presented again. For the average SAT instance in our test set, default SAPS runs 274 times slower than ISAC. Even if GGA is used to tune a parameter set specifically for this benchmark, GGA is still

expected to run almost 5 times slower than ISAC.

Table 4.7 presents the performance of an algorithm portfolio style solver tuned using ISAC. The table shows that by creating an algorithm where one of the parameters identifies the solver to use to evaluate the instance, the resulting solver can perform 3 times better than using the best overall solver on all the instances. However, it can also be seen that there is still room for improvement. Oracle is a best case scenario algorithm portfolio that holistically always chooses the best solver for the given instance. This best case scenario still requires half the time of the algorithm tuned by ISAC.

4.4 Chapter Summary

This chapter presented the possible enhancements that can be achieved by using the cluster-based training approach ISAC. The experiments were done on three different optimization problem types and six different solvers. In all cases solvers trained using the ISAC approach outperformed there alternatively tuned counterparts. This chapter began by showing that instance-oblivious parameter tuning is a powerful technique that can yield better results than an instance-specific regression approach. It is then shown how by using the cluster-based approach, ISAC is able to enhance the instance-oblivious parameter tuning. The remaining experiments are aimed at applying ISAC to a variety of solvers, showing improvements for each.

Chapter Five

Training Portfolios

The constraint programming and satisfiability community has a long and proud tradition of introducing ideas that are highly relevant to constraint solving but also clearly go far beyond that scope. One such contribution was the inception of algorithm portfolios [35, 57, 107, 72]. Based on the observation that solvers have complementary strengths and therefore exhibit incomparable behavior on different problem instances, the ideas of running multiple solvers in parallel or to select one solver based on the features of a given instance were introduced. Appropriately, these approaches have been named *algorithm portfolios*. Portfolio research has led to a wealth of different approaches and an amazing boost in solver performance in the past decade.

One of the biggest success stories is that of SATzilla, which combines existing Boolean Satisfiability (SAT) solvers and has now dominated various categories the SAT Competition for about half a decade [2]. Another example is CP-Hydra [72], a portfolio of CP solvers which won the CSP 2008 Competition. Instead of choosing a single solver for an instance, Smith-Miles [85] proposed a Dirichlet Compound Multinomial distribution to create a schedule of solvers to be run in sequence. Approaches like [45] dynamically switched between a portfolio of solvers based on the predicted completion time. Alternatively, ArgoSmart [64] and Hydra [106] focus on not only choosing the best solver for an instance, but also the best parametrization of that solver. For a further overview of the state-of-the-art in portfolio generation, see the thorough survey in [87].

Interestingly, SATzilla has always entered the three different categories of the SAT competition (Random, Crafted, and Industrial/Application) with three different variants, trained on three different sets of training instances. Ultimately, here we develop algorithm portfolios that are able to deal effectively with a vast range of input instances from a variety of sources. For example, in the context of SAT, one would ideally want to design a *single* portfolio that manages to determine the most appropriate solver given an instance regardless of the instance "type" (such as the aforementioned random, crafted, or industrial).

As discussed in previous chapters, ISAC is a generalization of instance-oblivious configurators such as ParamILS [49] or GGA [4]. Interestingly, through extensive experimentation, this chapter shows that the ideas behind ISAC can be effectively applied to algorithm selection, resulting in solvers that significantly outperform highly efficient SAT solver selectors.

5.1 Algorithm Configuration for Algorithm Selection

In this section we highlight the drawbacks behind regression-based portfolio algorithms and then show how ISAC adresses these issues.

5.1.1 Regression-Based Solver Selection

As mentioned in Chaper 2, regression-based training techniques assume that the expected performance of a solver can be modeled by some function of the instance features. That is, to facilitate the learning process, existing portfolio solvers like SATzilla[108] introduce a learning bias, an assumption that limits the parameters of the function that needs to be learned, at the cost of being able to express more complex functions between instance

features and runtime.

The problem with such a learning bias is that it forces us to learn functions that can be arbitrarily far from reality. Note that, as a direct consequence of this bias, along *any* line in the feature space no solver can define the minimum runtime in two disconnected intervals! This problem could be alleviated by using non-linear regression to better capture the relations between the features and prediction. This, however, demands the question of which function to fit. Similarly, the more complex the function the increasingly more data is needed to avoid over fitting. Alternatively, it is possible to improve linear regression by adding more dimensions (i.e., features), or by redefining features. For example, using the clause over variable ratio as a feature for linear regression means that we must predict an increasing or decreasing runtime as this ratio grows. We could use the distance from the critical threshold of 4.27 instead, which is more likely to have a monotonic relation with runtime.

SATzilla addresses this problem by starting with 48 core SAT features. Then, using feedforward selection, it incrementally selects the features that are most important for the runtime prediction of a solver. After a base set of features is chosen, a binomial combination of the features is created and subsequently filtered. This process of feature selection and combination is iterated multiple times. SATzilla shows that this works well in practice. However, due to the greedy nature of the employed filtering, it is possible that some features are prematurely discarded in order to fit the linear prediction function – especially since the "right" features may very well be solver-dependent. We conjecture that this may in fact be the reason why, despite its success in SAT, the SATzilla methodology has, to our knowledge, not been applied to any other domains.

Consider the following thought experiment. Assume we have a number of algorithms A_i, and each has a runtime that can be accurately predicted as $\text{time}_{A_i}(F) = e^{\beta_{A_i}^T F}$ for algorithm A_i and an input with features F. Now, we build a portfolio P of these solvers. The portfolio P is, of course, itself a solver, and its runtime distribution is $\text{time}_P(F) = \alpha + \min_i e^{\beta_{A_i}^T F}$, where α is the time needed to decide which solver should be invoked. Now, it is easy to see that in general there is no β_P such that $e^{\beta_P^T F} = \alpha + \min_i e^{\beta_{A_i}^T F}$ – simply because a piecewise linear concave function cannot be approximated well with a linear function. This shows that, in general, we cannot assume that the logarithmic runtime distribution of an arbitrary solver can be expressed accurately as a linear function over input features.

5.1.2 Cluster-Based Solver Selection

On the other hand, consider the idea of ISAC to cluster instances and to handle similar instances with the same solver. In fact, the clusters define Voronoi cells and each instance whose normalized feature vector falls into a cell is solved with the corresponding solver. In the example in Figure 5.1 note that solver A is assigned to multiple disconnected clusters that intersect the same line. Such an assignment of instances to solvers is only possible for regression-based approaches using highly complex non-linear functions. In fact, clustering allows us, at least in principle, to handle *all* continuous runtime distributions of algorithms, as the continuity of runtime will result in the same solver to define the minimum runtime in an entire neighborhood. Moreover, assuming continuity of the runtime distributions, Analysis 101 tells us that, at the borders where one solver begins to outperform another, we may assume that the optimal solver only slightly outperforms the other. Consequently,

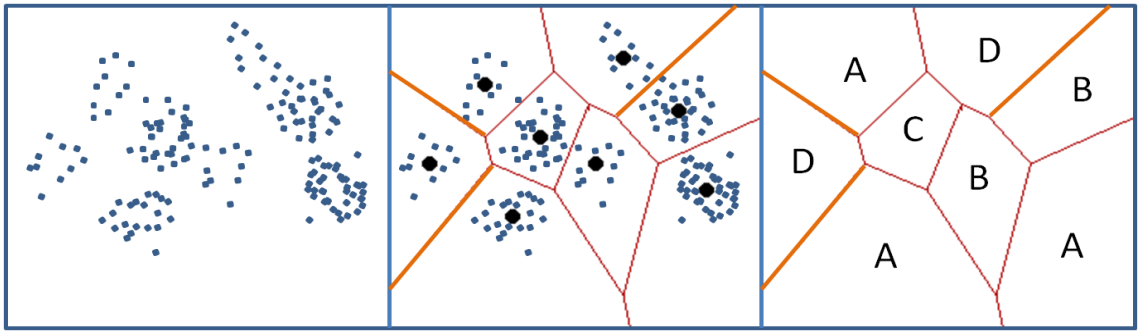

Figure 5.1: Clustering of Instances in a 2-Dimensional Feature Space. Un-clustered instances on the left. The middle cell shows the clustered instances with cluster centers and the corresponding partitioning of the feature space into Voronoi cells. On the right is a hypothetical assignment of algorithms A–D to the clusters.

the ISAC approach is somewhat fault-tolerant.

5.1.3 Using ISAC as Portfolio Generator

With this motivation, this chapter presents how ISAC methodology can be used to generate a portfolio of SAT solvers. Three ways, of differing complexity, are considered. Assume we are given a set of (potentially parameterized) algorithms $A_1, \ldots, A_n$, a set of training inputs T, the set of associated feature vectors F, and a function $Features$ that returns a feature vector for any given input x.

- **Pure Solver Portfolio:** Cluster the training instances according to their normalized features. For each cluster, determine the overall best algorithm. At runtime, determine the closest cluster and tackle the input with the corresponding solver.
- **Optimized Solver Portfolio:** Proceed as before. For each cluster, instance-obliviously tune the preferred algorithm for the instances within that cluster.
- **Instance-specific Meta-Solver Configuration:** Define a parameterized meta-algorithm where the first parameter determines which solver is invoked, and the remaining parameters determine the parameters for the underlying solvers. Use ISAC to tune this solver.

The difference between the pure solver portfolio and the other two approaches is that the first is limited to using the solvers with their default parameters. This means that the performance that can be achieved maximally is limited by that of the "virtually best solver" (a term used in the SAT competition) which gives the runtime of the best solver (with default parameters) for each respective instance. The difference between the optimized solver portfolio and the instance-specific meta-solver configuration is that the latter may find that a solver that is specifically tuned for a particular cluster of instances may work better overall than the best default solver for that cluster, even if the latter is tuned. Therefore, note that the potential for performance gains strictly increases from stage-

to-stage. For the optimized solver portfolio as well as the instance-specific meta-solver configuration it is possible to outperform the virtually best solver.

The remainder of this chapter presents numerical results comparing these three approaches to the state-of-the art portfolio generators for SAT.

5.2 Algorithm Configuration vs. Algorithm Selection of SAT Solvers

This section begins the experimental study by comparing ISAC with the SATzilla_R portfolio. To this end portfolios are generated based on the following solvers: Ag2wsat0 [102], Ag2wsat+ [103], gnovelty+ [76], Kcnfs04 [26], March_dl04 [40], Picosat 8.46 [10], and SATenstein [55]. Note that these solvers are *identical* to the ones that the SATzilla09_R [105] solver was based on when it was entered in the 2009 SAT solver competition.[1] To make the comparisons as fair as possible, these experiments use the same feature computation program made public by the developers of SATzilla to get the 48 core features to characterize a SAT input instance (see [108] for a detailed list of features).

The training set is comprised of the random instances from the 2002-2007 SAT Competitions, where instances that are solved in under a second by all the solvers in our portfolio are removed. Also removed were instances that can not be solved with any of the solvers within a time limit of 1,200 seconds (this is the same time-out as used in phase 1 of the SAT Competition). This left 1,582 training instances. The test set consisted of the 570 instances from the 2009 SAT Solver Competition [2] where SATzilla_R won gold. SATzilla_R is the version of SATzilla tuned specifically for random SAT instances. The random instances were chosen because they belonged to the category where SATzilla showed the most marked improvements over the competing approaches. The cluster-based approaches were trained on dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors and 24 GB of DDR-3 memory (1333 MHz).

Like SATzilla, ISAC utilized the PAR10 score, a penalized average of the runtimes: For each instance that is solved within 1,200 seconds, the actual runtime in seconds defines the penalty for that instance. For each instance that is not solved within the time limit, the penalty is set to 12,000, which is 10 times the original timeout. Note that for the pure solver portfolio, we require much less time than SATzilla to generate the portfolio. Clustering takes negligible time compared to running the instances on the various solvers. However, when determining the best solver for each cluster, we can race them against each other, which means that the total CPU time for each cluster is the number of solvers multiplied by the time taken by the fastest solver (as opposed to the total time of running all solvers on all instances).

For the optimized solver portfolio generator and the meta-solver configurator, the instance-oblivious configurator GGA is employed on each cluster of training instances. In particular, the parameters used for GGA are the following (please refer to [4] for details): The standard population size was set to 100 genomes, split evenly between the competitive and noncompetitive groups. Initial tournaments consider five randomly cho-

[1] Note that the benchmark for a portfolio generator consists in both the training and test sets of problem instances as well as the solvers used to build the portfolio!

Solver	gnovelty+	SATzilla_R	PSP	Cluster	MSC	VBS	MSC+pre
	Training Data Set						
PAR10	4828	685	1234	1234	505	58.6	456
σ	5846	2584	3504	3504	2189	156	2051
Ave	520	153	203	203	129	58.6	128
σ	574	301	501	501	267	156	261
Solved	951	1504	1431	1431	1527	1582	1534
%	60.1	95.1	90.5	90.5	96.5	100	97.0
	Testing Data Set						
PAR10	5874	3578	3569	3482	3258	2482	2480
σ	5963	5384	5322	5307	5187	4742	4717
Ave	626	452	500	470	452	341	357
σ	578	522	501	508	496	474	465
Solved	293	405	408	411	422	457	458
%	51.4	71.1	71.6	72.1	74.0	80.2	80.4

Table 5.1: Comparison of SATzilla, the pure solver portfolio (PSP), the instance-specific meta-solver configuration (MSC), and the virtually best solver (VBS). Also shown is the best possible performance that can be achieved if the same solver must be used for all instances in the same cluster (Cluster). The last columns show the performance of the meta-solver configuration with a pre-solver (MSC+pre). For the penalized and regular average of the time, σ, the standard deviation, is also presented.

sen training instances. The size of this random subset grows linearly with each iteration until the entire training set is included by iteration 75. GGA then proceeded tuning until the 100th generation or until no further improvement in performance is observed. These default parameters of GGA were used because of their performance in prior research.

For the meta-solver, parameters needed to be trained for 16 clusters. These clusters were built to have at least 50 instances each, which resulted in the average cluster having 99 instances and the largest cluster having 253 instances. In total, building the MSC required 260 CPU days of computation time. However, since each of the clusters could be tuned independently in parallel, only 14 days of tuning were required.

5.2.1 Pure Solver Portfolio vs. SATzilla

Table 5.1 reports the results of the experiments. As a pure solver, gnovelty+ performs best, solving 51% of all test instances in time, whereby a number of the other solvers exhibits similar performance. Even though no individual solver can do better, when they join forces in a portfolio performance can be significantly boosted, as the pioneers of this research thread have pointed out [35, 58]. SATzilla_R, for example, solves 71% of the test instances within the given captime of 1,200 seconds. Seeing that the virtually best solver (VBS) sets a hard limit of 80% of instances that can be solved in time by these solvers, this performance improvement is very significant: more than two-thirds of the gap between the best pure solver and the VBS is closed by the SATzilla portfolio. Here, VBS assumes an oracle-based portfolio approach that always chooses the fastest solver for each instance.

The table also shows the performance of the various portfolios based on ISAC. Observe that on the data SATzilla_R was trained on, it performs very well, but on the actual test data even the simple pure solver portfolio generated by ISAC manages to outperform SATzilla_R. On the test set, the pure solver portfolio has a slightly better PAR10 score (the measure that SATzilla was trained for), and it solves a few more instances (408 compared to 405) within the given time limit. That means, in terms of the average runtime, while SATzilla closes about two-thirds of the gap between the best individual solver and the VBS, the simple PSP already closes about 60% of the remaining gap between SATzilla and the VBS.

It is important to note here that in the 2011 SAT Competition, the difference between the winning solver in the industrial instance category and the tenth placed solver was 24 instances. This tenth place solver was also the winner of the 2009 SAT Competition. The improvements observed here with the PSP approach are significant.

In Table 5.1, given under 'Cluster' is the best PAR10 score, average runtime, and number of solved instances *when a single solver is committed to each cluster*. It can be observed that the clustering itself already incurs some cost in performance when compared to the VBS.

5.2.2 Meta-Solver Configuration vs. SATzilla

When considering the optimized solver portfolio, where the best solver is tuned for each cluster, none of the best solvers chosen for each cluster had parameters. Therefore, the performance of the OSP is identical to that of the PSP. The situation changes when the meta-solver configuration approach is used.

As Table 5.1 shows, the MSC provides a significant additional boost in test performance. This portfolio manages to solve 74% of all instances within the time limit, 17 instances more than SATzilla. This improvement over the VBS is due to the introduction of two new configurations of SATenstein that MSC tuned and assigned to two clusters.

In SATzilla the portfolio is not actually a pure algorithm selector. In the first minute, SATzilla employs both the mxc-sr08 [20] SAT solver and a specific parameterization of SATenstein. That is, SATzilla runs a schedule of three different solvers for each instance. In Table 5.1, MSC+pre is a version of an ISAC-tuned portfolio that uses the first minute of allotted time to run these same two solvers. The resulting portfolio outperformed the VBS. This was possible because the MSC added new parameterizations of SATenstein, and also because mxc-sr08 is not one of our pure solvers. As a result, the new portfolio solved 80% of all competition instances, 9% more than SATzilla. At the same time, runtime variance was also greatly reduced: Not only did the portfolio run more efficiently, it also worked more robustly. Seeing that ISAC was originally not developed with the intent to craft solver portfolios, this performance improvement over a portfolio approach that had dominated SAT competitions for half a decade was significant. Based on these results, the 3S Solver Portfolio was entered in the 2011 SAT Competition. 3S is just one portfolio (no sub-versions _R or _I) for all different categories, which comprises 36 different SAT solvers. 3S was the first sequential portfolio that won gold in more than one main category (SAT+UNSAT instances).

Although not explicitly shown, all the results are significant as per the Wilcoxon signed

rank test with continuity correction. MSC is faster than SATzilla_R with $p \leq 0.1\%$.

5.2.3 Improved Algorithm Selection

When compared with the PSP solution, it was noted that the MSC replaced some default solvers for some clusters with other default solvers and, while lowering the training performance, this resulted in an improved test performance. To explain this effect it is important to understand how GGA tunes this meta-solver. As discussed in previous chapters, GGA is a genetic algorithm with a specific mating scheme. Namely, some individuals need to compete against each other to gain the right of mating. This competition is executed by racing several individual parameter settings against one another *on a random subset of training instances*. That means that GGA will likely favor that solver for a cluster that has the greatest chance of winning the tournament on a random subset of instances.

Note that the latter is different from choosing the solver that achieves the best score on the entire cluster as was done for the pure solver portfolio (PSP). What is observed here is that the PSP over-fits the training data. GGA implicitly performs a type of bagging [21] which results in solver assignments that generalize better.

Motivated by this insight, two more methods were tested for the generation of a pure solver portfolio. The two alternative methods for generating cluster-based portfolios are:

- **Most Preferred Instances Portfolio (PSP-Pref):** Here, for each each cluster, the fastest solving algorithm is determined for each instance in that cluster . The cluster is then associated with the solver that most instances prefer.
- **Bagged Portfolio: (PSP-Bag)** For each cluster, the following is done: A random subset of our training instances in that cluster is chosen and determined and the fastest (in terms of PAR10 score) solver (note that each solver is only run once for each instance). This solver is the winner of this "tournament." The process is repeated 100 times and the solver that wins the most tournaments is associated with this cluster.

In Table 5.2 these three cluster-based algorithm selectors are compared with SATzilla_R (whereby these portfolios are again augmented by running SATenstein and mxc-sr08 for the first minute, which is indicate by adding '+pre' to the portfolio name). Observe that PSP-Pref+pre is clearly not resulting in good performance. This is likely because it is important to note not only which solver is best, but also how much better is it than its contenders. PSP+pre works much better, but it does not generalize as well on the test set as PSP-Bag+pre. Therefore, when the base solvers of a portfolio have no parameters, it is recommend to use the PSP-Bag approach to develop a high-performance algorithm selector.

5.2.4 Latent-Class Model-Based Algorithm Selection

In [85] an alternative model-based portfolio approach was presented. The paper addressed the problem of computing the prediction of a solver's performance on a given instance using natural generative models of solver behavior. Specifically, the authors use a Dirichlet Compound Multinomial (DCM) distribution to create a schedule of solvers that is, instead

Solver	SATzilla	PSP+pre	PSP-Pref+pre	PSP-Bag+pre
	Training Data Set			
PAR10	685	476	2324	531
σ	2584	2070	4666	2226
Ave	153	141	289	142
σ	301	489	465	280
Solved	1,504	1,533	1,284	1,525
%	95.1	97.0	81.2	96.4
	Testing Data Set			
PAR10	3578	2955	5032	2827
σ	5384	5024	5865	4946
Ave	452	416	560	402
σ	522	489	562	484
Solved	405	436	334	442
%	71.1	76.5	58.6	77.5

Table 5.2: Comparison of alternate strategies for selecting a solver for each cluster.

of choosing just one solver, they give each solver a reduced time limit and run this schedule until the instance is solved or the time limit is reached. For their experiments, the authors used the 570 instances from the 2009 SAT Competition in the Random category, along with the 40 additional random instances from the same competition originally used for a tie breaking round. This data set of 610 instances was then used to train a latent-class model using random sub-sampling.

In [85] the authors found that this portfolio leads to a slight improvement over SATzilla_R. However, the authors of DCM also mentioned that the comparison is not fully adequate because the latent-class model scheduler uses newer solvers than SATzilla and also the 610 instances were used for both training and testing.

For our experiments we used the same data used in the original research of DCM[2] These times were run on Intel quad core Xeon X5355 (2.66 GHz) with 32GB of RAM. As competitors, our algorithm selection portfolios were trained based on the previously mentioned 1,582 instances from the Random category of the 2002-2007 SAT Competitions.

Table 5.3 shows the performance of SATzilla_R, DCM, and our PSP and PSP-Bag (without the '-pre' option!) using a 5,000 second timeout. To account for the random nature of the underlying solvers the evaluation of the DCM schedule and our portfolios was repeated ten times. The table shows mean and median statistics. Even though, as mentioned earlier, the comparison with SATzilla_R is problematic, it is included here to make sure that our comparison is consistent with the finding in [85], DCM works slightly better than SATzilla. The results in the table confirm this. However, the PSP and PSP-Bag portfolios can do much better and boost the performance from 76% of all instances solved by the DCM to 87% solved by PSP-Bag. Keeping in mind the simplicity of clustering and solver assignment, this improvement in performance was noteworthy.

[2]Our thanks go to Bryan Silverthorn who provided the 610 instances used in the experiments in [85], as well as the runtime of the constituent solvers on his hardware, and also the final schedule of solvers that the latent class model found (see [85] for details).

Solver	SATzilla	DCM	PSP	PSP-Bag
PAR10	12794	12265	7092	7129
σ	182	314	180	293
Ave	1588	1546	1242	1250
σ	16.6	21.7	14.8	19.5
Solved	458	465	531	530
σ	2.38	4.03	2.36	3.83
%	75.1	76.2	87.0	86.9
σ	0.39	0.66	0.39	0.63
Solved (median)	458	464	531	531
% (median)	75.1	76.0	87.0	87.1

Table 5.3: Comparison with the DCM Portfolio developed by Silverthorn and Miikkulainen [85] (results presented here were reproduced by Silverthorn and sent to us in personal communication). The table presents mean run-times and median number of solved instances for 10 independent experiments.

5.3 Comparison with Other Algorithm Configurators

As shown in the previous section, when the employed solvers have parameters, ISAC and the meta-solver configuration approach offer more potential than the pure solver portfolios PSP and PSP-Bag which serve merely as algorithm selectors. In this section, ISAC is compared with two other approaches that train the parameters of their solvers, ArgoSmart [64] and Hydra [106].

5.3.1 ISAC vs. ArgoSmart

An alternate version of the idea that parameterized solvers can be used in a portfolio is also in considered in ArgoSmart [64]. Using a supervised clustering approach, the authors build groups of instances based on the directory structure in which the SAT Competition placed these instances. The authors enumerate all possible parameterizations of ArgoSAT (60 in total) and find the best parameterization for each family. For a test instance, ArgoSmart then computes the 33 of 48 core SATzilla features that do not involve runtime measurements [108] and then assigns the instance to one of the instance families based on majority k-nearest-neighbor classification based on a non-Euclidean distance metric. The best parameterization for that family is then used to tackle the given instance.

ISAC is more widely applicable, as it clusters instances in an unsupervised fashion. Moreover, ISAC employs GGA to find the solver parameters instead of enumerating all possible configurations. Therefore, if the parameter space were much bigger, the ArgoSmart approach would need to be augmented with an instance-oblivious parameter tuner to find parameters for each of the instances families that it inferred from the directory structure. Despite these current limitations of the ArgoSmart methodology, we compared our assignment of test instances to clusters based on the Euclidean distance to the nearest cluster center with more elaborate machine learning techniques.

To make this assessment, a PSP and a PSP-Bag were generated based on the time

Solver	ArgoSat	ArgoSmart	Unsupervised Clustering			Supervised Clustering			VBS
			PSP	PSP-Bag	Cluster	PSP	PSP-Bag	Cluster	
	Training Data Set								
PAR10	2704	-	2515	2527	2515	2464	2473	2464	2343
σ	2961	-	2935	2967	2935	2927	2959	2927	2906
Ave	294	-	276	276	276	270	271	270	255
σ	285	-	283	284	283	283	283	283	283
Solved	736	-	778	775	778	789	787	789	815
%	55.4	-	58.5	58.3	58.5	59.4	59.2	59.4	61.3
	Testing Data Set								
PAR10	2840	2650	2714	2705	2650	2650	2650	2628	2506
σ	2975	2959	2968	2967	2959	2959	2959	2959	2941
Ave	306	286	291	290	286	286	286	281	269
σ	286	286	287	287	286	286	286	287	286
Solved	337	357	350	351	357	357	357	359	372
%	53.1	56.2	55.1	55.3	56.2	56.2	56.2	56.5	58.6

Table 5.4: Comparison with ArgoSmart [64] (results presented here were reproduced by Nikolic and sent to us in personal communication).

of each of ArgoSAT's parameterizations on each instance[3]. These times were computed on Intel Xeon processors at 2 GHz with 2GB RAM. In Table 5.4 ArgoSmart is compared with two versions of PSP and PSP-Bag, respectively. Both use the same 33 features of ArgoSmart to classify a given test instance. In one version, unsupervised clustering of the training instances is used. The other version uses the supervised clustering gained from the directory structure of the training instances which ArgoSmart used as part of its input. For both variants the best possible cluster-based performance is given. Observe that the supervised clustering offers more potential. Moreover, when PSP-Bag has access to this clustering, despite its simple classification approach, it performs as well as the machine learning approach from [64]. However, even when no supervised clustering is available as part of the input, ISAC can still tune ArgoSAT effectively.

Note that the times of ArgoSmart are different from those reported in [64] because the authors only had the times for all parameterizations for the 2002 SAT data and not the 2007 SAT data they originally used for evaluation. The authors generously retuned their solver for a new partitioning of the 2002 dataset, to give the presented results.

5.3.2 ISAC vs. Hydra

The methodology behind our final competitor, Hydra [106], enjoys equal generality as ISAC. Hydra consists of a portfolio of various configurations of the highly parameterized local search SAT solver SATenstein. In Hydra, a SATzilla-like approach is used to determine whether a new configuration of SATenstein has the potential of improving a portfolio of parameterizations of SATenstein, and a ParamILS-inspired procedure is used to iteratively propose new instantiations of SATenstein. In other words Hydra creates and adds

[3]information that was generously provided by Mladen Nikolic

Solver	saps	stein (FACT)	Hydra	MSC-stein	PSP-Bag 11	PSP-Bag 17	MSC-12
	Training Data Set						
PAR10	102	26.8	-	1.78	18.03	1.41	1.41
σ	197	109	-	13.6	87.9	4.09	4.16
Ave	13.5	4.25	-	1.48	3.63	1.11	1.41
σ	19.6	11.3	-	4.41	10.4	3.05	4.16
Solved	1206	1425	-	1499	1452	1499	1500
%	80.4	95.0	-	99.9	96.8	99.9	100
	Testing Data Set						
PAR10	861	220	1.43	1.27	73.5	1.21	1.21
σ	2086	1118	5.27	3.73	635	4.42	3.27
Ave	97.8	26.0	1.43	1.27	12.3	1.20	1.21
σ	210	114	5.27	3.73	69.0	4.42	3.27
Solved	1288	1446	1500	1500	1483	1500	1500
%	85.9	96.4	100	100	98.9	100	100

Table 5.5: Comparison of Local-Search SAT Solvers and Portfolios Thereof on BM Data.

Solver	saps	stein (CMBC)	Hydra	MSC-stein	PSP-Bag 11	PSP-Bag 17	MSC-12
	Training Data Set						
PAR10	54.6	6.40	-	2.99	51.7	3.97	3.00
σ	147	23.5	-	3.94	143	22.6	4.47
Ave	10.5	5.50	-	2.99	10.3	3.07	3.00
σ	15.5	8.07	-	3.94	15.3	6.54	4.47
Solved	451	499	-	500	454	499	500
%	90.2	99.8	-	100	90.8	99.8	100
	Testing Data Set						
PAR10	208	5.35	5.11	2.97	209	3.34	2.84
σ	1055	8.54	9.41	4.08	1055	7.05	4.07
Ave	35.7	5.35	5.11	2.97	36.4	3.34	2.84
σ	116	8.54	9.41	4.08	116	7.05	4.07
Solved	484	500	500	500	484	500	500
%	96.8	100	100	100	96.8	100	100

Table 5.6: Comparison of Local-Search SAT Solvers and Portfolios Thereof on INDU Data.

solvers to its portfolio one at a time, even removing those same solvers when they are deemed to no longer help the overall performance of the portfolio.

To cover the breadth of possibilities, three different approaches are considered for building a portfolio of local search SAT solvers and are compared with Hydra[4] in Tables 5.5 and 5.6. The respective benchmarks BM and INDU were introduced in [106]. Both in-

[4]We are grateful to Lin Xu who provided the Hydra-tuned SATensteins as well as the mapping of test instances to solvers.

stance sets appear particularly hard for algorithm configuration: In [106], Hydra was not able to outperform an algorithm selection portfolio with 17 constituent solvers. The BM and INDU benchmarks consist of 1,500 train and 1,500 test instances, and 500 train and 500 test instances, respectively. The INDU dataset is comprised of only satisfiable industrial instances, while BM is composed of a mix of satisfiable crafted and industrial instances. These experiments used dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors and 24 GB of DDR-3 memory (1333 GHz) to compute the runtimes.

The training of the portfolios was conducted using a 50 second timeout, for testing a 600 second timeout was used. It is important to point out that, despite using a tenfold longer training timeout than [106], the total training time for each portfolio was about 72 CPU days, which is comparable with the 70 CPU days reported in [106] (note also that significantly slower machines for tuning were used). The reason is that GGA was used instead of ParamILS to train the solvers on each cluster. GGA is population-based and races parameter sets against each other, which means that runs can be interrupted prematurely when a better parameter set has already won the race. It is an inherent strength of ISAC that it can handle longer timeouts than Hydra. Compared in the presented results are the two approaches assuming they are given the same number of CPU days during which to tune.

The portfolio closest to Hydra is denoted MSC-stein. Here, like Hydra, only SATenstein is tuned. As usual, this approach clusters our training instances, and for each cluster SATenstein is tuned using GGA. For evaluation, like for the original Hydra experiments, each solver is run three times and the median time is presented. Observe again that the clustering approach to portfolio generation offers advantages. While Hydra uses a SATzilla-type algorithm selector to decide which tuned version of SATenstein an instance should be assigned to, ISAC employs clusters for this task. As a result, ISAC has a 12% reduction in runtime over Hydra on the BM data-set and more than 40% reduction on INDU. There is also a significant reduction in runtime variance over Hydra: Again, not only does the new portfolio work faster; it also works more robustly across various instances.

Next the ISAC methodology is used further to build portfolios with more constituent solvers. Following the same setting as in [106], an algorithm selector was built using 11 local search solvers (PSP-Bag 11): paws [95], rsaps [50], saps [96], agwsat0 [102], agwsat+ [103], agwsatp [101], gnovelty+ [76], g2wsat [60], ranov [75], vw [78], and anov09 [43]. In this setting, saps' performance is the best. The number of constituent solvers is further augmented through the addition of six fixed parameterizations of SATenstein, resulting in a total of 17 constituent solvers. The respective portfolio is denoted PSP-Bag 17. Finally, MSC-12 is built based on the (un-parameterized) 11 original solvers plus the (highly parameterized) SATenstein.

Consistent with [106] observe the following:

- Apart from the INDU data-set, where the portfolio of 11 solvers cannot improve the performance of the best constituent solver, the portfolios boost significantly the performance compared to the best constituent solver (saps for the 11 solvers on both benchmarks and, for the 17 solvers, SATenstein-FACT on the BM data-set and SATenstein-CMBC on the INDU data-set).
- The portfolio of 17 solvers dramatically improves performance over the portfolio of 11 solvers. Obviously the variants of SATenstein work very well and, on the INDU

benchmark, also provide some much needed variance so that the portfolio is now able to outperform the best solver.

In [106] it was found that Hydra, based on only the SATenstein solver, can match the performance of the portfolio of 17 solvers on both benchmarks. While this may be true when the portfolios are built using the SATzilla methodology, this is not true when using our algorithm selector PSP-Bag 17. On BM, PSP-Bag 17 works more than 15% faster than Hydra and on the INDU benchmark set it runs more than 33% faster.

The full potential of the ISAC approach is of course only realized when a portfolio is built using parameterized and un-parameterized solvers. The result is MSC-12 which clearly outperforms all others, working on average almost 18% faster than Hydra on BM and more than 45% faster than Hydra on INDU.

5.4 Chapter Summary

This chapter presented the idea of using instance-specific algorithm configuration (ISAC) for the construction of SAT solver portfolios. The approach works by clustering training instances according to normalized feature vectors. Then, for each cluster, it determines the best solver or computes a high performance parameterization for a solver. At runtime, the nearest cluster is identified for each instance and the corresponding solver/parameterization is invoked. In all experiments, to compare competing approaches, every precaution was taken to make sure that the conditions under which they were developed were as close as possible. This included using the same solvers in the portfolio, the same tuning times, and same training and testing sets.

The chapter showed that this very simple approach results in portfolios that clearly outperform the SAT portfolio generator SATzilla [108], a recent SAT solver scheduler based on a latent-class model, and the algorithm configuration method Hydra [106]. At the same time, ISAC is widely applicable and works completely unsupervised.

This study shows that instance-specific algorithm tuning by means of clustering instances and tuning parameters for the individual clusters is highly efficient even as an algorithm portfolio generator. The fact that, when tuning instance-specifically, ISAC considers portfolios of a potentially infinite number of solvers does not mean that it is necessary to revert to sub-standard portfolio selection. On the contrary: Unsupervised clustering, which originally was a mere concession to tuning portfolios with extremely large numbers of solvers, has resulted in a new state-of-the-art in portfolio generation.

Chapter Six

Feature Filtering

The previous chapters have shown the effectiveness of the ISAC on a number of problem types. Furthermore, the configurability of the methodology was shown through interchanging the techniques used for training and clustering. One thing that has so far stayed constant is the selection of features employed to identify the instances. This is arguably also the most crucial part of the success of ISAC. If the features do not properly represent the structures that mark similarities or differences of instances, then clustering and training is done on essentially random subsets of the data. It is important to note, however, that even in this case, ISAC does not hurt performance if we assume that solvers tuned on large enough subsets of the entire training set are likely to behave in a similar manner as a single solver trained on the entire training set. Yet the more accurately the features represent the instances, the better the clusters and thus the more accurate the training. This chapter therefore focuses on how to filter the large feature sets that were used until now.

To tackle this problem, the chapter introduces an approach that builds from our initial assumption that instances with similar features will behave comparably under the same parameter settings. Three new evaluation functions are designed that can be computed without retuning solvers for each iteration. These functions are first evaluated on two standard SAT benchmarks, and then confirmed in the CP domain.

6.1 Cluster Evaluation

The effect of the filtering algorithms, such as the ones discussed below in Section 6.2, strongly depends on the quality of the evaluation function. In order to evaluate a set of features using standard techniques, the training instances would be clustered and a new solver tuned for each cluster. The quality of the feature would then be defined as the performance of the portfolio solver on some validation set of instances. However, because of the long time needed to tune the algorithms, evaluation based on this kind of performance is impractical. This issue can by sidestepped by instead focusing on the primary assumption behind ISAC: that a solver will have consistent performance on instances that are clustered together. Based on this assumption, we introduce three possible evaluation functions that utilize a collection of untuned solvers to determine the quality of a cluster of instances.

The first evaluation criterion is presented in Algorithm 5 as E_Dist. Given the clustering of the instances C, the runtime of each untuned solver on each instance R, and the list of instances I, this algorithm tries to match the relative quality of solver runtimes on instances in the same cluster. In other words, the algorithm tries to make sure that the same solver works best on all instances in the cluster, and that the same solver provides the worst performance. Because the runtimes can vary significantly between instances, these times are normalized for each instance to range from 0 to 1, with 0 being the fastest runtime and 1 the slowest. These normalized runtimes N can then be used to judge how similar two instances are, and a *good* cluster is one where the average euclidean distances between the instances within the cluster is minimized. The evaluation of the overall clustering v^* is then the summation of the quality of each cluster weighted by the number of instances in that cluster, $|c|$. Here we do not consider the distances between clusters because it is not necessarily the case that different clusters require different solvers. Only the uniformity of the instances within a cluster determine the success of a clustering.

An alternative evaluation function measures the quality of the clustering directly by

Algorithm 5: Evaluation functions used to measure the quality of a clustering of instances.

```
1: E_Dist(C, R, I)
2: N ← ∅, v* ← 0
3: for i ∈ I do
4:    N_i ←Normalize(R_i)
5: end for
6: for c ∈ C do
7:    v* ← v* + |c| * Σ_{i∈I} Σ_{j>i} ||N_i − N_j||
8: end for
9: return v*
```

Algorithm 6: Evaluation functions used to measure the quality of a clustering of instances.

```
1: E_Time(C, R)
2: v* ← 0
3: for c ∈ C do
4:    v* ← v* + min_{s∈R}(Runtime(s, c))
5: end for
6: return v*
```

computing the performance of a portfolio algorithm based on the available solvers. E_Time in Algorithm 6 creates a portfolio of untuned solvers and chooses which solver to assign to each cluster. The algorithm finds the best performing solver in R on the instances of each cluster. The clustering can then be evaluated by summing the score for each cluster when using the best solver. This evaluation approach benefits from being similar to how ISAC will be evaluated in practice, without having to tune each solver for the cluster.

For Algorithm 6 we use the exact runtimes of the best solvers to evaluate a clustering. We also experimented with an algorithm where we again select the best solver for each cluster, but the evaluation is done using a penalized version of the runtimes, called Par10. Each instance not solved gets penalized with a runtime that is ten times the cutoff time. Using penalized runtimes makes the algorithms focus on minimizing the number of instances not solved. However, we found that using the regular non-penalized runtimes provided better performance, both in terms of the average runtimes achieved and number of instances solved.

Using the performance of a portfolio for evaluating the clustering can yield very good results if the solvers in the available portfolio are numerous and have a lot of variance in their performance. This is true in the case of a well studied problem like SAT, but is not necessarily the case in all problem domains. To circumvent this issue, we extend the evaluation criteria to generate the desired portfolio.

The third evaluation criteria exploits the idea that given a single, highly parameterized solver, it is possible to tune this solver using GGA. In this case, however, the best performing parameter set is not needed, but instead many parameter sets that behave reasonably well and with a lot of variance. These parameter sets can be initialized randomly, but the resulting solvers are likely to perform very poorly. In a case were every solver times

Algorithm 7: Feedforward feature selection

```
1: FeedForwardSelection(F, I, R)
2: F* ← ∅ , F̂ ← F , s ← ∞ , s* ← ∞
3: while s ≤ s* do
4:    f* ← ∅ , s ← ∞
5:    for f ∈ F̂ do
6:       v = EVALUATE(CLUSTER(F* ∪ f, I), R)
7:       if v ≤ s then
8:          f* ← f
9:          s ← v
10:      end if
11:   end for
12:   if s ≤ s* then
13:      F* ← F* ∪ f*
14:      s* ← s
15:   end if
16: end while
17: return F*
```

out, it is impossible to determine which solver is best. But, if we use the solvers from an intermediate generation of GGA, we will find that the poor parameter sets have already been killed off by the tuner, and all that remains are parameters that work well on different subsets of our training data. Using these parameter settings, we create a large portfolio of solvers that we can use for the direct evaluation of a clustering. This evaluation approach works as Algorithm 6, using the best solver on each cluster to compute the performance score of a clustering, the difference being that the runtimes of the generated solvers are used in place of the regular solvers.

6.2 Filtering Algorithms

It is well established that the success of a machine learning algorithm depends on the quality of its features. Too few features might not be enough to differentiate between two or more classes of instances properly. Alternatively, too many features often results in some of the features being noisy and even damaging. For example, imagine a feature set of 1,000 values where only the first 10 are needed make the perfect clustering. In such a scenario it is statistically very likely that if all the other features are just random noise, there is some subset that would provide a seemingly good clustering on the training data. Furthermore, as the feature space increases, more data is needed in order to make accurate predictions. This dissertation works with the three standard feature selection algorithms: feedforward selection, backward selection, and a hybrid of the two. All three of these algorithms can use any of the evaluation functions discussed in Section 6.1.

Feedforward selection (Algorithm 7) starts with an empty set F^* and tries to add each of the available features $\hat{F}$. Using each of these new subsets of features, the training set I is clustered and evaluated. The feature f^* whose addition to the current set yields the best performance, according to the evaluation function, s is added to the set and the process is

repeated until no more features can be added without the evaluation score deteriorating.

Alternatively, backward selection starts with the full feature set and removes features one at a time in a manner that is analogous with how feedforward selection adds them. The algorithm terminates when such a removal leads to a decrease in performance according to the evaluation function.

Both feedforwad and backward selection filtering strategies are greedy algorithms and it is possible for them to make suboptimal decisions. A natural extension of the above two algorithms is a hybrid approach of the two. As in backward selection, the algorithm begins with the full feature set, and removes features one at a time while the solution does not deteriorate. The algorithm, however, also checks if adding any of the removed features improves the solution. If this is the case, the feature is added back into the set. This helps when the beneficial effects are being obfuscated by many noisy features. Once the troublesome features are removed, the benefits are observed and the mistake of removing the feature is rectified.

6.3 Numerical Results

For our experiments we first focus on the SAT domain, a well-studied problem that has numerous solvers, benchmarks, and well defined features. SAT is also a domain where ISAC has been shown to yield state-of-the-art performance. Showing the performance gains on SAT, we then continue by switching to the CP domain. The timed experiments used dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors with 24 GB of DDR-3 memory (1333 GHz).

6.3.1 Benchmarks

For SAT, we choose to focus on local search solvers predominately due to the existence of SATenstein [55], a highly parameterized solver. SATenstein is used to explore our evaluation function that uses GGA to create a portfolio of solvers. Evaluation is based on the HAND and RAND datasets [106] that span a variety of problem types and were developed to test local search-based solvers. Because local search can never prove that a solution does not exist, these four datasets only have satisfiable instances. These two data sets were chosen because they have been shown to be hard for portfolio algorithms in [106].

The HAND and RAND datasets are respectively composed of hand-crafted and randomly generated instances. The HAND dataset has 342 training and 171 testing instances. The RAND dataset has 1141 training and 581 testing instances. For features, we use the well established 48 features introduced in [105]. It is important to note that these features have been used for SAT portfolios for over a decade, so they have been thoroughly vetted as being important.

In these experiments we used an assortment of successful local search solvers: paws [95], rsaps [50], saps [96], agwsat0 [102], agwsat+ [103], agwsatp [101], gnovelty+ [76], g2wsat [60], ranov [75], vw [78], and anov09 [43]. We also use six additional fixed parameterizations of SATenstein, known as Fact, Cmbc, R3fix, Hgen, Swgcp, and Qcp. For evaluation, a 600 second timeout was used.

HAND	BS	All	E_Dist			E_Time			E_Time (GGA)		
	rsaps	Features	Forward	Backward	Hybrid	Forward	Backward	Hybrid	Forward	Backward	Hybrid
Par 10 - avg	3034	2789	2784	2823	2823	**2546**	2712	2711	2752	**2748**	**2748**
Par 10 - std	2977	2975	2980	2979	2979	**2945**	2975	2976	2974	**2978**	**2978**
Runtime - avg	296.9	294.8	289.5	296.5	296.5	**273.0**	280.8	280.8	289.6	**285.1**	**285.1**
Runtime - std	142.5	289.5	293.6	290.6	290.6	**288.5**	292	292.6	290.5	**291.6**	**291.6**
features	-	48	10	33	33	**4**	20	23	5	**39**	**39**
clusters	-	4	5	5	5	**5**	5	5	5	**5**	**5**
solved	93	92	92	91	91	**99**	94	94	93	**93**	**93**
% solved	54.39	53.8	53.8	53.2	53.2	**57.89**	54.97	54.97	54.39	**54.39**	**54.39**
RAND	BS	All	E_Dist			E_Time			E_Time (GGA)		
	gnovelty+	Features	Forward	Backward	Hybrid	Forward	Backward	Hybrid	Forward	Backward	Hybrid
Par 10 - avg	1138	755.5	780.1	745.2	745.2	**698.5**	729.9	729.9	762.7	**688.9**	709.4
Par 10 - std	2239	1958	1995	1946	1946	**1899**	1936	1936	1971	**1886**	1911
Runtime - avg	126.2	95.61	92.33	94.58	94.58	**85.1**	88.59	88.59	93.48	**84.8**	86.72
Runtime - std	229.9	92.33	204.3	202.0	202.0	**195.4**	199.1	199.1	201.9	**194.8**	197.1
features	-	48	5	37	37	**6**	38	38	5	**30**	33
clusters	-	11	12	14	14	**11**	12	12	11	**11**	11
solved	483	510	507	511	511	**515**	512	512	509	**516**	514
% solved	83.13	87.78	87.26	88	88	**88.64**	88.12	88.12	87.61	**88.81**	88.47

Table 6.1: Results on the SAT benchmarks, comparing the best performing individual solver "BS," the original ISAC using all features "All Features," and all the combinations of evaluation functions and filtering algorithms. For each evaluation function, numbers that have the most improvement over "All Features" are in bold.

For the CP benchmarks we employ instances from CPAI08 [1]. We removed a small number of instances for which the cpHydra feature computation code [72] did not work. The remaining instances were split into 901 training instances and 930 test instances. Our portfolio consisted of a subset of the solvers that competed in the original competition: Abscon_112v4 (AC), Abscon_112v4 (ESAC), bpsolver (2008-06-27), casper (zao), casper (zito), choco2_dwdeg (2008-06-26), choco2_impwdeg (2008-06-26), cpHydra (k_10), cpHydra (k_40), MDG-noprobe (2008-06-27), MDG-probe (2008-06-27), Minion_Tailor (2008-07-04), Mistral-option (1.314), Mistral-prime (1.313), SAT4J_CSP (2008-06-13), Sugar (v1.13+minisat), Sugar (v1.13+picosat). For the runtimes, we used the runtimes from the competition results which had a 1,800 second timeout [1].

6.3.2 E_Dist Approach

Table 6.1 shows the results of the best performing solver over all the instances in each of the four benchmarks. The table then presents the performance of an algorithm portfolio of the 17 local search solvers tuned with ISAC using the complete set of 48 features. In this scenario, once the training instances are clustered, the best performing solver in the portfolio is assigned to each cluster. The decision of the solver to use is based on the best average runtime. In all cases the average runtime was improved. The change is especially significant for RAND. Furthermore, as is expected, in all cases ISAC usually does not use the solver that is found best over all instances, which suggests that certain solvers are better at solving certain instances while sacrificing performance on other types. For the HAND benchmark, however, the time gain is minimal and one fewer instance is solved. Judging by the performance of ISAC with feature filtering, this poor performance is due to a poor clustering as a result of some confounding and noisy features.

Table 6.1 then shows the results from running ISAC after the features are filtered using the euclidean distance evaluation criteria, E_Dist. It is interesting to note that for both of the datasets, it is possible to maintain the same level of performance with significantly fewer features. This is especially true for feedforward selection that uses less than a quarter of the features. However, we also observe that there is no significant improvement of the overall performance of the resulting portfolio.

6.3.3 E_Time Approach

Once we use a more accurate evaluation criteria, E_Time, we observe that ISAC's performance can be boosted, as seen in Table 6.1. Here, feedforward selection is again the best performing approach and we observe improvements in both of our datasets, although we also observe an increase in the number of used features. This seems to support the assumption that not all of the established 48 features are necessary, and are in fact damaging to the clustering in ISAC.

It is interesting to note that the features found by forward selection do not overlap much for four benchmarks with only two features appearing in both sets. The first is the maximum ratio of positive to negative literals per clause. The second is the number of unary literals in the problem.

Also of note is that feedforward selection only chooses the local search probing features twice for RAND. In backward selection, however, almost all these probing features are

CP	cpHydra (k = 40)	All Features	Forward	Backward
Parscore	2667	2421	2124	**1994**
Std. deviation	6695	6083	5756	**5589**
Avg. runtime	286.1	278.8	242.3	**234.5**
Std. deviation	617.0	613.5	579.5	**567.4**
# features	36	36	7	**29**
# clusters	-	18	20	**19**
# solved	807	807	822	**829**
% solved	86.77	86.77	88.39	**89.14**

Table 6.2: Results on the CP benchmark, comparing the best performing solver "cpHydra," ISAC using "All Features," and the Forward and Backward filtering algorithms using the E_Time evaluation function.

used in both benchmarks. These features are stochastic with potentially a lot of variance between computations. They are also very computationally expensive, especially for larger instances. Fortunately, according to a comparison of the forward and backward selection algorithms these features are not needed and do not improve performance.

The number of clusters does not change drastically when switching from the euclidean distance, E_Dist, to the time performance evaluation functions, E_Time. This suggests that simply increasing or decreasing the number of clusters is not enough to improve performance. This also validates our clustering approach, showing that if a cluster contains similar instances, then it is possible to tune a very high quality solver for those instances.

In these experiments, there is little difference in the features found by backward selection and our hybrid approach. This is not very surprising since the clustering is based on a linear relation between the features, and it is unlikely that a removed feature would become beneficial once another feature is removed.

These results on SAT are encouraging, and as can be seen in Table 6.2, they extend to the CP domain. When comparing all the single solvers, cpHydra significantly outperforms the other solvers. The closest competitor is Mistral-prime (1.313), which solves 780 instances. As shown in the table, applying ISAC to tune a portfolio algorithm leads to marginal improvements in the average runtime. However, once feature filtering is applied the performance of the tuned portfolio improves significantly. Using backward filtering creates a performance gap to cpHydra equal to the one separating cpHydra from its closest competitor.

6.3.4 E_Time (GGA) Approach

Running the feature filtering algorithms using the runtimes of the 100 GGA generated solvers yields the times presented under "E_Time (GGA)" in Table 6.1. In this case, the forward selection algorithm worsens performance on RAND, but otherwise remains competitive with the original ISAC with less than 11% of the features. Backward selection does not worsen performance on any datasets, and even gives the best performance of all the approaches on the RAND dataset, while removing a significant number of features in all cases.

In all cases, we see that using feedforward selection greatly reduces the number of needed features while usually improving the overall performance of the resulting tuned solver. Backward selection on the other hand seems like a more conservative approach, removing fewer features, but also offering consistent improvements over all datasets. This suggests that there are some features that should not be used for clustering, and all filtering algorithms remove these. But there are also some dependancies between the features, and including these features is important to improve the quality of the clusters used by ISAC.

While feedforward selection generally outperformed backward selection on all datasets when using the portfolios of solvers, we see that when using the GGA generated solvers, backward selection clearly outperforms forward selection. When using a portfolio of solvers, we have access to more variety in solvers, as opposed to using a set of parameterized versions of the same solver. This seems to indicate that feedforward selection has a higher need for diversity in solvers, as it struggles with picking out the most important features starting from none, whereas backward selection is able to remove a large set of noisy features successfully and provide performance gains using the less diverse GGA solvers.

6.4 Chapter Summary

This chapter showed how the ISAC approach depended on the quality of the features it was provided. It is well known in the machine learning community that a poorly selected set of features can have a strong detrimental effect on performance. Standard approaches for feature filtering were not applicable due the computational cost associated with the evaluation of a clustering using a subset of features. Three modifications were therefore shown to the evaluation function that remove the expensive portion of the approach: feedforward selection, backward selection, and a hybrid of the two. Applying feature filtering to ISAC, performance gains were shown in both SAT and CP domains, while reducing the size of the feature sets.

These performance gains are important in the case of SAT since the 48 features are already a subset of a larger set of 89 features which has been carefully studied for the last ten years. Yet even in this case, we show that proper feature filtering does not worsen the performance but has a chance to improve it greatly. This observation is confirmed in the CP domain where the features have not been as carefully vetted. Just applying ISAC on all 36 features did not improve the number of solved instances. But once feature filtering was applied, the performance improved noticeably.

Chapter Seven

Dynamic Training

ISAC is a powerful tool for training solvers for the instances they will be applied on. One of ISAC's strengths lies in its configurability. Composed of three steps (computing features, clustering, and training), this methodology is not restricted to any single approach for any of these steps. For example, so far we have shown how a local search, GGA, and selecting the single best solver in a portfolio are all possibilities for training a solver for a cluster. In this chapter we show how by changing the clustering portion of the methodology, it is possible to train the portfolio dynamically for each new test instance. Using SAT as the testbed, the chapter then demonstrates through extensive numerical experiments that this new technique is able to handle even highly diverse benchmarks, in particular a mix of random, crafted, and industrial instances, and even when the training set is not fully representative of the test set that needs to be solved.

7.1 Instance-Specific Clustering

The previous chapters used g-means[37] to analyze the training data and find a stable set of clusters. Although this has been shown to work very well in practice, improved performance is possible for the instances that are far from the found cluster centers. For example, in our version of g-means, a minimum cluster size is imposed to ensure that there are enough instances in each cluster to train a solver accurately. This means that instances in clusters that are smaller than this threshold are reassigned to the nearest neighboring cluster. This reassignment can potentially bias the training of the solver. To help avoid this scenario, a possible solution is to create the clusters dynamically using the k-nearest-neighbors approach to create the clusters dynamically for each new test instance.

7.1.1 Nearest-Neighbor-Based Solver Selection

Nearest-neighbor classification (k-NN) is a classic machine learning approach. In essence, the decision for a new example is based on prior experience in the k most similar cases. In our context, this means that we first identify which k training instances are the most "similar" to the one given at runtime, and then choose the solver that worked the "best" on these k training instance. Like before, we use Euclidean distance on 48 normalized[1] core features of SAT instances that SATzilla is based on [108] as the similarity measure, and the Par10-score of a solver on these k instances as the performance measure.

When $k = 1$ it is assumed that each training example is unique, and therefore that there are no errors on the training set as each instance is its own nearest neighbor. However, it is well-known in machine learning that 1-NN often does not generalize well to formerly unseen examples, as it tends to over-fit the training data. A very large value of k also obviously defeats the purpose of considering local neighborhoods. To address the challenge of finding the "right" value of k, another classic strategy in machine learning is employed, namely *random sub-sampling cross validation*. The idea is to utilize only a subset of the training data and to assess how well a learning technique performs when trained on this subset and evaluated on the remaining training instances. A split ratio of 67/33 is used to partition the training data and perform random sub-sampling 100 times to obtain a fairly good understanding of how well the technique generalizes to instances on which it was not

[1] We associate each feature with a linear normalization function ensuring that the feature's minimum and maximum values across the set of training instances are 0 and 1, respectively.

Algorithm 8: Algorithm Selection using Nearest-Neighbor Classification

k-NN-Algorithm-Selection Phase
Input : a problem instance F
Params: nearest neighborhood size k, candidate solvers $\mathcal{S}$, training instances $\mathcal{F}_{\text{train}}$ along with feature vectors and solver runtimes
Output: sat or unsat
begin
  compute normalized features of F
  $\mathcal{F} \leftarrow$ set of k instances from $\mathcal{F}_{\text{train}}$ that are closest to F
  $S \leftarrow$ solver in $\mathcal{S}$ with the best PAR10 score on $\mathcal{F}$
  return $S(F)$
end
Training Phase
Input : candidate solvers $\mathcal{S}$, training instances $\mathcal{F}_{\text{train}}$, time limit $T_{\max}$
Params: nearest neighborhood range $[k_{\min}, k_{\max}]$, perform random sub-sampling m times and split ratio m_b/m_v (default 70/30)
Output: best performing k, reduced $\mathcal{F}_{\text{train}}$ along with feature and runtimes
begin
  run each solver $S \in \mathcal{S}$ for time $T_{\max}$ on each $F \in \mathcal{F}_{\text{train}}$; record runtimes
  remove from $\mathcal{F}_{\text{train}}$ instances solved by no solver, or by all within 1 second
  compute feature vectors for each $F \in \mathcal{F}_{\text{train}}$ **for** $k \in [k_{min}, k_{max}]$ **do**
    score$[k] \leftarrow 0$
    for $i \in [1..m]$ **do**
      $(\mathcal{F}_{\text{base}}, \mathcal{F}_{\text{validation}}) \leftarrow$ a random m_b/m_v split of $\mathcal{F}$
      score$[k] \leftarrow$ score$[k]+$ performance of k-NN portfolio on $\mathcal{F}_{\text{validation}}$ using training instances $\mathcal{F}_{\text{base}}$ and solver selection based on PAR10 score
    end
  end
  $k_{\text{best}} \leftarrow \text{argmin}_k \text{score}[k]$
  return (k_{best}, $\mathcal{F}_{\text{train}}$, feature vectors, runtimes)
end

trained. Finally, the k yielding the best average performance on the 100 validation sets is chosen.

Algorithm 8 gives a more formal description of the entire algorithm, in terms of its usage as a portfolio solver (i.e., algorithm selection given a new instance, as described above) and the random sub-sampling based training phase performed to compute the best value for k to use. The training phase starts out by computing the runtimes of all solvers on all training instances, as well as the features of these instances. It then removes all instances that cannot be solved by any solver in the portfolio within the time limit, or are solved by every solver in the portfolio within marginal time (e.g., 1 second for reasonably challenging benchmarks); learning to distinguish between solvers based on data from such instances is pointless. Along with the estimated best k, the training phase passes along this reduced set of training instances, their runtimes for each solver, and their features to the main solver selection phase. Note that the training phase does not learn any sophisticated model (e.g.,

	Pure Solvers							Portfolios		VBS
	agw-sat0	agw-sat+	**gnov-elty+**	SAT-enstein	march	pico-sat	kcnfs	SAT-zilla	***k*-NN**	
PAR10	5940	6017	5874	5892	8072	10305	6846	3578	**3151**	2482
σ	5952	5935	5951	5921	5944	5828	5891	5684	**5488**	5280
Avg Time	634	636	626	625	872	1078	783	452	**442**	341
σ	574	576	573	570	574	574	580	542	**538**	527
# Solved	290	286	293	292	190	83	250	405	**427**	457
% Solved	50.9	50.2	51.4	51.2	33.3	14.6	43.9	71.1	**74.9**	80.2

Table 7.1: Comparison of Baseline Solvers, Portfolio, and Virtual Best Solver Performances: PAR10, average runtime in seconds, and number of instances solved (timeout 1,200 seconds).

a runtime prediction model); rather, it simply memorizes the training performances of all solvers and only actually "learns" the value of k.

Despite the simplicity of this approach – compared, for example, with the description of SATzilla in [108] – it is highly efficient and outperforms SATzilla2009_R, the Gold Medal winning solver in the random category of SAT Competition 2009. In Table 7.1 the k-NN algorithm selection is compared with SATzilla_R, using the 2,247 random category instances from SAT Competitions 2002-2007 as the training set and the 570 such instances from SAT Competition 2009 as the test set. Like in the previous chapter, both portfolios are based on the following local search solvers: Ag2wsat0 [102], Ag2wsat+ [103], gnovelty+ [76], Kcnfs04 [26], March_dl04 [40], Picosat 8.46 [10], and SATenstein [55], all in the versions that are *identical* with the ones that were used when SATzilla09_R [105] entered in the 2009 SAT solver competition. To make the comparison as fair as possible, k-NN uses only the 48 core instance features that SATzilla is based on (see [108] for a detailed list of features), and trained for Par10-score. For both training and testing, the time limit is set to 1,200 seconds. Table 7.1 shows that SATzilla boosts performance of individual solvers dramatically. The pure k-NN approach pushes the performance level substantially further. It solves 22 more instances than SATzilla and closes about one third of the gap between SATzilla and the virtual best solver (VBS),[2] which solves 457 instances. Given the utter simplicity of the k-NN approach, this performance is quite remarkable.

7.1.2 Improving Nearest-Neighbor-Based Solver Selection

This section discusses two techniques to improve the performance of the algorithm selector further. First, inspired by [72], training instances that are closer to the test instance are given more weight. Second, the neighborhood size k is adapted depending on the properties of the test instance to be solved.

[2]VBS refers to the "oracle" selector that always selects the solver that is the fastest on the given test instance. Its performance is the best one can hope to achieve with algorithm selection.

Distance-Based Weighting

A natural extension of k-NN is to scale the scores of the k neighbors of an instance based on the Euclidean distance to it. Intuitively, larger weights are assigned to instances that are closer to the test instance assuming that closer instances more accurately reflect the properties of the instance at hand. Hence, Line 17 in Algorithm 8 is updated to:

$$\text{score}[k] \leftarrow \text{score}[k] + PAR10 \times \left(1 - \frac{dist}{totalDist}\right),$$

whereby *dist* is the distance between the neighboring training instance and the current instance, and *totalDist* corresponds to the sum of all such distances. We proceed analogously in Line 5 when computing the best solver for a given test instance.

Adaptive Neighborhood Size

Another idea is to learn not a single value for k, but to adapt the size of the neighborhood based on the given test instance. It is possible to partition the instance feature space by pre-clustering the training instances (we use g-means clustering [37] for this purpose). Then, a given instance belongs to a cluster when it is nearest to that cluster, whereby ties can be broken arbitrarily. This way during training instead of only learning one k that is supposed to work uniformly well, a different k is learned for each cluster.

Algorithm 8 can be adapted easily to determine such cluster-based ks. Given a test instance, first the cluster to which it belongs is identified and then the value of k that was associated with this cluster during training is used. Please observe that this clustering is not used to limit the neighborhood of a test instance. That means that neighboring instances from other clusters can still be used to determine the best solver for a given instance. The clusters are only used to determine the size of the neighborhood.

Experimental Evaluation

Observe that the two techniques, weighting and adaptive neighborhoods, are orthogonal to each other and can be combined. In the following, weighting, clustering, and their combination are compared with the pure k-NN portfolio.

Benchmark Solvers. In order to illustrate the improvements achieved by the extensions of k-NN, a new benchmark setting is introduced that mixes incomplete and complete solvers as well as industrial, crafted, and random instances. The following 21 state-of-the-art complete and incomplete SAT solvers are considered: Clasp[31], CryptoMiniSat [88], Glucose [5], LySat i/c [36], March-hi [38], March-nn [39], MXC [19], MiniSAT 2.2.0 [90], Lineling [12], PrecoSAT [11], Adaptg2wsat2009 [61], Adaptg2wsat2009++ [61], Gnovelty+2 [77], Gnovelty+2-H [77], HybridGM3 [7], Kcnfs04SAT07 [26], Picosat [10], Saps [50], TNM [100], and six parametrizations of SATenstein [55]. In addition, all industrial and crafted instances are preprocessed with SatElite (version 1.0, with default option '+pre') where the following solvers were run on both the original and preprocessed version of each instance: Clasp, CryptoMiniSat, Glucose, Lineling, LySat c, LySat i, March-hi,

	Basic k-NN	Weighting	Clustering	Weight.+Clust.
# Solved	1609	1611	1615	**1617**
# Unsolved	114	112	108	**106**
% Solved	93.5	93.6	93.8	**93.9**
Avg Runtime	588	584	584	**577**
PAR10 Score	3518	3459	3368	**3314**

Table 7.2: Average Performance Comparison of Basic k-NN, Weighting, Clustering, and the combination of both using the k-NN Portfolio.

March-nn, MiniSat, MXC, and Precosat. In that way the portfolio was composed of 37 solvers.

Benchmark Instances. As before the set of benchmark instances was comprised of $5,464$ instances selected from all SAT Competitions and Races during 2002 and 2010 [2], filtered for all instances that cannot be solved by any of the aforementioned solvers within the competition time limit of 5,000 seconds (i.e., the VBS can solve 100% of all instances).

These instances were partitioned randomly multiple times into disjoint sets of training and testing instances, as well as into more challenging groups. The complex partition was based on omitting certain sets of instances from the training set, but including them all in the test set. To asses which instances were related, it was assumed that instances starting with the same three characters belong to the same benchmark family. To this end, at random, a fraction of about 5% of benchmark families were selected among all families. This usually resulted in roughly 15% of all instances being in the test partition. Aiming for a balance of 70% training instances and 30% test instances, the second step randomly choose instances until 30% of all instances had been assigned to the test partition. Unless stated otherwise, all of the following experiments are conducted on this set of solvers and instances.

Results. Table 7.2 shows a comparison of the basic k-NN approach with the extensions of using weighting, clustering, and the combination of the two on this benchmark. Shown is the average performance in terms of number of instances solved/not solved, average runtime, and PAR10 score achieved across the 10 test sets mentioned in the previous paragraph. Note that a perfect oracle can solve all instances as instances that could not be solved by any solver within the given time limit of 5,000 seconds were discarded.

According to all of these measures, both weighting and clustering are able to improve the performance of the basic k-NN approach. This improvement is amplified when both methods are used simultaneously. The combined approach consistently outperforms basic k-NN on all our splits, solving about 0.5% more instances.

For completeness, remark that these results also translate to the SATzilla_R benchmark discussed earlier in Table 7.1. In this setting the combination of weighting and clustering is able to solve 7 more instances than the basic k-NN approach and 29 more than SATzilla_R. Here, the gap to the virtual best solver in terms of instances solved is narrowed down further to only 5% compared to 6.6% and 11.4% lost by basic k-NN and SATzilla_R, respectively.

7.2 Building Solver Schedules

While the previous section shows that the k-nn-tuned algorithm portfolio is not only able to significantly outperform the single best solver but also the highly successful SATzilla portfolio, there is still room for improvement with regard to the virtual best solver. To increase the robustness of the approach further, an alternate training methodology is considered. It is no longer feasible to tune solvers offline using the nearest neighbor clustering. As an alternative it is possible to compute a schedule that defines the sequence of solvers, along with individual time limits, given a new test instance. This sequence of solvers is then used to solve the instance. This approach is well justified by the different runtime distribution of constraint solvers. While one solver may fail to solve a given instance even in a very long time, another solver may well be able to solve the instance very quickly.

The general idea of scheduling for algorithm portfolios was previously introduced by Streeter [93] and in CP-Hydra [72]. In fact, Streeter [93] uses the idea of scheduling to *generate* algorithm portfolios. While he suggested using schedules that can suspend solvers and let them continue later on in exactly the same state they were suspended in, this section will focus on solver schedules without preemption, i.e., each solver will appear in the schedule at most once. This setting was also used in CP-Hydra, which computes a schedule of CP solvers based on k nearest neighbors. Specifically, a schedule is devised that determines which solver is run for how much time in order to attempt to solve the given instance.

It is important to first note that the *optimal* performance cannot be improved by a schedule of solvers, simply because using the fastest solver and sticking to it is the best we can hope for. Consequently, a solver schedule is still limited by the optimal performance of the VBS. In fact, the best performance possible for a schedule of solvers is limited by the VBS *with a reduced captime of the longest running solver in the schedule*. Therefore, trivial schedules that split the available time evenly between all solvers have inherently limited performance.

Nevertheless, the reason to be interested in solver schedules is to hedge our bets: It is often observed that instances that cannot be solved by one solver even in a very long time can in fact be solved by another very quickly. Consequently, by allocating a reasonably small amount of time to other solvers, it is possible to provide a safety net in case the solver selection happens to be unfortunate.

Static Schedules

The simplest approach is to compute a static schedule of solvers. For example, one could compute a schedule that solves the most training instances within the allowed time (cf. [72]). This section does slightly more, namely computing a schedule that, first, solves most training instances and that, second, requires the lowest amount of time among all schedules that are able to solve the same amount of training instances.

This problem can be formulated as an integer program (IP), more precisely as a a resource constrained set covering problem (RCSCP):

Solver Scheduling IP:

$$\min \quad (C+1)\sum_i y_i + \sum_{S,t} t x_{S,t} \tag{7.1}$$

$$s.t. \quad y_i + \sum_{(S,t) \mid i \in V_{S,t}} x_{S,t} \geq 1 \qquad \forall i \tag{7.2}$$

$$\sum_{S,t} t x_{S,t} \leq C \tag{7.3}$$

$$y_i, x_{S,t} \in \{0,1\} \qquad \forall i, S, t \tag{7.4}$$

The constraints (7.2) in this model enforce that all training instances are covered, the additional resource constraint (7.3) ensures that the overall captime C is not exceeded. Binary variables $x_{S,t}$ in (7.4) correspond to sets of instances that can be solved by solver S within a time t. These sets have cost t and a resource consumption coefficient t. Finally, to make it possible that all training instances can be covered, additional binary variables y_i are introduced. These correspond to the set that contains only item i, they have cost $C+1$ and time resource consumption coefficient 0. The objective is obviously to minimize the total cost. Due to the high costs for variables y_i (which will be one if and only if instance i cannot be solved by the schedule) the schedules which solve most instances are favored, and among those the fastest schedule (cost of $x_{S,t}$ is t) is chosen.

A Column Generation Approach

The main problem with the above formulation is the sheer number of variables. For the benchmark with 37 solvers and more than 5,000 training instances, solving the above problem is impractical, even when the timeouts t are chosen smartly such that from timeout $t1$ to the next timeout $t2$ at least one more instance can be solved by the respective solver ($V_{S,t1} \subsetneq V_{S,t2}$). In our experiments we found that the actual time to solve these IPs may at times still be tolerable, but the memory consumption was in many cases so high that we could not solve the instances.

The above stated problem can be resolved by means of column generation. Column generation (aka Dantzig-Wolfe decomposition) [24, 33] is a well-known technique for handling linear programs (LPs) with a lot of variables:

$$\min c^T x, \quad \text{s.t. } A\,x \geq b, x \geq 0 \tag{7.5}$$

Due to its size it is often not practical to solve the large system (7.5) directly. The core observation underlying column generation is that only a few variables will be non-zero in any optimal LP solution (at most as many as there are constraints). Therefore, if we knew which variables are important, we can consider a much smaller system $A'\,x' = b$ where A' contains only a few columns of A. When we choose only some columns in the beginning, LP duality theory tells us which columns that we have left out so far are of interest for the optimization of the global LP. Namely, only columns with *negative reduced costs* (which are defined based on the optimal duals of the system $A'\,x' = b$) can be candidates for variables that can help the objective to decrease further.

Column generation proceeds by considering, in turn, a *master problem* (the reduced system $A'\,x' = b$) and a *subproblem* where we select a new column to be added to the

Algorithm 9: Subproblem: Column Generation

```
1  begin
2      minRedCosts ← ∞
3      forall Solvers S do
4          T ← 0
5          forall i do
6              j ← π(i)
7              T ← T + λ_j
8              t ← Time(S, j)
9              redCosts ← t(1 − μ) − T
10             if redCosts < minRedCosts then
11                 Solver ← S
12                 timeout ← t
13                 minRedCosts ← redCost
14             end
15         end
16     end
17     if minRedCosts < 0 then return x_{Solver,timeout}
18     else return None
19 end
```

master based on its current optimal dual solution. This process is iterated until there are no more columns with a negative reduced cost. At this point, we know that an optimal solution to (7.5) has been found – even though most columns have never been added to the master problem!

When using standard LP solvers to solve the master problem and obtain its optimal duals, all that is left is solving the subproblem. To develop a subproblem generator, we need to understand how exactly the reduced costs are computed. Assume we have a dual value $\lambda_i \geq 0$ for each constraint in A'. Then, the reduced cost of a column $\alpha := (\alpha_1, \ldots, \alpha_z)^T$ is defined as $\bar{c_\alpha} = c_\alpha - \sum_i \lambda_i \alpha_i$, where c_α is the cost of column α.

Equipped with this knowledge we compute a new column for A' that has minimal reduced costs. The process is begun by adding all columns to A' that correspond to variables y. Therefore, when we want to add a new column to the model it will regard a variable $x_{S,t}$ which corresponds to the solver-runtime pair (S, t). The goal of the subproblem at each step is to suggest a solver-runtime pair that is likely to increase the objective value of the (continuous) master problem the most.

To find this solver-runtime pair, first, for all solvers S, we compute a permutation π of the instances such that the time that S needs to solve instance $\pi_S(i)$ is less than or equal that the solver needs to solve instance $\pi_S(i+1)$ (for appropriate i). Obviously, we only need to do this once for each solver and not each time we want to generate a new column.

Now, let us denote with $\lambda_i \geq 0$ the optimal dual value for the restriction to cover instance i (7.2). Moreover, denote with $\mu \leq 0$ the dual value of the resource constraint (7.3) (since that constraint enforces a lower-or-equal restriction μ is guaranteed to be non-positive).

Now, for each solver S we iterate over i and compute the term $T \leftarrow \sum_{k \leq i} \lambda_{\pi_S(k)}$ (which in each iteration we can obviously derive from the previous value for T). Let t denote the time that solver S needs to solve instance $\pi(i)$. Then, the reduced costs of the column that corresponds to variable $x_{S,t}$ are $t - t\mu - T$. We choose the column with the most negative reduced costs and add it to the master problem. If there is no more column with negative reduced costs, we stop.

It is important to note two things. First, that what we have actually done is to pretend that all columns were present in the matrix and computed the reduced costs for all of them. This is not usually the case in column generation approaches where most columns are usually found to have larger reduced costs *implicitly* rather than explicitly. Second, note that the solution returned from this process will in general not be integer but contain fractional values. Therefore, the solution obtained cannot be interpreted as a solver schedule directly.

This situation can be overcome in two ways. The first is to start branching and to generate more columns – which may still be needed by the optimal integer solution even though they were superfluous for the optimal fractional solution. This process is known in the literature as branch-and-price.

Alternatively what we do, and is in fact the reason why we solved the original problem by means of column generation in the first place, is stick to the columns that were added during the column generation process and solve the remaining system as an IP. Obviously, this is just a heuristic that may return sub-optimal schedules for the training set. However, we found that this process is very fast and nevertheless provides high quality solutions (see empirical results below). Even when the performance on the training set is at times slightly worse than optimal, the performance on the test set often turned out as good or sometimes even better than that of the optimal training schedule – a case where the optimal schedule overfits the training data.

The last aspect to address is the case where the final schedule does not utilize the entire available time. Recall that we even deliberately minimize the time needed to solve as many instances as possible. Obviously, at runtime it would be a waste of resources not to utilize the entire time that is at our disposal. In this case, we scale each solver's time in the schedule equally so that the total time of the resulting schedule will be exactly the captime C.

Dynamic Schedules

As mentioned earlier, CP-Hydra [72] is based on the idea of solver schedules. In their paper, the authors found that static schedules work only moderately well. Therefore, they introduced the idea of computing *dynamic* schedules: At runtime, for a given instance, CP-Hydra considers the ten nearest neighbors (in case of ties up to fifty nearest instances) and computes a schedule that solves most of these instances in the given time limit. That is, rather than considering all training instances, the constraints in the Solver Scheduling IP are limited to the instances in the neighborhood.

In [72] the authors use a brute-force approach to compute dynamic schedules and mention that this works due to the small neighborhood size and the fact that CP-Hydra only has three constituent solvers (note that the time to produce a dynamic schedule takes

	No Sched.	Dynamic Schedules				SAT-Hydra
	Wtg+Clu	Basic k-NN	Weighting	Clustering	Wtg+Clu	
# Solved	1617	**1621**	**1621**	1619	1618	**1621**
# Unsolved	106	**102**	**102**	104	105	**102**
% solved	93.9	**94.2**	**94.2**	94.0	94.0	**94.2**
Avg Runtime	**577**	637	629	629	631	626
PAR10 score	3314	3257	**3246**	3310	3324	3249

Table 7.3: Average performance of dynamic schedules. Addl. comparison: *SAT-Hydra*.

away time for solving the actual problem instance!). Our column generation approach, yielding potentially sub-optimal but usually high quality solutions, works fast enough to handle even 37 solvers and 5,000 instances within seconds. This allows us to embed the idea of dynamic schedules in the previously developed nearest-neighbor approach which selects optimal neighborhood sizes by random subsampling cross validation – which requires us to solve hundreds of thousands of these IPs.

Note that the idea of adaptive neighborhoods is orthogonal to dynamic solver scheduling: We can select the size of the neighborhood based on the distance to the nearest training cluster independent of whether we use that neighborhood size for solver selection or solver scheduling. Moreover, the idea of giving more weight to instances closer to the test instance can also be incorporated in solver scheduling. This is another idea that CP-Hydra also exploits, albeit in a slightly different fashion than shown here. Here we adapt the objective function in the Solver Scheduling IP by multiplying the costs for the variables y_i (recall that originally these costs were $C+1$) with $2 - \frac{\text{dist}_i}{\text{totalDist}}$. This favors schedules that solve more training instances that are closer to the one that is to be solved.

Table 7.3 compares the four resulting dynamic schedules with our best algorithm selector from Section 7.1.2. Moreover, we also used a setting inspired by the CP-Hydra approach. Here, we use a fixed size neighborhood of ten instances to build a dynamic schedule by means of column generation. Moreover, for this approach we use the weighting scheme introduced in [72]. We refer to this approach as SAT-Hydra.

Observe that these dynamic schedules are all achieving roughly the same performance. Weighting and clustering do not appear to have any significant impact on performance. Moreover, all dynamic portfolios consistently outperform even our best algorithm selector, albeit only slightly: The dynamic schedule increase the number of instances solved by roughly one quarter percent.

Semi-Static Solver Schedules

Clearly, dynamic schedules do not result in the improvements that we had hoped for. Here we therefore consider another way of creating a solver schedule. Observe that the algorithm selection portfolios that we developed in Section 7.1.1 can themselves be considered solvers. This means that we can add the portfolio itself to our set of constituent solvers and compute a "static" schedule for this augmented collection of solvers. We quote "static" here because the resulting schedule is of course still instance-specific. After all, the algorithm selector portfolio chooses one of the constituent solvers based on the test instance's features. We

	No Sched.	Static Sched.	Semi-Static Schedules			
	Wtg+Clu	Wtg+Clu	Basic k-NN	Weighting	Clustering	Wtg+Clu
# Solved	1617	1572	1628	1635	1633	**1636**
# Unsolved	106	151	94.6	87.5	90.2	**87.2**
% solved	93.9	91.2	94.6	94.9	94.8	**95.0**
Avg Runtime	577	562	448	451	**446**	449
PAR10 score	3314	4522	2896	2728	2789	**2716**

Table 7.4: Average performance of semi-static schedules compared with no schedules and with static schedules based only on the available solvers.

Schedule by	# Solved	# Unsolved	% Solved	Avg Runtime (s)	PAR10 score
Optimal IP	1635.8	87.1	95.0	442.5	2708.4
Column Generation	1635.7	87.2	95.0	448.9	2716.2

Table 7.5: Comparison of Column Generation and the Solution to the Optimal IP.

refer to the result of this process as *semi-static solver schedules*.

Depending on which of the portfolios from Section 7.1.1 used we obtain again four semi-static schedules. We show the performance of these portfolios in Table 7.4. While weighting and clustering did not lead to performance improvements for dynamic schedules, here we observe that the relative differences in performance between basic k-NN and its extensions shown in Section 7.1.2 translates to the setting with scheduling as well.

Moreover, semi-static scheduling significantly improves the overall performance (compare with the first column in the table for the best results without scheduling). In terms of instances solved, all semi-static schedules solve at least 20 more instances within the time limit. Again, the combination of weighting and clustering achieves the best performance and it narrows the gap in percentage of instances solved to nearly 5%. For further comparison, the second column shows the performance of a static schedule that was trained on the entire training set and is the same for all test instances. This confirms the finding in [72] that static solver schedules are indeed inferior to dynamic schedules, and finds that they are considerably outperformed by semi-static solver schedules.

Quality of results generated by Column Generation. Table 7.5 illustrates the performance of the Column Generation approach. The table shows a comparison of the resulting performance achieved by the *optimal* schedule. In order to compute the optimal solution to the IP we used Cplex on a machine with sufficient memory and a 15 second resolution to fit the problem into the available memory. As can be observed, the column generation is able to determine a high quality schedule that results in a performance that nearly matches the one of the optimal schedule according to displayed measures.

	Semi-Static Schedules	Fixed-Split Schedules			
	Wtg+Clu	Basic k-NN	Weighting	Clustering	Wtg+Clu
# Solved	1636	1637	1641	1638	**1642**
# Unsolved	87.2	94.6	87.5	90.2	**87.2**
% solved	95.0	95.0	95.3	95.1	**95.3**
Avg Runtime	449	455	447	452	**445**
PAR10 score	2716	2686	2570	2652	**2554**

Table 7.6: Average Performance Comparison of Basic k-NN, Weighting, Clustering, and the combination of both using the k-NN Portfolio with a Static Schedule for 10% of the total available runtime and the Portfolio on the remaining runtime.

Fixed-Split Selection Schedules

Based on this success, we consider a parametrized way of computing solver schedules. As discussed earlier, the motivation for using solver schedules is to increase robustness and hedge against an unfortunate selection of a long-running solver. At the same time, the best achievable performance of a portfolio is that of the VBS *with a captime of the longest individual run*. In both dynamic and semi-static schedules, the runtime of the longest running solver(s) was determined by the column generation approach working solely on training instances. This procedure inherently runs the risk of overfitting the training set.

Consequently, we now consider splitting the time between an algorithm selection portfolio and the constituent solvers based on a parameter. For example, we could allocate 90% of the available time for the solver selected by the portfolio. For the remaining 10% of the time, we run a static solver schedule. We refer to these schedules as *90/10-selection schedules*. Note that choosing a fixed amount of time for the schedule of constituent solvers is likely to be suboptimal for the training set but offers the possibility of improving test performance.

Table 7.6 captures the corresponding results. We observe clearly that using this restricted application of scheduling is able to outperform our best approach so far (semi-static scheduling, shown again in the first column). We are able to solve nearly 1642 instances on average which is 6 more than we were able to solve before. The gap to the virtual best solver is narrowed down to a mere 4.69 percent! Recall that we consider a highly diverse set of benchmark instances from the Random, Crafted, and Industrial categories. Moreover, we do not work with plain random splits, but splits where complete families of instances in the test set are not represented in the training set at all. In this setting, an accuracy above 95% of the VBS is truly remarkable. Moreover, compared to the plain k-NN approach that was started with, the fixed-splitselection schedules close roughly one third of the gap to the VBS.

7.3 Chapter Summary

This chapter showed how the ISAC methodology could be adopted to dynamically create clusters in a more refined instance-specific manner. Specifically, this chapter considered the problem of algorithm selection and scheduling so as to maximize performance when

	SATzilla_R	SAT-Hydra	k-NN	90-10	VBS
# Solved	405	419	427	**435**	457
# Unsolved	165	151	143	**135**	113
% solved	71.5	73.5	74.9	**76.3**	80.2
Avg Runtime	452	313	441.9	**400**	341
PAR10 score	3578	1211	3151	**2958**	2482

Table 7.7: Comparison of Major Portfolios for the SAT-Rand Benchmark (570 test instances, timeout 1,200 seconds).

given a hard time limit within which a solution needs to be provided. Two improvements were considered for the simple nearest-neighbor solver selection, weighting and adaptive neighborhood sizes based on clustering.

Furthermore, this chapter showed how the training of the solvers could also be done dynamically by developing a light-weight optimization algorithm to compute near-optimal schedules for a given set of training instances. This allows us to provide an extensive comparison of pure algorithm selection, static solver schedules, dynamic solver schedules, and semi-static solver schedules which are essentially static schedules combined with an algorithm selector.

It was shown that the semi-static schedules work the best among these options. Finally, two alternatives were compared: using the optimization component or using a fixed percentage of the allotted time when deciding how much time to allocate to the solver suggested by the algorithm selector. In either case, a static schedule was used for the remaining time. This latter parametrization allowed us to avoid overfitting the training data and overall resulted in the best performance.

The discussed approach was tested on a highly diverse benchmark set with random, crafted, and industrial SAT instances where we even deliberately removed entire families of instances from the training set. Semi-static selection schedules demonstrated an astounding performance and solved, on average, over 95% of the instances that the virtual best solver is able to solve.

As a final remark, Table 7.7 closes the loop and considers again the first benchmark set from Section 7.1.1 which compared portfolios for SAT Competition's random category benchmark set based on the same solvers as the gold-medal winning SATzilla_R. Overall, we go up from 405 (or 88.6% of the VBS) for SATzilla_R to 435 (or 95.1% of the VBS) instances solved for our fixed-split semi-static solver schedules. In other words, fixed-split selection schedule closes over 50% of the performance gap between SATzilla_R and the VBS.

Chapter Eight

Training Parallel Solvers

In the past decade, solver portfolios have boosted the capability to solve hard combinatorial problems. Portfolios of existing solution algorithms have excelled in competitions in satisfiability (SAT), constraint programming (CP), and quantified Boolean formulae (QBF) [108, 72, 91].

In the past years, a new trend has emerged, namely the development of parallel solver portfolios. The gold-winning ManySAT solver [36] is, when features like clause-sharing are ignored, a static parallel portfolio of the MiniSAT solver [27] with different parameterizations. At the SAT Competition 2011, an extremely simple static parallel portfolio, ppfolio [80], dominated the wall-clock categories on random and crafted SAT instances and came very close to winning the applications category as well. In [74] another method was introduced to compute static parallel schedules that are optimal with respect to the training instances, based on formulating the problem as a non-linear optimization problem and considering only sequential constituent solvers.

The obvious next step is to therefore consider dynamic parallel portfolios, i.e., portfolios that are composed based on the features of the given problem instance. Traditionally, sequential portfolios simply *select* one of the constituent solvers which appears best suited for the given problem instance. And as seen in Chapter 7, at least since the invention of CP-Hydra [72] and SatPlan [92], sequential portfolios also *schedule* solvers. That is, they may select more than just one constituent solver and assign each one a portion of the time available for solving the given instance.

The solver presented in the end of Chapter 7 dominated the sequential portfolio solvers at the SAT Competition 2011 was where it won gold medals in the CPU-time category on random and crafted instances. In this chapter, the 3S methodology is augmented to devise dynamic parallel SAT solver portfolios.

8.1 Parallel Solver Portfolios

The objective of this chapter is to show how to generalize the ISAC technology for the development of parallel SAT solver portfolios. Recall that at the core of 3S lie two optimization problems. The first is the selection of the long running solver primarily based on the maximum number of instances solved. The second is the solver scheduling problem.

Consider the first problem when there are $p > 1$ processors available. The objective is to select p solvers that, as a set, will solve the most number of instances. Note that this problem can no longer be solved by simply choosing the one solver that solves most instances in time. Moreover, it is now necessary to decide how to integrate the newly chosen solvers with the ones from the static schedule. The second problem is the solver scheduling problem discussed before, with the additional problem that solvers need to be assigned to processors so that the total makespan is within the allowed time limit.

A major obstacle in solving these problems efficiently is the symmetry induced by the identical processors to which each solver can be assigned. Symmetries can hinder optimization very dramatically as equivalent (partial) schedules (which can be transformed into one another by permuting processor indices) will be considered again and again by a systematic solver. For example, when there are 8 processors, for each schedule over 40,000 (8 factorial) equivalent versions exist. An optimization that used to take about half a second may now easily take 6 hours.

Another consideration is the fact that a parallel solver portfolio may obviously include parallel solvers as well. Assuming there are 8 processors and a parallel solver employs 4 of them, there are 70 different ways to allocate processors for this solver. The developed portfolio will have 37 sequential and 2 4-core parallel solvers. The solver scheduling IP that needs to be solved for this case has over 1.5 million variables.

8.1.1 Parallel Solver Scheduling

Both optimization problems are addressed at the same time by considering the following IP. Let $t_S \geq 0$ denote the minimum time that solver S must run in the schedule, let $M = \{S; |; t_S > 0\}$ be the set of solvers that have a minimal runtime, let p be the number of processors, and let $n_S \leq p$ denote the number of processors that solver S requires.

Parallel Solver Scheduling IP - CPU time

$$
\begin{aligned}
\min \quad & (pC+1)\sum_i y_i + \sum_{S,t,P} t n_S x_{S,t,P} & & \\
s.t. \quad & y_i + \sum_{(S,t) \mid i \in V_{S,t}, P \subseteq \{1,\ldots,p\}, |P| = n_S} x_{S,t,P} \geq 1 & & \forall i \\
& \sum_{S,t,P \subseteq \{1,\ldots,p\} \cup \{q\}, |P| = n_S} t x_{S,t,P} \leq C & & \forall q \in \{1,\ldots,p\} \\
& \sum_{S,t,P \subseteq \{1,\ldots,p\}, |P| = n_S, t \geq t_S} x_{S,t,P} \geq 1 & & \forall S \in M \\
& \sum_{S,t,P \subseteq \{1,\ldots,p\}, |P| = n_S} x_{S,t,P} \leq N & & \\
& y_i, x_{S,t,P} \in \{0,1\} & & \forall i, S, t, P \subseteq \{1,\ldots,p\}, |P| = n_S
\end{aligned}
$$

Variables y_i are exactly what they were before. There are now variables $x_{S,t,P}$ for all solvers S, time limits t, and subsets of processors $P \subseteq \{1,\ldots,p\}$ with $|P| = n_S$. $x_{S,t,P}$ is one if an only if solver S is run for time t on the processors in P in the schedule.

The first constraint is again to solve all instances with the schedule or count them as not covered. There is now a time limit constraint for each processor. The third set of constraints ensures that all solvers that have a minimal solver time are included in the schedule with an appropriate time limit. The last constraint finally places a limit on the number of solvers that can be included in the schedule.

The objective is again to minimize the number of uncovered instances. The secondary criterion is to minimize the total CPU time of the schedule.

Note that this problem needs to be solved both offline to determine the static solver schedule (for this problem $M = \emptyset$ and the solver limit is infinite) and *during the execution phase* (when M and the solver limit are determined by the static schedule computed offline). Therefore, it is absolutely necessary to solve this problem quickly, despite its huge size and its inherent symmetry caused by the multiple processors.

Note also that the parallel solver scheduling IP does not directly result in an executable solver schedule. Namely, the IP does not specify the actual start times of solvers. In the sequential case this does not matter as solvers can be sequenced in any way without affecting the total schedule time or the number of instances solved. In the parallel case, however, it is necessary to ensure that the parallel processes are in fact run in parallel.

This aspect is omitted from the IP above to avoid further complicating the optimization. Instead, after solving the parallel solver IP, the solvers are heuristically scheduled in a best effort approach, whereby solvers may be preempted and the runtime of the solvers may eventually be lowered to obtain a legal schedule. In the experiments presented later in the chapter it is shown that in practice the latter is never necessary. Hence, the quality of the schedule is never diminished by the necessity to schedule processes that belong to the same parallel solver at the same time.

8.1.2 Solving the Parallel Solver Scheduling IP

We cannot afford to solve the parallel solver scheduling IP exactly during the execution phase. Each second spent on solving this problem is one second less for solving the actual SAT instance. Hence, like 3S, we revert to solving the problem heuristically by not considering variables that were never introduced during column generation.

While 3S **could afford to price all columns** in the IP during each iteration, fortunately it is not actually necessary to do this here. Consider the reduced costs of a variable. Denote with $\mu_i \leq 0$ the dual prices for the instance-cover constraints, $\pi_q \leq 0$ the dual prices for the processor time limits, $\nu_S \geq 0$ the dual prices for the minimum time solver constraints, and $\sigma \leq 0$ the dual price for the limit on the number of solvers. Finally, let $\bar{\nu}_S = \nu_S$ when $S \in M$ and 0 otherwise. Then:

$$\bar{c}_{S,t,P} = n_S t - \sum_{i \in V_{S,t}} \mu_i - \sum_{q \in P} t\pi_q - \bar{\nu}_S - \sigma.$$

The are two important things to note here: First, the fact that only variables introduced during the column generation process are considered means that the processor symmetry is reduced in the final IP. While it is not impossible, it is unlikely that the variables that would form a symmetric solution to a schedule that can already be formed from the variables already introduced would have negative reduced costs.

Second, to find a new variable that has the most negative reduced costs, it is not necessary to iterate through all $P \subseteq \{1, \ldots, p\}$ for all solver/time pairs (S, t). Instead, the processors can be ordered by their decreasing dual prices. The next variable introduced will use the first n_S processors in this order as all other selections of processors would result in higher reduced costs.

8.1.3 Minimizing Makespan and Post Processing the Schedule

Everything is now in place to develop the parallel SAT solver portfolio. In the training phase a static solver schedule is computed based on all training instances for 10% of the available time. This schedule is used to determine a set M of solvers that must be run for at least the static scheduler time at runtime. During the execution phase, given a new SAT instance its features are computed, the k closest training instances are determined, and a parallel schedule is computed that will solve as many of these k instances in the shortest amount of CPU time possible.

In these experiments a second variant of the parallel solver scheduling IP is considered where the secondary criterion is not to minimize CPU time but the makespan of the

schedule. The corresponding IP is given below, where variable m measures the minimum idle time for all processors. The reduced cost computation changes accordingly.

Parallel Solver Scheduling IP - Makespan

$$
\begin{aligned}
\min \quad & (C+1)\sum_i y_i - m & & \\
s.t. \quad & y_i + \sum_{(S,t) \mid i \in V_{S,t}, P \subseteq \{1,\ldots,p\}, |P|=n_S} x_{S,t,P} \geq 1 & & \forall i \\
& m + \sum_{S,t,P \subseteq \{1,\ldots,p\} \cup \{q\}, |P|=n_S} t x_{S,t,P} \leq C & & \forall q \in \{1,\ldots,p\} \\
& \sum_{S,t,P \subseteq \{1,\ldots,p\}, |P|=n_S, t \geq t_S} x_{S,t,P} \geq 1 & & \forall S \in M \\
& \sum_{S,t,P \subseteq \{1,\ldots,p\}, |P|=n_S} x_{S,t,P} \leq N & & \\
& y_i, x_{S,t,P} \in \{0,1\} & & \forall i,S,t,P \subseteq \{1,\ldots,p\}, |P| = n_S
\end{aligned}
$$

Whether the CPU time or makespan is minimized, as remarked earlier, the result is post processed by assigning actual start times to solvers heuristically. The resulting solver times are also scaled to use as much of the available time as possible. For low values of k, schedules are often computed that solve all k instances in a short amount of time without utilizing all available processors. In this case, new solvers are assigned to the unused processors in the order of their ability to solve the highest number of the k neighboring instances.

8.2 Experimental Results

Using the methodology above, two parallel portfolios are built. The first based on the 37 constituent solvers of 3S. This portfolio is referred to as p3S-37. The second portfolio built includes two additional solvers, 'Cryptominisat (2.9.0)' [89] and 'Plingeling (276)' [13], both executed on four cores. This portfolio is referred to as p3S-39. It is important to emphasize again that all solvers that are part of our portfolio were available *before* the SAT Competition 2011. In the experiments these parallel portfolios will be compared with the parallel solver portfolio 'ppfolio' [80] as well as 'Plingeling (587f)' [14], both executed on eight cores. Note that these competing solvers are new solvers that were introduced for the SAT Competition 2011.

The benchmark set of SAT instances is the same as in prior sections composed of the 5,464 instances from all SAT Competitions and Races between 2002 and 2010 [2], the 1,200 (300 application, 300 crafted, 600 random) instances from last years SAT Competition 2011 were also added. Based on this large set of SAT instances, a number of benchmarks are created. Based on all SAT instances that can be solved by at least one of the solvers considered in p3S-39 within 5,000 seconds, an equal 10 partition is created. This partition is used to conduct a ten-fold cross validation, whereby in each fold nine partitions are used as the training set (for building the respective p3S-37 and p3S-39 portfolios), and the performance is evaluated on the partition that was left out before. For this benchmark average performance over all ten splits is reported. On top of this cross-validation benchmark, the split induced by the SAT Competition 2011 is also considered. Here all instances prior to

CPU Time	10	25	50	100	200
Average (σ)	320 (45)	322 (43.7)	329 (42.2)	338 (43.9)	344 (49.9)
Par 10 (σ)	776 (241)	680 (212)	694 (150)	697 (156)	711 (221)
# Solved (σ)	634 (2.62)	636 (2.22)	636 (1.35)	636 (1.84)	636 (2.37)
% Solved (σ)	99.0 (0.47)	99.2 (0.39)	99.2 (0.27)	99.2 (0.28)	99.2 (0.41)

Table 8.1: Average performance comparison of parallel portfolios when optimizing CPU time and varying neighborhood size k based on 10-fold cross validation.

Makespan	10	25	50	100	200
Average (σ)	376.1 (40.8)	369.2 (42.9)	374 (40.7)	371 (40.8)	366 (36.9)
Par 10 (σ)	917 (200)	777 (192)	782 (221)	750 (153)	661 (164)
# Solved (σ)	633 (2.16)	635 (2.28)	634.9 (2.92)	635 (1.89)	637 (2.01)
% Solved (σ)	98.8 (0.39)	99.1 (0.39)	99.1 (0.46)	99.2 (0.32)	99.3 (0.34)

Table 8.2: Average performance comparison of parallel portfolios when optimizing Makespan and varying neighborhood size k based on 10-fold cross validation.

the competition are used as the training set, and the SAT Competition instances as the test set. Lastly, a competition split was created based on application instances only.

As performance measures the number of instances solved, average runtime, and PAR10 score are considered. The PAR10 is a penalized average runtime where instances that time out are penalized with 10 times the timeout. Experiments were run on dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors with 24 GB of DDR-3 memory.

Impact of the IP Formulation and Neighborhood Size Tables 8.1 and 8.2 show the average cross-validation performance of p3S-39 when using different neighborhood sizes k and the two different IP formulations (tie breaking by minimum CPU time or minimizing schedule makespan). As can be seen, the size of the neighborhood k affects the most important performance measure, the number of instances solved, only very little. There is a slight trend towards larger k's working a little bit better. Moreover, there is also not a great difference between the two IP formulations, but on average it is found that the version that breaks ties by minimizing the makespan solves about 1 instance more per split. Based on these results, p3S in the future refers to the portfolio learned on the respective training benchmark using $k = 200$ and the IP formulation that minimizes the makespan.

8.2.1 Impact of Parallel Solvers and the Number of Processors

Next the impact of employing parallel solvers in the portfolio is demonstrated. Tables 8.3 and 8.4 compare the performance of p3S-37 (without parallel solvers) and p3S-39 (which employs two 4-core parallel solvers) on the cross-validation and on the competition split. A small difference can be obsereved in the number of solved instances in the cross-validation, and a significant gap in the competition split.

Two issues are noteworthy about that competition split. First, since this was the latest competition, the instances in the test set of this split are probably significantly harder than

Cross Validation	p3S-37		p3S-39	
	4 core	8 core	4 core	8 core
Average (σ)	420 (22.1)	**355 (31.3)**	435 (48.5)	366 (36.9)
Par 10 (σ)	991 (306)	679 (176)	1116 (256)	**661 (164)**
Solved (σ)	630 (4.12)	633 (2.38)	631 (2.75)	**637 (2.01)**
% Solved (σ)	98.3 (0.63)	98.8 (0.35)	98.5 (0.49)	**99.3 (0.34)**

Table 8.3: Performance of 10-fold cross validation on all data. Results are averages over the 10 folds.

Competition	p3S-37		p3S-39		VBS
	4 cores	8 cores	4 cores	8 cores	
Average	1907	1791	1787	**1640**	1317
Par 10	12,782	12,666	11,124	**10,977**	10,580
Solved	843	865	853	**892**	953
% Solved	70.3	72.1	71.1	**74.3**	79.4

Table 8.4: Performance of the solvers on all 2011 SAT Competition data.

the instances from earlier years. The relatively low percentage of instances solved even by the best solvers at the SAT Competition 2011 is another indication for this. Second, in some instance families in this test set are completely missing in the training partition. That is, for a good number of instances in the test set there may be no training instance that is very similar. These features of any competition-induced split (which is the realistic split scenario!) explain why the average cross-validation performance is often significantly better than the competition performance. Moreover, they explain why p3S-39 has a significant advantage over p3S-37: When a lot of the instances are out of reach of the sequential solvers within the competition timeout then the portfolio must necessarily include parallel solvers to perform well.

As a side remark: the presence of parallel portfolios is what makes the computation of parallel portfolios challenging in the first place. In the extreme case, we could otherwise have as many processors as parallel processors, and then a trivial portfolio would achieve the performance of the virtual best solver. That is to say: The more processors one has, the easier sequential solver selection becomes. To show what would happen when the selection is made harder than it actually is under the competition setting and reduced the number of available processors to 4. For both p3S-37 and p3S-39, the cross-validation performance decreases only moderately while, under the competition split, performance decays significantly. At the same time, the advantages of p3S-39 over p3S-37 shrink a lot. As one would expect, the advantage of employing parallel solvers decays with a shrinking number of processors.

8.2.2 Parallel Solver Selection and Scheduling vs. State-of-the-Art

The dominating parallel portfolio to date is 'ppfolio' [80]. In the parallel track at the SAT Competition 2011, it won gold in the crafted and random categories and came in just shy

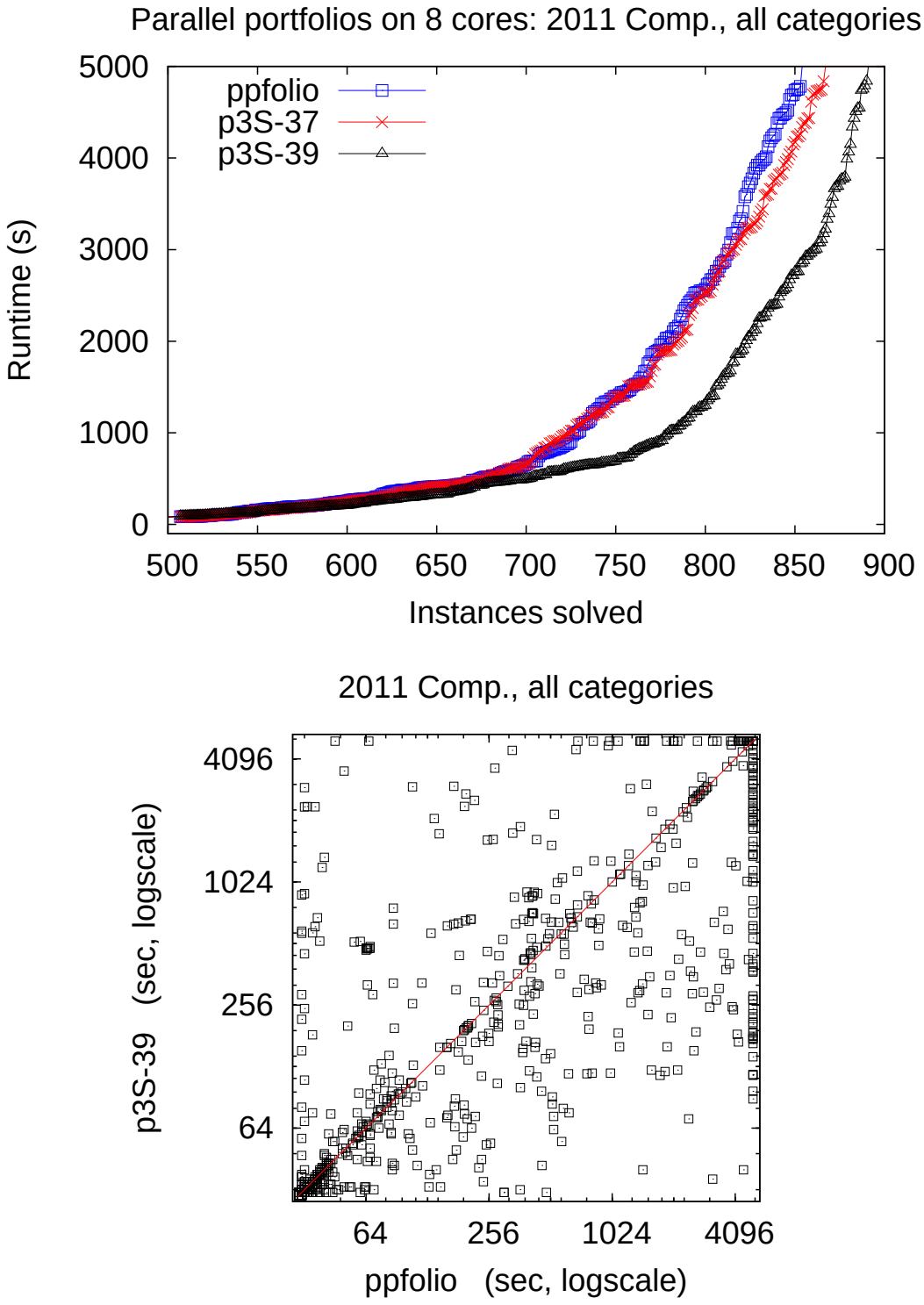

Figure 8.1: Comparison on all 1200 instances used in the 2011 SAT Competition, across all categories. Left: cactus plot depicting the scaling behavior of solvers. Right: per-instance comparison between ppfolio and p3S-39.

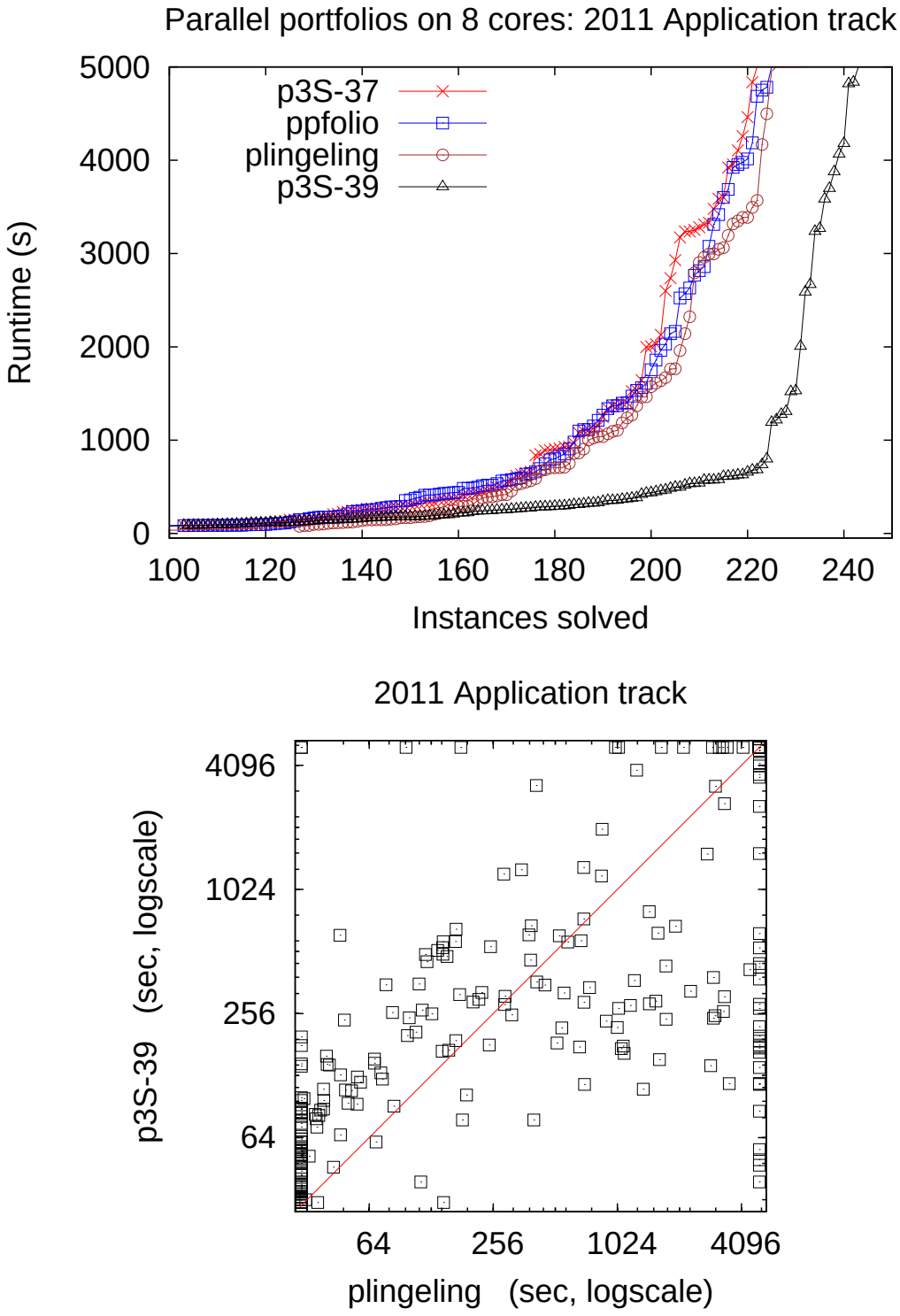

Figure 8.2: Comparison on the 300 application category instances used in the 2011 SAT Competition. Left: cactus plot depicting the scaling behavior of solvers. Right: per-instance comparison between Plingeling and p3S-39.

to winning the application category as well where it was beat by just one instance. In the application category, the winning solver was 'Plingeling (587f)' run on 8 cores. Both competing approaches are compared in Figures 8.1 and 8.2.

The left hand side plot in Figure 8.1 shows the scaling behavior in the form of a "cactus plot" for 8-core runs of ppfolio, p3S-37, and p3S-39,[1] for the competition split containing all 1,200 instances used in the 2011 SAT Competition. This plot shows that p3S-39 (whose curve stays the lowest as it moves to the right) can solve significantly more instances than the other two approaches for any given time limit larger than around 800 sec. It is also seen that p3S-37, based solely on sequential constituent solvers, performs similar to ppfolio for time limits up to 3,000 sec, and begins to outperform it for larger time limits.

The right hand side plot in Figure 8.1 shows the per-instance performance of p3S-39 vs. ppfolio, with runtimes in log-scale on both axes. More points being below the diagonal red line signifies that p3S-39 is faster than ppfolio on a large majority of the instances. ppfolio also times out on many instances that p3S-39 can solve, as evidenced the large number of points on the right margin of the plot.

Overall, p3S-39 was able to solve 892 instances, 47 more than ppfolio. p3S-37 was somewhere in-between, solving 20 more than ppfolio. In fact, even with only 4 cores, p3S-37 and p3S-39 solved 846 and 850 instances, respectively, more than the 845 ppfolio solved on 8 cores.

Figure 8.2 shows similar comparisons, but on the competition split restricted to the application category, and with Plingeling as one of the competing solvers. The cactus plot on the left still shows a significantly better scaling behavior of p3S-39 than both Plingeling and ppfolio. The scatter plot shows that Plingeling, not surprisingly, is able to solve several easy instances within just a few seconds (as evidenced by the points on the bottom part of the left edge of the plot), but begins to take more time than p3S-39 on challenging instances and also times out on many more instances (shown as points on the right edge of the plot).

Overall, with 8 cores, p3S-39 solved 248 application category instances, 23 more than ppfolio and 22 more than Plingeling. Moreover, p3S-37, based only on sequential constituent solvers, was only two instances shy of matching Plingeling's performance.

8.3 Chapter Summary

The Chapter expanded the previously introduced 3S solver by presenting a methodology for devising dynamic parallel solver portfolios. Core methods from machine learning (nearest neighbor classification) and from optimization (integer programming and column generation) are combined to select parallel solver schedules. Different formulations of the underlying optimization problems are compared and it was found that minimizing makespan as a tie breaking rule works slightly better than minimizing CPU time. The resulting portfolio, p3S-39, was compared with the current state-of-the-art parallel solvers on instances from all SAT categories and from the application category only. It was found that p3S-39 marks a significant improvement in the ability to solve SAT instances.

[1] The plots shown here are for the CPU time optimization variant of p3S-37 and p3S-39. The ones for makespan optimization were very similar.

Chapter Nine

Adaptive Solver

Search is an integral part of solution approaches for NP-hard combinatorial optimization and decision problems. Once the ability to reason deterministically is exhausted, state-of-the-art solvers try out different alternatives which may lead to an improved (in case of optimization) or feasible (in case of satisfaction) solution. This consideration of alternatives may take place highly opportunistically as in local search approaches, or systematically as in backtracking-based methods.

Efficiency could be much improved if one could effectively favor alternatives that lead to optimal or feasible solutions and a search space partition that allows short proofs of optimality or infeasibility. After all, the existence of an "oracle" is what distinguishes a non-deterministic from a deterministic Turing machine. This of course means that perfect choices are impossible to guarantee. The important insight is to realize that this is a worst-case statement. In practice, one may still hope to be able to make very good choices on average.

The view outlined above has motivated research on exploiting statistical methods to guide the search. The idea of using survey propagation in SAT [18] has led to a remarkable performance improvement of systematic solvers for random SAT instances. In stochastic offline programming [65], biased randomized search decisions are based on an offline training of the solver. A precursor of ISAC here offline training is used to associate certain features of the problem instance with specific parameter settings for the solver, whereby the latter may include the choice of branching heuristic to be used. In [81] branching heuristics for quantified Boolean formulae (QBF) were selected based on the features of the current subproblem which led to more robust performance and solutions to formerly unsolved instances.

In this chapter, the idea of instance-specific algorithm configuration is combined with the idea of a dynamic branching scheme that bases branching decisions on the features of the current **subproblem** to be solved.

In short, the chapter follows-up on the idea of choosing a branching heuristic dynamically based on certain features of the current subproblem. This idea, to adapt the search to the instance or subproblem to be solved, is by no means new. The dynamic search engine [28] for example adapts the search heuristics based on the current state of the search. In [56], value selection heuristics for Knapsack were studied and it was found that accuracy of search guidance may depend heavily on the effect that decisions higher in the search tree have on the distribution of subproblems that are encountered deeper in the tree. The latter obviously creates a serious chicken-and-egg problem for statistical learning approaches: **the distribution of instances that require search guidance affects the choice of heuristic but the latter then affects the distribution of subproblems that are encountered deeper in the tree.** In [81] a method for adaptive search guidance for QBF solvers was based on logistic regression. The issue of subproblem distributions was addressed by adding subproblems to the training set that were encountered during previous runs.

Inspired by the success of the approach in [81], the approach presented in this chapter aims to boost the Cplex MIP solver to faster solve set partitioning problems. To this end, the branching heuristics were modified, basing them on the features of the current subproblem to be solved. The objective of this study is to find out whether such a system can be effectively trained to improve the performance of a generalized solver for a specific

application. Like in previous chapters, training instances are clustered according to their features and an assignment of branching heuristics to clusters is determined that results in the best performance when the branching heuristic is dynamically chosen based on the current subproblem's nearest cluster. The approach is then examined and evaluated on the MIP-solver Cplex that we use to tackle set partitioning problems. These experiments show that this approach can effectively boost search performance even when trained on a rather small set of instances.

9.1 Learning Dynamic Search Heuristics

ISAC is adapted by modifying the systematic solver used to tackle the combinatorial problem in question. The following approach is employed:

1. First, cluster the training instances based on the normalized feature vectors like in the original ISAC.
2. Parametrize the solver by leaving open the association of branching heuristic to cluster.
3. At runtime, whenever the solver reaches a new search node (or at selected nodes), the features of the current subproblem are computed.
4. Compute the nearest cluster and use the corresponding heuristic to determine the next branching constraint.

In this way, the problem has been reduced to finding a good assignment of heuristics to clusters. At this point the problem is stated in such a way that a standard instance-oblivious algorithm configuration system can be used to find such an assignment. And like before, GGA is employed for tuning.

Note how this approach circumvents the chicken-and-egg problem mentioned in the beginning that results from the tight correlation of the distribution of subproblems encountered during search and the way branching constraints are selected. Namely, by associating heuristics and clusters *simultaneously* it is implicitly taken into account that changes in the branching strategy result in different subproblem distributions, and that the best branching decision at a search node depends heavily on the way how branching constraints will be selected further down in the tree.

That being said, the clusters themselves should reflect not only the root-node problems but also the subproblems that may be encountered during search. To this end, the training set is expanded with subproblems encountered during the runs of individual branching heuristics on the training instances to the clusters. This changes the shape of the clusters and may also create new ones. However, note that these subproblems are *not* used to learn a good assignment of heuristics to clusters which is purely based on the original training instances. This assures that the assignment of heuristics to clusters is not based on subproblems that will not be encountered.

9.2 Boosting Branching in Cplex for SPP

The methodology established above will now be applied to improve branching in the state-of-the-art MIP solver Cplex when solving set partitioning problems.

Definition Given *items* $1 \dots n$ and a collection of sets of these items, which will be referred to as *bags*, and a cost associated with each bag, the *set partitioning problem (SPP)* consists in finding a set of bags such that the union of all bags contains all items, the bags are pairwise intersection-free, and the cost of the selection is minimized.

To apply ISAC, instance features need to be defined for set partitioning problems, and various branching heuristics that our solver can choose from need to be devised.

9.2.1 Set Partitioning Features

In order to characterize instances to the set partitioning problem, the following vectors are computed:

- the normalized cost vector $c' \in [1, 100]^m$,
- the vector of bag densities $(|S_i|/n)_{i=1\dots m}$,
- the vector of item costs $(\sum_{i,j \in S_i} c'_i)_{j=1\dots n}$,
- the vector of item coverings $(|\{i \mid j \in S_i\}|/m)_{j=1\dots n}$,
- the vector of costs over density $(c'_i/|S_i|)_{i=1\dots m}$,
- the vector of costs over square density $(c'_i/|S_i|^2)_{i=1\dots m}$,
- the vector of costs over $k \log k$-density $(c'_i/(|S_i| \log |S_i|))_{i=1\dots m}$, and
- the vector of root-costs over square density $(\sqrt{c'_i}/|S_i|^2)_{i=1\dots m}$.

As features the averages, median, standard deviations, and the entropies of all these statistics are computed for all vectors. In addition to the the eight vectors above one more feature is added that represents the number of sets divided by the number of items, therefore ending up with 33 features which are used to characterize set partitioning instances. One of the benefits of this feature set is that it is invariant under column and row permutations of the problem matrix.

9.2.2 Branching Heuristics

It is also necessary to provide a portfolio of different branching selection heuristics. The following are compared, all of which are implement using Cplex's built-in branching methods.

Most-Fractional Rounding (Fractional Rounding):

One of the simplest MIP branching techniques is to select the variable that has a relaxed LP solution whose fractional part is closest to 0.5 and to round it first. That is, if the fractional part is lower than 0.5 an upper bound is first enforced to the floor of the current LP value. Otherwise, the branch is chosen where the lower bound was enforced as the lower bound of the ceiling of the current LP value.

Most-Fractional Up (Fractional Up):

For binary IPs like set partitioning it has often been noted that rounding the branching variable up first is beneficial. The common understanding is that forcing a variable to 1 will force many other binaries to 0 and thus increases the integrality of many variables when diving in this direction. This in turn may lead to shallower subtrees and integer feasible solutions faster. With this motivation in mind this branching heuristic selects the variable with the most fractional LP value and enforce its ceiling as lower bound first.

Best Pseudo-Cost Lower Estimate First (Pseudo Best):

One of the most substantial contributions to search in mixed integer programming was the discovery that the running average of the per unit cost-deprivation of prior branching decisions on a variable provides a very good estimate of the per unit deprivation that are likely to be encountered when imposing a new branching constraint on a variable. This is surprising for two reasons. First, there is no theoretical explanation yet why a variable X with fractional value 36.7, when rounded up, should deprivate the relaxed objective cost by the same amount as at a later time when X has fractional value 7.7. Second, and this is of more importance for binary IPs, assume at the root node variable Y is branched on and set to 0 first. There is no good explanation why the average deprivation of branching restrictions imposed on X encountered when Y was 0 should give a good estimate for the case when Y is 1. In the terms of computational statistics, one may say that there is really no good reason to assume that the learning variables are independently and identically distributed (i.i.d.). However, in practice it was found that these pseudo-costs are surprisingly accurate and they are widely used in state-of-the-art MIP solvers.

For this branching heuristic the variable is selected that has the lowest pseudo-costs and branch in the direction that is estimated to hurt the objective least first.

Best Pseudo-Cost Up (Pseudo Up):

With the same motivation as for most-fractional branching the variable with the lowest pseudo-costs is chosen, but this time the variable is always rounded up first.

Lowest Fractional Set Down (Non-Unary 0):

A new non-unary branching heuristic is introduced for set partitioning that is based on a construction heuristic for capacitated network design problems [42].

Inspired by this primal heuristic the following is proposed. The variables are selected in the order of lowest fractional LP value, up until the total sum of all fractional values is closest to 0.5. For example, say the variables with the lowest fractional values are X_7 with a current LP value of 0.05, X_1 with a value of 0.1, X_9 with 0.15, and X_3 with 0.9. Then X_7, X_1, X_9 would be selected as their sum is 0.3. Had X_3 been included the sum would have been 1.2 which has an absolute difference from 0.5 of 0.7 whereas the sum without X_3 is only 0.2 away. On the other hand, had X_3 had a fractional value of 0.3 it would have been included as the sum would now be 0.6 which is only 0.1 away from the desired value of 0.5.

Now, we split the search space by requiring that all variables equal 0, or that their sum is at least 1. We branch in the zero direction first. In the example above, the branching constraints added are $X_7, X_1, X_9 = 0$ first and on backtrack $X_7 + X_1 + X_9 \geq 1$. Note that both constraints are violated by the current LP relaxation.

Highest Fractional Set Up (Non-Unary 1):

A modification of the previous approach is also used, but this time the highest variables are focused on first. The variables are selected in order with the highest fractional LP values. For each the "missing fraction" is considered which is 1 minus the current LP value. Again these missing fractions are added until a sum that is closest to 0.5 is achieved. In this case, all variables are set to 1 first and on backtrack it is enforced that their sum is lower or equal the number of variables in the set minus 1.

For example, assume X_4 has a current LP value of 0.95, X_5 0.9, X_2 0.85, and X_3 0.1. Then, branching is done by enforcing $X_4, X_5, X_2 = 1$ first. Upon backtracking the constraint $X_4 + X_5 + X_2 <= 2$ is added.

9.3 Numerical Results

9.3.1 Implementation

The above heuristics is embedded in the state-of-the-art MIP solver Cplex Version 12.1. Note that only the branching strategy is modified through the implementation of a branch callback function. When compared with default Cplex an empty branch callback is used to ensure the comparability of the approaches.[1] The empty branch callback causes Cplex to compute the branching constraint using its internal default heuristic. None of the other Cplex behavior is changed, the system uses all the standard features like pre-solving, cutting planes, etc. Also, the search strategy, i.e., what open node to consider next, is left to Cplex. Note, however, that when Cplex dives in the tree, it considers the first node returned by the branch callback first so that, e.g., it does make a difference whether a variable is rounded up or down first.

[1]Note that Cplex switches off certain heuristics as soon as branch callbacks, even empty ones, are being used so that the entire search behavior is different.

Trace:

In this new approach, refered to as *Trace*, the branch callback works as follows. First, the features of the current subproblem are computed. This is done by adapting the new upper and lower bounds on variables as given by Cplex in an internal data structure which incrementally adjusts the 33 feature values. This is done for two reasons. First, it is difficult to efficiently get access to the internal pre-solved problem from Cplex. Secondly, due to cuts added and the non-unary branching constraints the resulting MIP will in general no longer be a pure set partitioning problem for which the features were defined. By using Cplex' bounds on the variables at least the solver benefits from any inferences that Cplex may have conducted, which may allow it to set variables to one of their bounds even if these variables were not branched on in the path from the root to the current node.

The branching is determined by using the normalized features to find the nearest cluster center and heuristic associated with that cluster is used. To learn a good assignment of heuristics to clusters offline GGA is employed. On the training set with 300 instances learning took 28 CPU days.

Trace is compared with the Cplex default as well as each of the pure heuristics (*pure* in the sense one heuristic is chosen and stuck with throughout the search), each of which is used to solve all the instances. This comparison by itself is what is commonly seen in operations research papers when it comes to determining the quality of branching heuristics.

Online Best Cluster Approach (OBCA):

We add two more approaches to our comparison. The first is the online best cluster approach (OBCA). OBCA determines offline which pure heuristic works best for each cluster. During the search, it determines which cluster is nearest to the current subproblem and uses the associated heuristic for branching. The difference to Trace is that the latter uses GGA to assign heuristics to clusters. Note that Trace may very well assign a heuristic to a cluster that is not the best when used throughout the entire search.

Naturally, OBCA's assignment of heuristics to clusters is highly biased by the instances in those clusters. Consequently, we consider a version of OBCA where we add subproblems to the set of instances which are encountered when solving the original training instances using the pure heuristics. This way, we might hope to give better search guidance for the subproblems that we will encounter during search. We refer to this version as OBCA+

9.3.2 ISAC

The second approach added to the comparison is the traditional ISAC. Here, the choice of a pure branching heuristic is considered as a Cplex parameter and ISAC is used to determine for which instance it should employ which pure search heuristic. Again, a version is considered where only the training instances are used to determine the clusters, as well as an ISAC+ where the subproblems encountered during the search of pure heuristic are also added to the training set.

9.3.3 Benchmark Instances

An important drawback of the research presented here (and algorithm tuning and solver portfolios in general) is that enough training instances are needed to allow the system to learn effectively. This is a hard pre-condition that must be met before the work presented here can be applied. Any benchmark set that consists of only a few dozen instances is not enough to allow any meaningful learning. In fact, it can be argued that benchmarks with a low number of instances (say, less than 100) can hardly be used to draw any conclusions that would generalize. However, the fact is that a lot of research is still based on such benchmark sets, and often it is even the case that approaches are designed and tested on the very same set of instances. From the prior research on algorithm tuning understand that results obtained in this way cannot be assumed to generalize to other instances.

To have access to a meaningful number of both training and test instances an instance generator is developed for set partitioning problems. Unlike most generators in the literature the generator is designed in such a way that it produces a highly heterogeneous set of benchmark instances. The generator first picks uniformly at random a number of bags between 200 and 2000, as well as a number of items between 20 and 50. It then flips three fair coins. The first coin determines whether the costs for each bag are chosen uniformly at random between 1 and 1,000 or whether this random number also gets multiplied by the number of items in each respective bag (making bags with more items more costly in general than bags with low numbers of items). The second coin determines whether, for the instance under construction, all sets will contain the same number of items or whether the number of items for each bag is determined individually by choosing a density uniformly at random between 10% and 30% of the total number of items. The last coin determines how we fill each bag. The generator either picks a subset of the desired size uniformly at random out of all such subsets, or it clusters the items that get added to bags by choosing a normal distribution around some item index and adding items in the proximity of that target item with higher probability than other items. Finally, to ensure feasibility, for each item the generator adds one bag which contains only that item at a high cost of 10^5.

Set partitioning instances were generated in this way and then default Cplex was run on each instance. To ensure instances of meaningful hardness, the first 500 instances for which Cplex needed at least 1,000 search nodes to solve the instance but took at most five minutes to solve were kept. These 500 instances were split into a training set with 300 instances and a test set with 200 instances. Note that this is a very modest training set. A learning approach like the one presented here will generally benefit a lot from larger training sets with at least 1,000 training instances, especially when the instances exhibit such a great diversity. The experiments were limited to a lower number of training instances for several reasons. First, it makes learning more challenging. Secondly, it is more realistic that only a limited number of training instances would be available (although again, we must assume there are more than a few dozen). Finally, ISAC+ and OBCA+ needed additional subproblems that were encountered while solving the original 300 training instances using the pure heuristics. Doing so resulted in 14 clusters with a total of over 1,900 instances which were used for training by ISAC+ and OBCA+ as well as to determine the clusters for Trace (recall, however, that for the latter the subproblem instances were not used for training by GGA).

9.3.4 Results

The experimental results are presented in Table 9.1. All experiments were run on Dell PowerEdge M610s, with 16 Xeon 2.4 CPUs and 24Gb of memory.

In the tables, apart from the usual average CPU time, standard deviation, median time, and number of instances solved (timeout was 300 seconds), also given is the Par10 score (a weighted average where unsolved instances are scored with 10 times the timeout) and the shifted geometric mean. The latter is the geometric mean of the runtimes plus 10 seconds and is used for benchmark sets where runtimes of individual instances can differ greatly. This causes the long running instances to dominate the average runtime comparison. The runtimes are shifted by ten seconds to prevent instances that are solved in extremely short amounts of time to greatly influence the mean – after all, in practice it is rarely important whether an instance is solved in 10 or 100 milliseconds.

Considering the pure heuristics first, in all measures it can be observed that on both training and test set the somewhat simplistic most fractional up and most fractional rounding heuristics fare best and even outperform Cplex' default heuristic. The table only presents the performance of the best pure heuristic, *Fractional (UP)*.

In the column "Oracle" the tables give the performance of the fastest pure heuristic for each individual instance. This is a natural limit for the ISAC approach that, at the very best, could choose the fastest pure heuristic for each instance.

As can be seen, there is significant room for improvement. Unsurprisingly, traditional ISAC is able to realize some of this potential on the training set, yet it does not generalize too well. On the test set it would have been better to just use the best heuristic that we had found on the training set. Adding subproblems to the training set in ISAC+ does not help either.

Considering OBCA, it is found that changing the branching heuristic during search is clearly beneficial. OBCA can reduce the runtime by over 15% compared to the Cplex default. Surprisingly, the performance of OBCA+ is much worse. Recall that some subproblems were added to the training set to allow OBCA+ to get a more realistic view into the problems where it would need to make a branching decision. Clearly, this does not work at all. If anything, OBCA was misled considering subproblems it was unlikely to see when the branching heuristic is changed during search.

Trace was invented to avoid exactly this problem. Recall that Trace only considers subproblem instances for clustering purposes, but bases its assignment of heuristics to clusters *solely* on the performance when running one of the original training instances *while* changing the branching heuristics in accordance to the heuristic/cluster assignment during search. As the experimental results show, Trace significantly reduces the runtime by over 20% when compared with the Cplex default. Note that the new branching heuristic is not directly embedded into the system and therefore cannot exploit branching heuristics like strong-branching which requires a tight integration with the solver. In light of this, an improvement by 20% over one of the most efficient Set Partitioning solvers is very encouraging and a proof of concept that dynamic branching strategies can be learned effectively, even on a relatively small heterogenous set of instances.

Test Set 300 instances	**Default**	**Fractional** (UP)	**Oracle**	**ISAC**	**ISAC+**	**OBCA**	**OBCA+**	**Trace**
Average (σ)	57.1 (63.2)	44.6 (53.2)	**38.0** (45.1)	41 (48.6)	42 (48.2)	41.1 (47.7)	50.9 (61.9))	38.4 (45.5)
Median	31.9	23.2	18.5	21.1	23.1	20.6	25.7	**17.6**
Min	0.04	0.04	0.02	0.02	0.32	0.04	0.06	0.04
Max	298	299	153	245	256	244	300	241
PAR 10 (σ)	57.1 (63.2)	44.6 (53.2)	**38.0** (45.1)	41 (48.6)	42 (48.2)	41.1 (47.7)	68.9 (247)	38.4 (45.4)
Shifted Geo Mean	43.2	36.2	**32.9**	34.2	35.2	34.7	39.1	33.2
Average # nodes (σ)	46K (61K)	30K (51K)	28K (49K)	30K (51K)	30K (51K)	24K (51K)	30K (51K)	**23K** (51K)
Nodes per second	**806**	673	736	732	715	584	590	599
Solved	300	300	300	300	300	300	298	300
Unsolved	0	0	0	0	0	0	2	0
% Solved	100	100	100	100	100	100	99.3	100

Test Set 200 instances	**Default**	**Fractional** (UP)	**Oracle**	**ISAC**	**ISAC+**	**OBCA**	**OBCA+**	**Trace**
Average (σ)	46.4 (52.7)	42.2 (51.8)	**35.0** (40.7)	42.4 (50.7)	42.3 (50.4)	38.7 (45.3)	52.9 (61.7)	35.3 (40.3)
Median	24.2	20.4	**18.0**	21.1	21	18.8	28.9	18.9
Min	0.06	0.04	0.04	0.04	0.04	0.04	0.06	0.04
Max	254	267	201	267	267	227	300	183
PAR 10 (σ)	46.4 (52.7)	42.2 (51.8)	**35.0** (40.7)	42.4 (50.7)	42.3 (50.4)	38.7 (45.3)	66.4 (216)	35.3 (40.3)
Shifted Geo. Mean	38.1	35.2	**32.0**	35.6	35.6	33.9	41	32.6
Average # nodes (σ)	44K (69K)	30K (64K)	29K (63K)	30K (63K)	30K (63K)	24K (63K)	30K (63K)	**22K** (64K)
Nodes per second	**949**	711	828	708	710	620	567	623
Solved	200	200	200	200	200	200	199	200
Unsolved	0	0	0	0	0	0	1	0
% Solved	100	100	100	100	100	100	99.5	100

Table 9.1: Training and Testing Results. All times are CPU times in seconds. Timeout was 300 seconds.

At the same time, Trace works very robustly. Its standard deviations in runtime are lower that those of any pure branching heuristic, and the spread of runtimes (min to max) is also greatly reduced. None of the instances are solved barely within the allowed timeout which cannot be said for any pure heuristic.

On the other hand, it can be seen that changing heuristics during search imposes a noticeable cost – the number of nodes per second is clearly less due to the costly re-computations of the subproblem features. This is outweighed by a very significant reduction in choice points, though: Trace consistently visits only about 50% of the number of nodes of the Cplex default.

9.4 Chapter Summary

This chapter introduced the idea to use an offline algorithm tuning tool to learn an assignment of branching heuristics to training instance clusters which is used dynamically during the search to determine a preferable branching heuristic for each subproblem encountered during search. This approach, named Trace, was evaluated on a set of highly diverse set partitioning instances. We found that the approach clearly outperforms the Cplex default and also the best pure branching heuristic considered here. While not limited by it, it comes very close to choosing the performance of an oracle that magically tells us which pure branching heuristic to use for each individual instance.

The chapter concluded that mixing branching heuristics can be very beneficial, yet care must be taken when learning when to choose which heuristics, as early branching decisions determine the distribution of instances that must be dealt with deeper in the tree. This problem was solved by using the offline algorithm tuning tool GGA to determine a favorable synchronous assignment of heuristics to clusters so that instances can be solved most efficiently.

The chapter's approach requires a reasonable amount of training instances, as well as a number of branching heuristics, and of course meaningful features that can characterize subproblems during search. For practitioners, who actually need their problems to be solved repeatedly, access to a good number of training instances is less of a problem as it poses for academics. As things stand, the main obstacle of applying Trace to other problems is therefore the definition of good problem features.

Chapter Ten

Conclusion

This dissertation introduces the new methodology of instance-specific algorithm configuration or ISAC. Although there has recently been a surge of research in the area of automatic algorithm configuration, ISAC enhances the existing work by merging the strengths of two powerful techniques: instance-oblivious tuning and instance-specific regression. When used in isolation, these two methodologies have major drawbacks. Existing instance-oblivious parameter tuners assume that there is a single parameter set that will provide optimal performance over all instances, an assumption that is not provably true for NP-hard problems. Instance-specific regression, on the other hand, depends on accurately fitting a model to map from features to a parameter, which is a challenging task requiring a lot of training data when the features and parameters have non-linear interactions. ISAC resolves these issues by relying on machine learning techniques.

This approach has been shown to be beneficial on a variety of problem types and solvers. This dissertation has presented a number of possible configurations and consistently expanded the possible applications of ISAC. The main idea behind this methodology is the assumption that although solvers have varied performance on different instances, instances that are similar in structure result in similar performance for a particular solver. The objective then becomes to identify these clusters of similar instances and then tune a solver for each cluster. To find such clusters, ISAC identifies each instance as a vector of descriptive features. When a new instance needs to be solved, its computed features are used to assign it to a cluster, which in turn determines the parameterization of the solver used to solve it. Based on this methodology, the thesis began by using a parameterized solver, a normalized vector of all the supplied instance features, g-means [37] for clustering, and GGA [4] for training. This was shown to be highly effective for Set Covering Problems, Mixed Integer Problems, and Satisfiability Problems.

The approach was then extended to be applicable to portfolios of solvers by changing the training methodology. There are oftentimes many solvers that can be tuned, so instead of choosing and relying on only tuning a single parameterized solver, it was shown how to create a meta solver whose parameters could determine not only which solver should be employed but also the best parameters for that solver. When applied to Satisfiability problems, this portfolio based approach was empirically shown to outperform the existing state-of-the-art regression based algorithm portfolio solvers like SATzilla [108] and Hydra [106].

The dissertation then showed how ISAC can be trained dynamically for each new test instance by changing the clustering methodology to k-nearest-neighbor and further improved by training sequential schedules of solvers in a portfolio. Although similar to the previously existing constraint satisfaction solver CPHydra [72], this thesis showed how to use particular integer programming and column generation to create a more efficient scheduling algorithm which was able to efficiently handle over 60 solvers and hundreds of instances in an online setting. This last implementation was the basis of a the 3S SAT solver that won 7 medals in the SAT 2011 Competition.

ISAC was then further expanded in three orthogonal ways. First, the dissertation showed how to expand the methodology behind 3S to create a parallel schedule of solvers dynamically. The thesis compared the resulting portfolio, with the current state-of-the-art parallel solvers on instances from all SAT categories and showed that by creating these parallel schedules marks a very significant improvement in the ability to solve SAT instances. The new portfolio generator was then used to generate a parallel portfolio for

application instances based on the latest parallel and sequential SAT solvers available. This portfolio is currently participating in the 2012 SAT Challenge.

Next, the dissertation showed how to effectively identify and filter unwanted features. Having good, descriptive features is paramount for ISAC to achieve significant improvements. However, most standard filtering techniques are impractical since iteratively trying different feature sets would require retraining the solvers multiple times, which can often take an impractical amount of time. To resolve this issue, this thesis presented three efficient cluster evaluation criteria that avoid the costly tuning step. The result then showed in the case of a feature set like the one for constraint satisfaction problems, filtering allowed to find clusters that further improved the performance that could be achieved by ISAC. Even in well studied feature set like the one available for SAT, can be improved through filtering and result in better cluster for ISAC.

Finally, this dissertation showed how the ISAC methodology can be used to create an adaptive tree search based solver that dynamically chooses the branching heuristic that is most appropriate for the current sub-problem it observes. Here, the thesis showed that in many cases in optimization when performing a complete search, each subtree is still a problem of the same type as the original but with a slightly different structure. By identifying how this structure changes during search, the solver can dynamically change its guiding heuristics to best accommodate the new sub tree. Tested on the set partitioning problem, this adaptive technique was shown to be highly effective.

In its entirety, the dissertation showed that ISAC is a highly configurable and effective methodology, demonstrating that it is possible to train a solver or algorithm automatically for enhanced performance while requiring minimal expertise and involvement on the part of the user.

Having laid out the groundwork, there are a number of future directions that can be pursued to push the state-of-the-art further. One such direction involves a deeper analysis of the base assumption that instances with similar features tend to yield to the same algorithm. One way to do this is by creating instances that have a specific feature vector. This way arbitrarily tight clusters can be generated and trained on automatically. An additional benefit to this line of research will be the ability to expand the small existing benchmarks. As was noted in this dissertation certain problem types have limited number of instances that cannot be used for training effectively. This is the case for the set partitioning problems, the standard MIP benchmark, and the industrial SAT instances.

Furthermore, by being able to generate instances with a specific feature vector automatically, the entirety of the problem space can be explored. Such an overview of the search space can give insight into how smooth the transitions are between two clusters, how wide the clusters are, and numerous are the hard or easy clusters. Additionally, solvers won't need to created for the general case, but instead research can be focused on each separate cluster where state-of-the-art solvers will be struggling.

Exploring the entire problem space has an added bonus of identifying clusters of easier and harder instances. Such knowledge can then be exploited by studying instance perturbation techniques. In such a scenario, when an instance is identified as belonging to a hard cluster, it might be possible to apply some efficient modifications to the instance, changing its features and thereby shifting it into an easier cluster. Observing these areas of easier and harder instances in the problem space would also allow for a better understanding of

the type of structure that leads to difficult problems.

Alternatively, it would be interesting to explore the relations between problem types. It is well known that any NP complete problem can be reduced to another NP complete problem in polynomial time. This fact could be used to avoid coming up with new features for each new problem type. Instead it might be enough to convert the instance to SAT and use the 48 well established features. Some preliminary testing with cryptography instances suggest that regardless of the conversion technique used to get the SAT instance, the resulting clusters are usually very similar. Additionally for CP problems, a domain where a vector of features has already been proposed, converting the problems to SAT results in similar clusters regardless of whether the CP or the SAT features are employed.

Transitions to SAT are always for NP complete problems but not always easy or straightforward, so additional research needs to be done in automating the feature generation. This can be done by converting the problem into black box optimization (BBO). These problems arise in numerous applications, especially in scientific and engineering contexts in problems that are too incomplete to develop effective problem specific heuristics. So to feature computation can be done through sampling of the instance's solution space to get a sense of the search terrain. The question that will need to be answered is how much critical information when converting a problem like SAT into a BBO in order to compute its features.

Alternatively to generating features automatically, additional work can be done in terms of filtering. This thesis has highlighted the importance of having a good feature set and its impact on ISACs performance, But it would be interesting to tune not only the solver, but also the clustering algorithm and feature set simultaneously.

Further research should also be done for the instance-oblivious tuning. In the experiments presented in this dissertation, GGA performed well, but in all of the tuned experiments had a short cutoff time - under twenty minutes. Tuning problems where each instance can take more time or that have more instances becomes computationally infeasible. For example, simulations are frequently relied on for disaster recovery, sustainability research, etc., and tuning these simulation algorithms to work quickly and efficiently would be of immense immediate benefit. These simulations, however, tend to run for extended periods of time in order to get accurate results. It is therefore important to investigate new techniques that can find useable parameterizations within a reasonable timeframe.

In summary, this dissertation lays out the groundwork for a highly configurable and effective instance-specific algorithm configuration methodology and hopefully further research will enhance and expand its applicability.

Bibliography

[1] Cpai08 competition. `http://www.cril.univ-artois.fr/CPAI08/`.

[2] Sat competition. `http://www.satcomptition.org`.

[3] Belarmino Adenso-Diaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. In *Operations Research*, volume 54, pages 99–114, 2006.

[4] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *CP*, pages 142–157, 2009.

[5] G. Audemard and L. Simon. Glucose: a solver that predicts learnt clauses quality. SAT Competition, 2009.

[6] Charles Audet and Dominique Orban. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization*, 16(3):642, 2006.

[7] A. Balint, M. Henn, and O. Gableske. hybridgm. solver description. SAT Competition, 2009.

[8] Roberto Battiti, Giampietro Tecchiolli, Istituto Nazionale, and Fisica Nucleare. The reactive tabu search. In *INFORMS Journal on Computing*, volume 6, pages 126–140, 1993.

[9] Pavel Berkhin. Survey of clustering data mining techniques. 2002.

[10] A. Biere. Picosat version 846. solver description. SAT Competition, 2007.

[11] A. Biere. P{re,i}cosatsc'09. SAT Competition, 2009.

[12] A. Biere. Lingeling. SAT Race, 2010.

[13] A. Biere. Plingeling. SAT Race, 2010.

[14] A. Biere. Lingeling and friends at the sat competition 2011. Technical report, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[15] Mauro Birattari. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers, 2002.

[16] Mauro Birattari, Thomas Stutzle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, 2002.

[17] Justin Boyan, Andrew W. Moore, and Pack Kaelbling. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1, 2000.

[18] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 201–226:201–226, 2005.

[19] D. R. Bregman. The sat solver mxc, version 0.99. SAT Competition, 2009.

[20] D. R. Bregman and D. G. Mitchell. The sat solver mxc, version 0.75. solver description. SAT Race, 2008.

[21] Leo Breiman and Leo Breiman. Bagging predictors. In *Machine Learning*, pages 123–140, 1996.

[22] Alberto Caprara, Matteo Fischetti, Paolo Toth, Daniele Vigo, and Pier Luigi Guida. Algorithms for railway crew management, 1997.

[23] Steven P. Coy, Bruce L. Golden, George C. Runger, and Edward A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7:77–97, 2001.

[24] G.B. Dantzig and P. Wolfe. The decomposition algorithm for linear programs. *Econometrica*, 29(4):767–778, 1961.

[25] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, 1962.

[26] G. Dequen and O. Dubois. kcnfs. solver description. SAT Competition, 2007.

[27] Niklas Een and Niklas Sörensson. Minisat. `http://minisat.se`, 2010.

[28] S. L. Epstein, E. C. Freuder, R. J. Wallace, A. Morozov, and B. Samuels. The adaptive constraint engine. *CP*, pages 525–542, 2002.

[29] Alex S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.

[30] Matteo Gagliolo and Jürgen Schmidhuber. Dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47:3–4, 2006.

[31] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Martin gebser , benjamin kaufmann , torsten schaub. *CPAIOR*, 2009.

[32] Ian P. Gent, Holger H. Hoos, Patrick Prosser, and Toby Walsh. Morphing: Combining structure and randomness, 1999.

[33] Ian P. Gent, Holger H. Hoos, Patrick Prosser, and Toby Walsh. Morphing: Combining structure and randomness. *AAAI*, 9:849–859, 1999.

[34] Carla P. Gomes and Bart Selman. Problem structure in the presence of perturbations. *AAAI*, pages 221–226, 1997.

[35] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.

[36] Y. Hamadi, S. Jabbour, and L. Sais. Lysat: solver description. SAT Competition, 2009.

[37] Greg Hamerly and Charles Elkan. Learning the k in k-means. In *In Neural Information Processing Systems*. MIT Press, 2003.

[38] M. Heule and H. van Marren. march hi: solver description. SAT Competition, 2009.

[39] M. Heule and H. van Marren. march nn. `http://www.st.ewi.tudelft.nl/sat/download.php`, 2009.

[40] Marijn Heule, Mark Dufour, Joris Van Zwieten, and Hans Van Maaren. March eq: implementing additional reasoning into an efficient lookahead sat solver. *Theory and Applications of Satisfiability Testing*, 3542:345–359, 2004.

[41] Karla L. Hoffman and Manfred Padberg. Solving airline crew scheduling problems by branch-and-cut. In *Management Science*, volume 39, pages 657–682, 1993.

[42] K. Holmberg and D. Yuan. Lagrangean heuristic based branch-and-bound approach for the capacitated network design problem. *Operations Research*, 48:461–481, 2000.

[43] H.H. Hoos. Adaptive novelty+: Novelty+ with adaptive noise. *AAAI*, 2002.

[44] Efthymios Housos and Tony Elmroth. Automatic optimization of subproblems in scheduling airline crews. In *Interfaces*, volume 27, pages 68–77, 1997.

[45] B. A. Huberman, R. M. Lukose, and T. Hogg. An economic approach to hard computational problems. *Science*, 27:51–53, 1997.

[46] Frank Hutter and Youssef Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. TechReport MSR-TR-2005-125, Microsoft Research, 2005.

[47] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP*, pages 213–228, 2006.

[48] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin Murphy. Time-bounded sequential parameter optimization. In *LION*, pages 281–298, 2010.

[49] Frank Hutter, Holger H. Hoos, Kevin Leyton-brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *JAIR*, 36:267–306, 2009.

[50] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Rsaps: Reactive scaling and probabilistic smoothing. In *CP*, 2002.

[51] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *CP*, pages 233–248, 2002.

[52] IBM. Reference manual and user manual. v12.1, 2009.

[53] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac – instance-specific algorithm configuration. *ECAI*, pages 751–756, 2010.

[54] Serdar Kadioglu and Meinolf Sellmann. Dialectic search. *CP*, 2009.

[55] Ashiqur R. Khudabukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-brown. Satenstein: Automatically building local search sat solvers from components. In *IJCAI*, 2009.

[56] D. H. Leventhal and M. Sellmann. The accuracy of search heuristics: An empirical study on knapsack problems. *CPAIOR*, pages 142–157, 2008.

[57] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim Mcfadden, and Yoav Shoham. Boosting as a metaphor for algorithm design. In *CP*, pages 899–903, 2003.

[58] Kevin Leyton-brown, Eugene Nudelman, Galen Andrew, Jim Mcfadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *IJCAI*, pages 1542–1543, 2003.

[59] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM*, pages 66–76, 2000.

[60] C.M. Li and W.Q. Huang. G2wsat: Gradient-based greedy walksat. In *SAT*, volume 3569, pages 158–172, 2005.

[61] C.M. Li and W. We. Combining adaptive noise and promising decreasing variables in local search for sat. solver description. SAT Competition, 2009.

[62] Xiaoming Li, María Jesús Garzarán, and David Padua. Optimizing sorting with genetic algorithms. *In the International Symposium on Code Generation and Optimization*, pages 99–110, 2005.

[63] Stuart P. Lloyd. Least squares quantization in pcm. *Transactions on Information Theory*, 28:129–137, 1982.

[64] P. Janici M. Nikolic, F. Maric. Instance based selection of policies for sat solvers. *Theory and Applications of Satisfiability Testing*, pages 326–340, 2009.

[65] Yuri Malitsky and Meinolf Sellmann. Stochastic offline programming. In *ICTAI*, 2009.

[66] David McAllester, Bart Selman, and Henry A. Kautz. Evidence for invariants in local search. *AAAI*, pages 321–326, 1997.

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

[68] Mitsuo Motoki and Ryuhei Uehara. Unique solution instance generation for the 3-satisfiability (3sat) problem. pages 293–305, 2000.

[69] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[70] Nysret Musliu. Local search algorithm for unicost set covering problem. *Advances in Applied Artificial Intelligence*, pages 302–311, 2006.

[71] Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[72] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[73] Donald J. Patterson and Henry Kautz. Auto-walksat: A self-tuning implementation of walksat. In *Electronic Notes in Discrete Mathematics*, volume 9, pages 360–368, 2001.

[74] M.P. Petrik and S. Zilberstein. Learning static parallel portfolios of algorithms. *International Symposium on Artificial Intelligence and Mathematics*, 2006.

[75] D.N. Pham and Anbulagan. ranov. solver description. SAT Competition, 2007.

[76] D.N. Pham and C. Gretton. gnovelty+. solver description. SAT Competition, 2007.

[77] D.N. Pham and C. Gretton. gnovelty+ (v.2). solver description. SAT Competition, 2009.

[78] S. Prestwich. Vw: Variable weighting scheme. *SAT*, 2005.

[79] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[80] O. Roussel. Description of ppfolio, 2011. `http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf`.

[81] H. Samulowitz and R. Memisevic. Learning to solve qbf. *AAAI*, 2007.

[82] A. Saxena. Mip benchmark instances. `http://www.andrew.cmu.edu/user/anureets/mpsInstances.htm`.

[83] Meinolf Sellmann. Disco - novo - gogo: Integrating local search and complete search with restarts. *AAAI*, pages 1051–1056, 2006.

[84] Bart Selman and Henry Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. *IJCAI-*, pages 290–295, 1993.

[85] Bryan Silverthorn and Risto Miikkulainen. Latent class models for algorithm portfolio methods. *AAAI*, 2010.

[86] A. Slater. Modelling more realistic sat problems. In *Australian Joint Conference on Artificial Intelligence*, pages 291–602, 2002.

[87] Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1), 2008.

[88] M. Soos. Cryptominisat 2.5.0. solver description. SAT Race, 2010.

[89] M. Soos. Cryptominisat 2.9.0, 2011.

[90] N. Sorensson and N. Een. Minisat 2.2.0. `http://minisat.se`, 2010.

[91] D. Stern, H. Samulowitz, R. Herbrich, T. Graepel, L. Pulina, and A. Tacchella. Collaborative expert portfolio management. *AAAI*, 2010.

[92] M. Streeter and S. F. Smith. New techniques for algorithm portfolio design. *Robotics*, 10, 2008.

[93] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *AAAI*, pages 1197–1203, 2007.

[94] Hugo Terashima-Marín and Peter Ross. Evolution of constraint satisfaction strategies in examination timetabling. *GECCO*, pages 635–642, 1999.

[95] John Thornton, , John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira. Additive versus multiplicative clause weighting for sat. In *AAAI*, pages 191–196, 2004.

[96] D.A.D Tompkins, F. Hutter, and H. H. Hoos. saps. solver description. SAT Competition, 2007.

[97] C. Toregas, R. Swain, C. ReVelle, and L. Bergman. The location of emergency service facilities. *Operational Research*, pages 1363–1373, 1971.

[98] T. Uchida and O. Watanabe. Hard sat instance generation based on the factorization problem. `http://www.is.titech.ac.jp/~watanabe/gensat/a2/index.html`, 2010.

[99] F. J. Vasko, F. E. Wolf, and K. L. Stott. Optimal selection of ingot sizes via set covering. In *Operations Research*, volume 35, pages 346–353, 1987.

[100] W. Wei and C. M. Li. Switching between two adaptive noise mechanisms in local search for sat. solver description. SAT Competition, 2009.

[101] W. Wei, C. M. Li, and H. Zhang. adaptg2wsatp. solver description. SAT Competition, 2007.

[102] W. Wei, C. M. Li, and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for sat. solver description. SAT Competition, 2007.

[103] W. Wei, C. M. Li, and H. Zhang. Deterministic and random selection of variables in local search for sat. solver description. SAT Competition, 2007.

[104] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation*, 1(1):67–82, 1997.

[105] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla2009: an automatic algorithm portfolio for sat. solver description. SAT Competition, 2009.

[106] Lin Xu, Holger H. Hoos, and Kevin Leyton-brown. Hydra: Automatically configuring algorithms for portfolio-based selection. *AAAI*, 2010.

[107] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-brown. Satzilla-07: The design and analysis of an algorithm portfolio for sat. *CP*, pages 712–727, 2007.

[108] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-brown. Satzilla: Portfolio-based algorithm selection for sat. In *Journal Of Artificial Intelligence Research*, volume 32, pages 565–606, 2008.