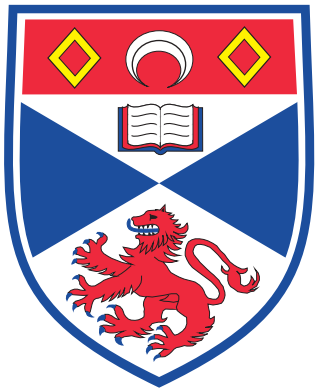
On Algorithm Selection, with an Application to Combinatorial Search Problems

Lars Kotthoff



This thesis is submitted in partial fulfilment for the degree of PhD at the University of St Andrews.

15th December 2011

Abstract

The Algorithm Selection Problem is to select the most appropriate way for solving a problem given a choice of different ways. Some of the most prominent and successful applications come from Artificial Intelligence and in particular combinatorial search problems. Machine Learning has established itself as the de facto way of tackling the Algorithm Selection Problem. Yet even after a decade of intensive research, there are no established guidelines as to what kind of Machine Learning to use and how.

This dissertation presents an overview of the field of Algorithm Selection and associated research and highlights the fundamental questions left open and problems facing practitioners. In a series of case studies, it underlines the difficulty of doing Algorithm Selection in practice and tackles issues related to this. The case studies apply Algorithm Selection techniques to new problem domains and show how to achieve significant performance improvements. Lazy learning in constraint solving and the implementation of the `alldifferent` constraint are the areas in which we improve on the performance of current state of the art systems. The case studies furthermore provide empirical evidence for the effectiveness of using the misclassification penalty as an input to Machine Learning.

After having established the difficulty, we present an effective technique for reducing it. Machine Learning ensembles are a way of reducing the background knowledge and experimentation required from the researcher while increasing the robustness of the system. Ensembles do not only decrease the difficulty, but can also increase the performance of Algorithm Selection systems. They are used to much the same ends in Machine Learning itself.

We finally tackle one of the great remaining challenges of Algorithm Selection – which Machine Learning technique to use in practice. Through a large-scale empirical evaluation on diverse data taken from Algorithm Selection applications in the literature, we establish recommendations for Machine Learning algorithms that are likely to perform well in Algorithm Selection for combinatorial search problems. The recommendations are based on strong empirical evidence and additional statistical simulations.

The research presented in this dissertation significantly reduces the knowledge threshold for researchers who want to perform Algorithm Selection in practice. It makes major contributions to the field of Algorithm Selection by investigating fundamental issues that have been largely ignored by the research community so far.

Candidate's declaration

I, Lars Kotthoff, hereby certify that this thesis, which is approximately 41,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in September 2008 and as a candidate for the degree of PhD in September 2008; the higher study for which this is a record was carried out in the University of St Andrews between 2008 and 2011.

15th December 2011, signature of candidate

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

15th December 2011, signature of supervisor

Permission for electronic publication

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of this thesis:

Access to printed copy and electronic publication of thesis through the University of St Andrews.

15th December 2011, signature of candidate

15th December 2011, signature of supervisor

Acknowledgements

I would like to thank my supervisors, Ian Miguel and Ian Gent, for their support during the course of my PhD. Their feedback on my work was especially helpful during the beginning of my studies. Derek Long provided feedback on my work in his capacity as my SICSA supervisor and Christian Schulte as my mentor at my first Constraint Programming conference.

I would also like to thank my colleagues and collaborators; in alphabetical order, Andrea Rendl, Andy Grayland, Chris Jefferson, Dharini Balasubramaniam, Karen Petrie, Lakshitha de Silva, Neil Moore, Özgür Akgün, Pete Nightingale and Tom Kelsey. Thanks go to the administrative and technical support team and especially Angie Miguel and John McDermott.

I have had the opportunity to attend many conferences and workshops and meet many people there. Some of them influenced my research. Kristian Kersting pointed out SVM^{struct} to me. Holger Hoos, Kevin Leyton-Brown and Lin Xu discovered and helped debug a mistake in a previous version of the evaluation of the performance of SATzilla in Chapter 7. Jesse Hoey, James Cussens and Alan Frisch gave useful feedback on some Machine Learning aspects.

Many reviewers evaluated and commented on my work at various venues. I thank them for their feedback and hope that I will be able to return to them what they have given to me.

Finally, I would like to thank everybody else at St Andrews and who I met at conferences and workshops who improved my overall PhD experience. This goes especially to everybody who I forgot to mention by name. Thanks to CIRCA for all the cake.

My thesis was examined by Mark-Jan Nederhof and Ken Brown. They let me off in time for lunch (and helped to improve this document as well).

My studies were supported by a prize studentship from the Scottish Informatics and Computer Science Alliance (SICSA) and by EPSRC grant EP/H004092/1. For the research presented in this dissertation, I received additional support from Amazon Web Services, the School of Computer Science at the University of St Andrews, the Association for Constraint Programming, the International Symposium on Combinatorial Search and the Australian National University. Without this support, this PhD would not have happened.

Lucy Miguel called me a silly monkey and she is probably right.

Für die, die ich nicht mehr erlebten.

Contents

List of Figures

List of Tables

Publications

The contents of this thesis have appeared in large parts in the following publications. The contributions of the author of this thesis to jointly authored publications are listed below the respective publication.

Lars Kotthoff. Constraint solvers: An empirical evaluation of design decisions. CIRCA preprint 2009/7, University of St Andrews, Centre for Interdisciplinary Research in Computational Algebra (CIRCA), 2009. URL http://www-circa.mcs.st-and.ac.uk/Preprints/solver-design.pdf.

Lars Kotthoff. Dominion – A constraint solver generator. In *Doctoral Program of CP*, September 2009.

Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, Neil Moore, Peter Nightingale, and Karen E. Petrie. Learning When to Use Lazy Learning in Constraint Solving. In *19th European Conference on Artificial Intelligence*, pages 873–878, August 2010.

The contributions to this paper were,

- the experimental evaluation of the two candidate solvers on the problem instances,
- the specification and extraction of features,
- the post-processing of the experimental data,
- the methodology for learning a decision tree,
- the methodology for refining the decision tree and
- the evaluation of the performance and generality of the decision tree.

Lars Kotthoff, Ian P. Gent, and Ian Miguel. Using machine learning to make constraint solver implementation decisions. In *SICSA PhD conference*, 2010.

Ian P. Gent, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Machine learning for constraint solver design -– A case study for the alldifferent constraint. In *3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS)*, pages 13–25, 2010.

The contributions to these papers were,

- the experimental evaluation of the candidate algorithms on the problems,

- the specification and extraction of features,
- the post-processing of the experimental data,
- the Machine Learning methodology and
- the evaluation of the performance of the learned classifiers.

Lars Kotthoff, Ian Miguel, and Peter Nightingale. Ensemble classification for constraint solver configuration. In *16th International Conference on Principles and Practices of Constraint Programming*, pages 321–329, September 2010.

The contributions to this paper were,

- the idea of applying ensemble classification to Algorithm Selection,
- the experimental evaluation of the candidate algorithms on the problems,
- the specification and extraction of features,
- the post-processing of experimental data,
- the Machine Learning methodology and
- the evaluation of the performance of the learned classifiers and the ensemble.

Lars Kotthoff, Ian P. Gent, and Ian Miguel. A Preliminary Evaluation of Machine Learning in Algorithm Selection for Search Problems. In *Fourth Annual Symposium on Combinatorial Search*, pages 84–91, July 2011.

The contributions this papers were,

- the idea of performing the comparison,
- the idea of applying statistical relational learning,
- the selection of data sets and systems to compare with,
- the experimental evaluation of the Machine Learning algorithms on the problems,
- the Machine Learning methodology,
- the post-processing of the experimental data,
- the evaluation and analysis of the experimental data and
- establishing the recommendations as to which Machine Learning algorithms to use.

Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Specification of the Dominion Input Language Version 0.1. Technical Report, University of St Andrews, 2009. URL http://www-circa.mcs.st-and.ac.uk/Preprints/InLangSpec.pdf.

The contributions to this paper were,

- participation in the design of the language and
- the implementation of a parser and type checker for the language.

Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, and Ian Miguel. Modelling Constraint Solver Architecture Design as a Constraint Problem. In *Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, April 2011.

The contributions to this paper were,

- the encoding of the problem as a constraint problem,
- the automatic conversion into a Minion input file and
- the mapping of the solution to solver components.

Dharini Balasubramaniam, Lakshitha de Silva, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Dominion: An Architecture-driven Approach to Generating Efficient Constraint Solvers. In *9th Working IEEE/IFIP Conference on Software Architecture*, June 2011.

The contributions to this paper were,

- participation in the design of the system,
- the implementation of parts of the system and
- the description of related work and parts of the system.

1

Introduction

A large part of daily life involves making decisions. What should I eat? How should I eat? Where should we have lunch? The choice can be easy if there is only one option or a clear preference, but many decisions are difficult. Computer Science is no different in this respect. Be it the choice of a theorem to apply to a problem, the design of an algorithm to solve the problem or the actual implementation of the chosen algorithm – there are always choices to make.

In Computer Science, "Algorithm Selection" is an umbrella term for decisions like this. The Algorithm Selection Problem has been known and formally described for decades (Rice, 1976). Given are a set of algorithms and a set of problems. Each algorithm can be applied to solving each of the problems, but the algorithms exhibit different behaviour when solving a problem. The task is, for each problem, to select the algorithm with the best behaviour on that particular problem.

As an example, consider selecting a way to sort a list of numbers. There are a number of well-established algorithms available, such as Quicksort, Heapsort and Bubblesort. In terms of computational complexity, Heapsort would be preferred over Quicksort, but in practice Quicksort often performs better. For a specific list to sort, the choice of algorithm depends on various factors such as to what extent the list is sorted already, whether the list can be sorted in place and whether the elements of the list can be accessed at random. To select a sorting algorithm for a specific list, all these factors would need to be considered.

The most frequently used metric for assessing the utility of an algorithm on a problem is the time the algorithm requires to solve the problem. In this formulation, the Algorithm Selection Problem becomes the problem of minimising the total time required to solve all problems (e.g. Gomes and Selman (2001)). There are other variants – instead of considering the solution time, the quality of a solution obtained (e.g. Soares et al. (2004)), the progress towards finding a solution or other resource requirements, such as memory (a criteria in e.g. Smith and Setliff (1992)), can be considered.

In some cases, solving the Algorithm Selection Problem can be easy. One of the algorithms from the set to choose from may be superior to all the other ones in every case. The factors that affect the performance of an algorithm on a problem may be sufficiently well known and understood that the best algorithm for a given problem can be determined easily. A human expert may have such a deep understanding of

the algorithms and problems that she can pick the best algorithm every time.

In practice, Algorithm Selection is hard in all but the most trivial cases. Rice (1976) notes on this topic that,

> “We conclude that most realistic algorithm selection problems are [...] quite complex.”

For most applications, the choice to make is not obvious even for domain experts. This may be due to any number of reasons – the algorithms or the problems may be poorly understood, there may be unpredictable factors that have a significant effect on the performance or the previously observed performance may exhibit such large variations that an estimate of future performance cannot be made.

After decades of almost no research into it, the Algorithm Selection Problem has recently attracted a lot of interest (e.g. Gomes and Selman (2001), Beck and Freuder (2004), Xu et al. (2008)). The use of Machine Learning to solve this problem is a current research topic (e.g. Gebruers et al. (2004), O’Mahony et al. (2008), Silverthorn and Miikkulainen (2010)). This can be attributed to two main factors.

1. The performance of computers is nowadays so high that we can easily afford to have them spend some of their time on “clerical” work. For most problems, the performance bottleneck has shifted from the computer to the human – humans spend more time formulating a problem in a way that the computer can work with it than the computer requires to solve the problem. This is not least witnessed by a vast amount of Artificial Intelligence research that enables us to solve even complex problems within seconds (e.g. Gent et al. (2006b)).

2. The number of different algorithms or systems available for solving a given problem is in general large. Most of these algorithms are complementary such that the performance of one dominates the performance of another only on a limited set of problems. There are no easy ways to assess the performance of any of these algorithms on a problem that has not been solved before.

A good example for illustrating both points is the Milepost GCC Project (Fursin et al., 2011), which optimises compiled code using Machine Learning techniques. The GNU Compiler Collection (GCC) is one of the most complex pieces of software ever written, with substantial effort invested into making the generated machine code more efficient. Not least the dozens of different compiler flags that control optimisation bear witness to this. Choosing the flags that give the best performance is something of a dark art, even with the provided predefined levels of optimisation that control groups of individual optimisation settings.

The compilation process of a programme can be a lengthy one, especially for C++ code. In that respect, Milepost GCC is a counter-intuitive effort, because it even increases the compilation time, in most cases disproportionally to the achieved improvement in performance of the compiled code. The performance of modern hardware however makes such an additional investment affordable. At the same time,

the ubiquitousness of computers in everyday life means that the piece of code that more effort was spent on compiling will probably be run thousands, if not millions of times such that even a small improvement in performance of the compiled code is well worth the additional effort.

Despite decades of manual tuning and optimisation, Milepost GCC was able to improve the efficiency of the generated code substantially by using Machine Learning techniques. One of the main factors in its success was the complexity of the underlying problem that made it infeasible for humans to tackle in the same way that Machine Learning did. It is however possible to explore efficiently the very large space of possible optimisations automatically on modern hardware.

Research a few decades ago focused on building more sophisticated systems to deliver ever improving performance. Today, we are faced with a situation where the probability is high that the system already exists that can solve a given problem efficiently, but choosing it from the plethora of those available is hard. This is especially true in many fields of Artificial Intelligence that deal with computationally hard problems. The difference between making the right and the wrong choice can be the difference between solving the problem in a matter of seconds or requiring more time than the age of the universe.

Two areas of Artificial Intelligence that illustrate this point especially well are Boolean satisfiability and constraint programming. In both areas, decades of research have facilitated the creation of many different systems for solving these kinds of problems. In addition to the difficulty of choosing which system to use, there are often parameters that the user can tune to affect the performance. With the right parameter setting, it is possible that a system that previously exhibited poor performance becomes the top performer. In practice, the user base of a system often does not extend far beyond the people who are involved with its development and know it in detail.

Even experts in such systems often struggle to make the right choice. Especially when confronted with a problem that seems to be unlike any problem ever seen before, one is often reduced to merely making a guess as to the system with the best performance. Researchers who want to apply techniques from an area that they have no background in to their problems are in an even worse position.

Recent efforts have used Machine Learning to tackle the problem of selecting the system with the best performance. Especially in Artificial Intelligence, this methodology has achieved great success. One of the most famous systems that use Machine Learning to perform Algorithm Selection is SATzilla (Xu et al., 2008), which selects from a portfolio of satisfiability solvers. The problem of proving or disproving whether a Boolean formulae is satisfiable is the first problem known to be $\mathcal{NP}$-complete.

Apart from the performance improvements that can be achieved in practice, SATzilla also showed that it is possible to predict the solve time of $\mathcal{NP}$-complete problems with accuracy sufficient to perform Algorithm Selection and achieve the demonstrated improvements. This is somewhat surprising, given that the definition

of the complexity class involves an element of non-determinism that in theory makes the behaviour of algorithms on its problems inherently hard to predict.

Subsequent research has applied Machine Learning in many different forms to many more application domains. Today, it has established itself as the de facto standard way of tackling Algorithm Selection problems. Particularly in complex application domains, such as combinatorial search problems, it is the only feasible choice.

After many years of research into techniques for solving Algorithm Selection problems, we are faced with a similar situation to the rest of Artificial Intelligence – there is a very high probability that there already is a method for solving a particular problem effectively and efficiently, the challenge is to find it. In a way, efforts to solve the Algorithm Selection Problem have created another Algorithm Selection problem instead of a solution.

This dissertation looks at ways of solving that conundrum, in particular for the most important problem domain of combinatorial search problems. The main thread throughout can be put in the form of the following question.

How should we do Algorithm Selection for combinatorial search problems in practice?

The difficulty of answering this question will be demonstrated throughout this dissertation.

Answering this question involves a number of different things. Apart from assessing the suitability and performance of methods that have been used before, there is the question of whether techniques that have not yet been applied to the Algorithm Selection Problem would be better. It is furthermore worth investigating whether there are general techniques that can assist with Algorithm Selection in practice.

The scope of this dissertation limits the extent to which this question can be answered. While we aim to show the general applicability of techniques and try to draw general conclusions, there may be cases in which the conclusions do not hold or the techniques are not applicable. We focus the investigation on combinatorial search problems, as they represent an important and prominent application domain, and Algorithm Selection problems in constraint programming in particular.

The central thesis of this dissertation can be formulated as follows. There are Machine Learning techniques that can be applied to Algorithm Selection to decrease the background knowledge required to perform it in practice and achieve good performance. The thesis follows from the central question and the research presented in this dissertation will serve to defend it.

1.1. Contribution

The contributions of this dissertation focus, in line with the main question, on practical aspects of Algorithm Selection for combinatorial search problems. The back-

ground of previous work both establishes the context for this dissertation as well as highlighting the need for a guide on how to do Algorithm Selection in practice.

Two case studies of applying techniques for Algorithm Selection to problems in specific domains illustrate the problems faced by practitioners and touch on ways of tackling them. The case studies culminate in the application of a technique that significantly decreases the difficulty of doing Algorithm Selection in practice and an in-depth comparison of many different Machine Learning methods to establish which ones are likely to yield good performance.

The main contributions of this dissertation are as follows.

- The identification of Machine Learning techniques that are likely to perform well in the context of Algorithm Selection for combinatorial search problems based on large-scale empirical evidence. Either linear regression to predict the run time of an algorithm on a problem or alternating decision trees should be used, both as implemented in the WEKA Machine Learning toolkit.

- The identification and evaluation of ensemble classification as a promising technique for reducing the amount of Machine Learning expertise required to perform Algorithm Selection while maintaining a high level of performance improvements. In addition, ensembles increase the robustness in the sense that bad performance of one constituent of the ensemble can be alleviated by the other constituents.

- A comprehensive survey and comparison of previous approaches to solving the Algorithm Selection Problem. Apart from establishing the context for this work, it highlights the diversity of the area and the difficulty of choosing a particular technique.

- The use of decision trees to improve our understanding for the issues underlying an Algorithm Selection problem. Although decision trees have been used frequently in the literature, using them for this purpose has not been described.

- The demonstration of the feasibility of making multi-level decisions that depend on each other. Even when combining the output of several Machine Learning stages, each with an associated uncertainty, the resulting system is still effective and efficient.

- Two case studies that apply Algorithm Selection techniques to new problem domains and raise issues that are addressed in other parts of this dissertation.

The answers to the main question can be formulated as follows.

- Systems that perform Algorithm Selection for combinatorial search problems should use linear regression to predict the performance of each algorithm on a problem and make the decision to choose which one based on that prediction or use alternating decision trees to directly predict the best algorithm.

Other techniques that will probably exhibit good performance are identified in Chapter 7.

- These techniques can be combined in a Machine Learning ensemble to make the Algorithm Selection system more robust.

In a nutshell, this dissertation serves to bring the power of Algorithm Selection for combinatorial search problems within the grasp of researchers who do not have a background in it. It identifies techniques that are likely to perform well in practice and thus alleviates one of the major obstacles to a more widespread adoption of Algorithm Selection – the difficulty of choosing how to do it.

1.2. Organisation

The structure of this dissertation roughly follows the contributions described above and is centred around the main question posed earlier. Most of the chapters have been published previously, at least in parts. Where this is the case, the respective publication or publications are mentioned in a footnote at the beginning of the chapter. The content of the chapters concentrates on the contributions of the author of this dissertation; a detailed description of the contributions to the previously published materials can be found on page xix.

1.2.1. Motivation

Chapter 2 on page 11 presents an example that reinforces some of the points made here and provides some motivation for Algorithm Selection and evidence for its importance and difficulty. A survey of four modern constraint solvers on five different problem classes shows that not only are there significant differences in the performance, but that it is also not obvious how to select the most appropriate solver. Although one of the solvers provides the best performance most of the time, there are cases in which using another solver provides better performance.

The chapter also looks at the configuration of a solver, which can be adjusted to achieve better performance in some cases. Again it is not obvious how to arrive at the best configuration for a specific problem; in this case the datum required to inform this decision is only available after having solved the problem to completion.

1.2.2. Background

Chapter 3 on page 25 presents a detailed survey of the Algorithm Selection literature. At the beginning, the Algorithm Selection Problem itself is described in its various forms in detail. After that, the terminology and some background knowledge for the most common application domains is given. An analysis and classification of the

literature according to specific criteria that are instrumental in Algorithm Selection systems follows.

Over the years, many different papers have advocated many different approaches and methodologies. Chapter 3 puts all of this work into context and contrasts and compares the different approaches. The field of automatically finding the best configuration for a system on a problem is closely related and the literature from that field most relevant to Algorithm Selection is described as well.

A survey of the literature makes clear that there is no single best approach to solving the Algorithm Selection problem and again reinforces the difficulty of the decision of which technique to choose for doing Algorithm Selecting in practice. It furthermore highlights the lack of studies that compare the behaviour of different methods – there is only a handful that do, and even fewer provide quantitative evidence. However, the number of different approaches shows the crucial need for such studies.

1.2.3. Case study I

Chapter 4 on page 47 uses Algorithm Selecting techniques to tackle a problem in a specific domain. In constraint solving, so-called lazy learning is a technique that in some cases improves the solving performance dramatically, but slows the process down in most cases. The problem of deciding whether to use lazy learning or not is an obvious application for Algorithm Selecting and Machine Learning techniques, as the factors that affect the performance on a particular problem are poorly understood.

Apart from demonstrating performance improvements by applying a series of learned decision trees to choose lazy learning or not, the chapter employs the problem features used in the decision trees to improve our understanding of lazy learning and when it is effective. This is a relatively obvious use of the outcome of the Machine Learning stage, but done very rarely in practice.

The Machine Learning methodology steps through a series of decision trees that are refined in order to make the system more efficient and more likely to generalise to unseen problems. A variation on an established Machine Learning technique for estimating the generalisation error of a classifier is used to provide strong evidence for the general applicability of the final decision tree.

1.2.4. Case study II

Chapter 5 on page 61 presents the second case study and looks at implementation alternatives for the `alldifferent` constraint. Again taken from constraint solving, this application domain is more difficult than lazy learning because the set of options to choose from is larger, with some of the options dependent on another option. This characteristic establishes the need to make a series of dependent decisions instead of a single one.

Instead of learning and manually refining a decision tree as in the previous chap-

ter, this case study aims to automate the processes involved to a greater extent and presents a comparison of different Machine Learning classifiers on the problem. Following the main thread of this dissertation, this comparison provides quantitative evidence for the difficulty of doing Algorithm Selection in practice.

Apart from establishing and demonstrating the effectiveness of a particular Machine Learning methodology that will be used in evaluations in the remainder of this dissertation, the case study provides evidence that making a series of decisions instead of a single one can be done efficiently and effectively using standard Machine Learning techniques.

1.2.5. Ensemble classification

Chapter 6 on page 73 addresses some of the issues raised especially in the second case study and presents one of the major contributions of this dissertation. By applying the well-established Machine Learning technique of ensemble classification to Algorithm Selection, it shows that effectiveness and efficiency can be retained while avoiding having to pick a particular Machine Learning technique.

Ensemble classification directly addresses the main question of this dissertation and provides an elegant means of alleviating the difficulty of selecting a technique for doing Algorithm Selection manually, especially for researchers without a strong background in Algorithm Selection. An ensemble of classifiers is furthermore more robust than a single classifier with respect to its performance across different problems and thus reduces the amount of work necessary to ensure good performance of the learned model on new problems.

Chapter 6 does not only show the effectiveness of an ensemble of a large number of classifiers, but also that this effectiveness is retained when reducing the size of the ensemble significantly. The main advantage of having a small ensemble is that the efficiency of the system is improved, as fewer classifiers have to be trained and run on data.

1.2.6. Comparison of different techniques

Chapter 7 on page 81 presents another major contribution of this dissertation and addresses the lack of studies comparing the performance of different Machine Learning techniques for solving the Algorithm Selection Problem. It performs a large scale comparison of many different Machine Learning approaches and methodologies on several Algorithm Selection data sets taken from the literature. It also applies a new Machine Learning technology to the Algorithm Selection Problem for the first time.

The main results presented in the chapter are based on a statistical simulation technique that estimates the performance of a particular technique in general. This, combined with the experimental results on data sets from different and representative application domains, suggests the general applicability of the results and establishes the recommended Machine Learning techniques as suitable choices for Algorithm

Selection.

Apart from the main line of investigation, the chapter looks at a range of related issues. Each one of these issues has an effect on the performance of the Algorithm Selection system, but has not been investigated systematically in the literature. Another measure taken that is rarely done in the literature is that the configuration with the best performance was selected for each of the compared Machine Learning algorithms.

The chapter addresses the question of how to do Algorithm Selection in practice further as well as complementing the results presented in Chapter 6 – even when using an ensemble of classifiers, there is still the question of which classifiers to use in the ensemble. The techniques recommended in this chapter can be used either on their own or within an ensemble to increase the robustness of the Algorithm Selection system.

1.2.7. Conclusions

Chapter 8 on page 103 summarises the other chapters. It wraps up the dissertation by listing the contributions and discussing the results. The scope of the work is clarified and limitations outlined. The chapter closes by presenting an outlook on future work and outlining promising avenues for further research.

2

Motivation

The fundamental question that arises when dealing with Algorithm Selection problems is whether it matters in practice which algorithm to choose. While it is intuitively plausible that there is not a single algorithm that will always be best, it is less intuitive that Algorithm Selection would make a significant difference in practice.

In this chapter, we will provide evidence that it can indeed make a significant difference. We compare the performance of five state of the art constraint solvers. We chose to compare constraint solvers for several reasons. First, constraint programming is a relatively mature area of Artificial Intelligence with many decades of research. There are established and proven ways of solving constraint problems and contemporary constraint solvers implement these. Second, there are several constraint solvers from which to choose. Most of them have been successfully used in academic and industrial applications.

Finally, researchers in constraint programming have recognised the difficulty of choosing a way of solving a constraint problem and there is at least one solver that aims to provide good performance out of the box without the need for manual intervention. Constraint programming also represents an important application domain of Algorithm Selection research.

2.1. Introduction

A *constraint satisfaction problem* (CSP, Dechter (2003)) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. An assignment maps a variable to a value from its domain. Each constraint specifies allowed combinations of assignments of values to a subset of the variables. A *solution* to a CSP is an assignment to all the variables that satisfies all the constraints. Solutions are typically found for CSPs through systematic search of possible assignments to variables. During search, constraint *propagation* algorithms are used. These propagators make inferences, usually recorded as domain reductions, based on the domains of the variables constrained and the assignments that satisfy the constraints. If at any point these inferences result in any variable having an empty

Part of the material in this chapter has been published in: Lars Kotthoff. Constraint Solvers: An empirical evaluation of design decisions. CIRCA preprint 2009/7, University of St Andrews, 2009. The contributions of the author of this thesis are listed on page xix et seqq.

domain then search backtracks and a new branch is considered. A *constraint solver* is a software system that provides means of representing variables and constraints and implements propagation and search algorithms that enable the user to find solutions.

Contemporary constraint solvers are very complex software systems. Each one of the many available today has its own characteristics, its own design decisions that the implementers made and its own philosophy. The traits of a solver that will affect the performance for a particular problem class or instance often cannot be determined easily. Picking a particular solver is therefore a difficult task that requires specialist knowledge about each solver and is likely to have a significant impact on performance. In addition, each solver has different ways of modelling problems, i.e. of expressing an abstract problem in a way that the solver can search for a solution. Not only do users need experience with a particular solver to model a problem in a way that enables it to be solved efficiently, but it is also hard to compare solvers objectively.

This chapter studies a selection of constraint solvers and assesses their performance on problem models that were made as similar as possible. The aim is to show the performance differences that can occur even between state-of-the-art systems. A reasonable default choice, i.e. a solver that will always have good performance, can be made easily, but picking the solver with the best performance for each particular problem is much harder.

The best solver varies not only depending on the problem class, i.e. a family of similar problems, but also depending on the problem instance, i.e. a particular problem from a family. We also provide evidence that solvers that exhibit bad performance on some problem instances have the potential to perform much better on other instances.

The differences in performance between the individual solvers and the difficulty of choosing a particular one for a given problem provide a motivation for the work carried out in this dissertation. Without effective means of performing Algorithm Selection, we can achieve reasonable performance, but will always fall short of achieving the best possible performance. While it is unlikely that a system will ever achieve the best possible performance, Algorithm Selection will get us some of the way there.

The investigation was performed in 2008; the solver versions that were the most recent at that time were used.

2.1.1. Caveat

The performance of the individual solvers in the experiments should *not* be taken as a benchmark or as a suggestion of the general performance of a solver. The focus of the experiments was to compare the solvers on models that are as similar as possible. For any other application, the problem model will be tuned for a particular solver to use its specific strengths which cannot be compared here. It is entirely possible that with a carefully-tuned model a solver that performs badly in an experiment reported here becomes much better than any other solver.

solver	programming language	first release	modelling approach
Choco	Java	1999	code with library functions
ECLiPSe	C/Prolog	1990	code with library functions
Gecode	C++	2005	code with library functions
Minion	C++	2006	modelling language

Table 2.1. Summary of the characteristics of the investigated solvers.

2.2. Background

The first constraint solvers were implemented as constraint logic programming environments in logic programming languages such as Prolog in the early 1980s. The logic programming paradigm lends itself naturally to solving constraint problems because facilities like depth-first backtracking search and nondeterminism are already built into the host language. Related ideas also arose in Operations Research and Artificial Intelligence. Notable developments of that time include extensions to Prolog and the CHIP constraint programming system (Dincbas et al., 1988).

Starting in the 1990s, constraint programming found its way into procedural and object-oriented languages, most notably C++. ILOG Solver (IBM, 2011) pioneered this area. It became apparent that it would be beneficial to separate the solving of constraint problems into two phases; modelling the problem and programming search. Since then, constraint solvers have improved significantly in terms of performance as well as in terms of ease of use.

2.3. Surveyed constraint solvers

The constraint solvers chosen for this investigation are Choco (The Choco Team, 2011), version 2.0.0.3, ECLiPSe (Aggoun et al., 2011), version 6.0_42, Gecode (Schulte et al., 2011), version 2.2.0 and Minion (Jefferson et al., 2011), version 0.7. The solvers were chosen because all of them are currently under active development. Furthermore they are Open Source; implementation details not described in papers or the manual can be investigated by looking at the source code.

Table 2.1 presents a brief summary of the solvers and their basic characteristics.

2.3.1. Choco

Choco was initially developed in the CLAIRE programming language as a national effort of French researchers for an open constraint solver for teaching and research purposes. Since then, it has been reimplemented in the Java programming language and gone through a series of other changes. Version 2 is a major refactoring to provide a better separation between modelling and solving a problem, as well as

performance improvements.

2.3.2. ECLiPSe

ECLiPSe is one of the oldest constraint programming environments still being used and in active development. It was initially developed at the European Computer-Industry Research Centre in Munich and then at IC-Parc, Imperial College London, until the end of 2005 when it became Open Source. It is implemented in Prolog and therefore provides a higher level of abstraction than the other systems.

2.3.3. Gecode

Gecode is a C++ environment for developing constraint-based systems and applications. It is developed by researchers in Sweden, Germany and Belgium. It aims to be simple and accessible. One of its key features is the availability of extensive documentation in the form of manuals, tutorials and academic publications.

2.3.4. Minion

Minion was implemented to be a solver that requires only an input file to run and no written code. This way the solver could be made fast by not being extensible or programmable and fixing the design decisions. There are options that can be adjusted through command line switches, but the main intention for it is to be a black box solver that aims to provide good performance on a wide range of problems in the spirit of the "Model and Run" paradigm (Puget, 2004).

2.4. Surveyed constraint problems

Five classes of constraint problems were investigated. They are the n-Queens, Golomb Ruler, Magic Square, Social Golfers and Balanced Incomplete Block Design problems. Most of the problems are described in CSPLib (Gent and Walsh, 1999). Their characteristics are,

n-Queens Place n queens on an $n \times n$ chessboard such that no queen is attacking another queen.

Golomb Ruler (CSPLib problem 6)
For a given m, a Golomb ruler is defined as a set of m integers $0 = a_1 < a_2 < \ldots < a_m$ such that the $\frac{m(m-1)}{2}$ differences $a_j - a_i, 1 \leq i < j \leq m$ are distinct. Such a ruler is said to contain m marks and is of length a_m, i.e. the position of the last mark. The length is to be minimised.

Magic Square (CSPLib problem 19)

An order n magic square is a $n \times n$ matrix containing the numbers 1 to n^2, with each row, column and main diagonal equal the same sum $\frac{n(n^2+1)}{2}$.

Social Golfers (CSPLib problem 10)
In a golf club where m groups of n golfers play over p weeks, schedule the groups such that no golfer plays in the same group as any other golfer twice.

Balanced Incomplete Block Design (CSPLib problem 28)
A Balanced Incomplete Block Design (BIBD) is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. v, b, r, k and λ are given, although b and r can be derived from the other parameters.

For each problem class, several different instances were chosen. This choice was purely based on the time required to solve the problems to be able to compare both long and short runs. The instances selected were,

n-Queens $n = \{20, 21, 22, 23, 24, 25, 26, 27, 28, 29\}$

Golomb Ruler $m = \{9, 10, 11, 12, 13\}$

Magic Square $n = \{4, 5, 6\}$

Social Golfers $\langle p, m, n \rangle = \{\langle 2, 4, 4 \rangle, \langle 2, 5, 4 \rangle, \langle 2, 6, 4 \rangle, \langle 2, 7, 4 \rangle, \langle 2, 8, 4 \rangle, \langle 2, 9, 4 \rangle, \langle 2, 10, 4 \rangle\}$

BIBD $\langle v, k, \lambda \rangle = \{\langle 7, 3, 10 \rangle, \langle 7, 3, 20 \rangle, \langle 7, 3, 30 \rangle, \langle 7, 3, 40 \rangle, \langle 7, 3, 50 \rangle, \langle 7, 3, 60 \rangle, \langle 7, 3, 70 \rangle\}$

The models were derived from the examples included with the distributions of the solvers. For some solvers and some problems the example model was simply adapted to match the models for the other solvers, in other cases the problem was modelled from scratch. The models are described below.

n-Queens The problem was modelled with n variables, one for each queen and one auxiliary variable for each pair of rows holding the difference of the column positions of the queens in those rows. These auxiliary variables were constrained to not be equal to the difference in rows or the negative thereof to enforce the constraint that no two queens can be on the same diagonal. The n decision variables were constrained by an `alldifferent` constraint, which requires all variables in its scope to have a value that is different from the values assigned to any of the other variables.

Golomb Ruler The Golomb Ruler model had m variables, one for each tick, and one auxiliary variable for each pair of ticks to hold the difference between them. Additional constraints set the value of the first tick to be 0 and enforced an

increasing monotonic ordering on the ticks. The auxiliary variables holding the differences between the ticks were constrained by an `alldifferent` constraint. An `alldifferent` constraint requires all variables that it constrains to have distinct values and is a more powerful way of expressing binary not-equals constraints over a set of variables. The optimisation constraint minimised the value of the last tick, which is equivalent to the length a_m.

Magic Square There were n^2 variables, one for each of the cells of the magic square. The constraints required all those variables to be different and all rows, columns and diagonals to sum to the magic sum. Additionally, four constraints were introduced to require the value in the top left square to be less than or equal to the values in the other corners of the square and the top right value to be less than or equal to the bottom left value.

Social Golfers The model of the Social Golfers problem used a $p \times m \times (n \cdot m)$ matrix of decision variables. The first dimension represented the weeks, the second the groups and the third the players by group. The constraints required that each player plays exactly once per week, the sum of the players in each group is equal to the number of players per group specified and each pair of players meets at most once. For the last constraint, one auxiliary variable for each pair of players by group times weeks times groups was introduced. Additional ordering constraints were introduced among weeks, groups and players to prohibit different solutions that merely swap two weeks, groups or players and thus reduce the search space.

Balanced Incomplete Block Design The BIBD model introduced a matrix of $v \times b$ decision variables. The rows were constrained to sum to r, the columns to k and the scalar product between each pair of rows was constrained to equal λ. For the last constraint, one auxiliary constraint per pair of rows times b was introduced. Ordering constraints were put on each pair of rows and each pair of columns.

All models except the BIBD and Social Golfers used variables with integer domains. The models of BIBD and Social Golfers used Boolean variables in the solvers that provide specialised Boolean variables – Choco, Gecode and Minion. For all models, the same variable and value ordering was specified. The solutions the different solvers found for each problem instance were the same.

This set of benchmarks covers a variety of different constraint problems, such as optimisation problems and problems usually modelled with integer and Boolean variable domains. The models involve binary constraints, i.e. constraints on two variables, as well as global constraints, i.e. constraints on two or more variables.

Table 2.2 on page 18 lists the number of variables, their domains and constraints for each problem instance. If the domains of the auxiliary variables are different from the domains of the main variables, they are given in parentheses. Minion does not provide a sum equals constraint; it can however be emulated by combining a

sum less than and sum greater than constraint. This results in a higher number of constraints for Minion; this number is given in parentheses.

The purpose of this investigation is to compare the solvers on equivalent models to be able to assess how the design decisions they have made affect their performance. The models of the problems are in no case the optimal model for the particular solver or the particular problem. The results cannot be seen as providing a performance comparison of the solvers in general, as for such a comparison the models would have to be tailored to each solver to achieve the best performance. For such a comparison, see for example Lecoutre et al. (2008).

The evaluation only takes the CPU time each solver requires to solve the problem into account. Other measures, such as elapsed time or required memory, are not considered.

2.4.1. Amount of search

The amount of search each solver does on each problem instance is roughly the same. This was ensured by comparing the node counts for each instance for the solvers which provide node counts, visually inspecting the search tree for solvers that provide visualisation tools and manually comparing the decisions made at each node of the search tree for smaller instances.

2.5. Results

The following figures show the performance of the solvers for each problem class and instance.

All experiments were conducted on a dual quad-core Intel Xeon 2.66 GHz with 16 GB of memory running CentOS Linux 5. The CPU time was measured with the `time` command line tool. The numbers reported as CPU time are the sum of user and system time. Each problem was solved five times by each solver; we report the median of those runs. The coefficient of variation* was less than 10% in general. Instances where it was larger are discussed below.

Figure 2.3 on page 20 shows that for the Magic Square problem models, Gecode has the best performance. For the Golomb Ruler problem models (Figure 2.2 on page 19), Gecode and Minion show a very similar performance. For n-Queens (Figure 2.1 on page 19), Gecode and Minion both win on different problem instances. For the other problem models, Minion was fastest.

The best choice for a default solver with reasonable performance in all cases would, based on these results, be Minion. There is a significant number of cases however where Gecode is faster, sometimes substantially.

Figure 2.1 on page 19 shows that the relative differences in CPU time between the solvers stays approximately the same across different instances, except for very

*The coefficient of variation is the standard deviation divided by the mean.

problem	instance	variables	domains	constraints
n-Queens	20	210	{0..19} ({−19..19})	571 (761)
	21	231	{0..20} ({−20..20})	631 (841)
	22	253	{0..21} ({−21..21})	694 (925)
	23	276	{0..22} ({−22..22})	760 (1013)
	24	300	{0..23} ({−23..23})	829 (1105)
	25	325	{0..24} ({−24..24})	901 (1201)
	26	351	{0..25} ({−25..25})	976 (1301)
	27	378	{0..26} ({−26..26})	1054 (1405)
	28	406	{0..27} ({−27..27})	1135 (1513)
	29	435	{0..28} ({−28..28})	1219 (1625)
Golomb Ruler	9	45	{0..81}	46 (82)
	10	55	{0..100}	56 (101)
	11	66	{0..121}	67 (122)
	12	78	{0..144}	79 (145)
	13	91	{0..169}	92 (170)
Magic Square	4	16	{1..16}	15 (25)
	5	25	{1..25}	17 (29)
	6	36	{1..36}	19 (33)
Social Golfers	2,4,4	1088	{0..1}	1133 (1293)
	2,5,4	2100	{0..1}	2161 (2401)
	2,6,4	3600	{0..1}	3679 (4015)
	2,7,4	5684	{0..1}	5783 (6231)
	2,8,4	8448	{0..1}	8569 (9145)
	2,9,4	11988	{0..1}	12133 (12853)
	2,10,4	16400	{0..1}	16571 (17451)
BIBD	7,3,10	1960	{0..1}	1643 (1741)
	7,3,20	3920	{0..1}	3253 (3421)
	7,3,30	5880	{0..1}	4863 (5101)
	7,3,40	7840	{0..1}	6473 (6781)
	7,3,50	9800	{0..1}	8083 (8461)
	7,3,60	11760	{0..1}	9693 (10141)
	7,3,70	13720	{0..1}	11303 (11821)

Table 2.2. Number of variables and constraints for the investigated problems.

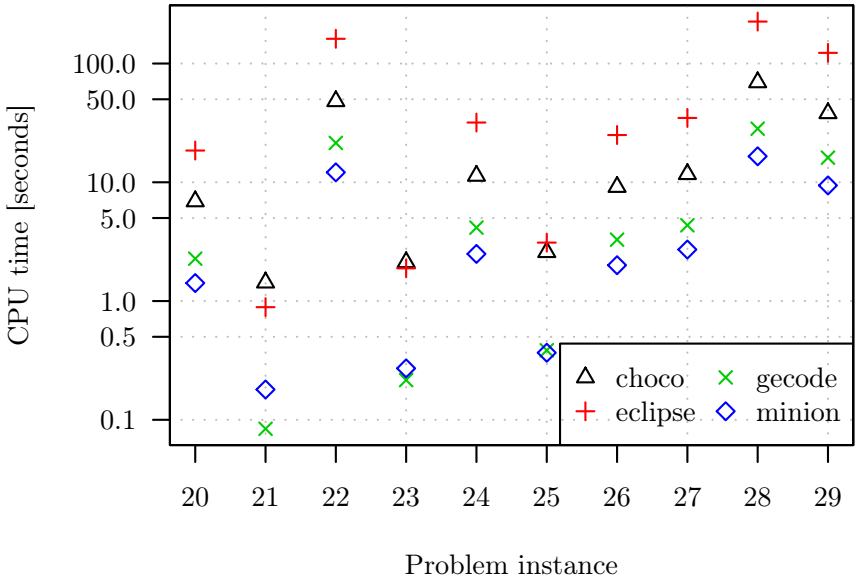


Figure 2.1. CPU time comparison for n-Queens.

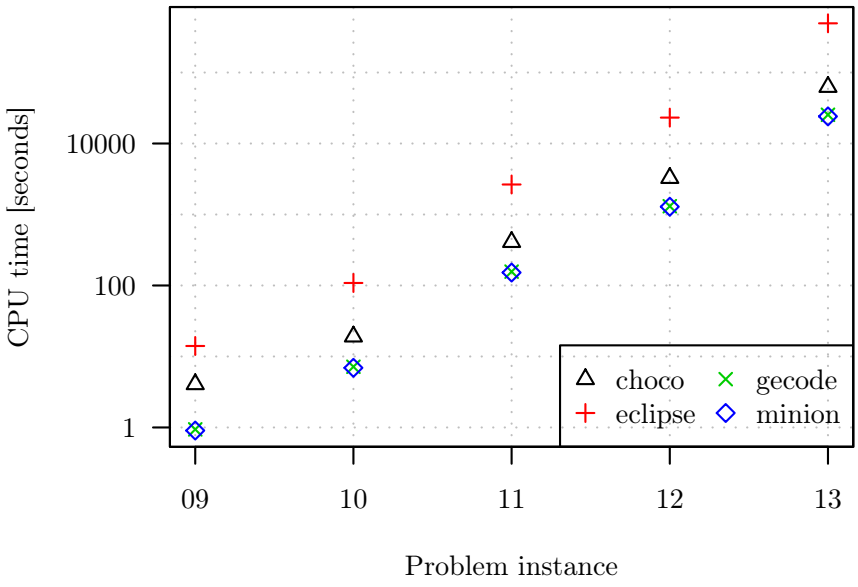


Figure 2.2. CPU time comparison for Golomb Ruler.

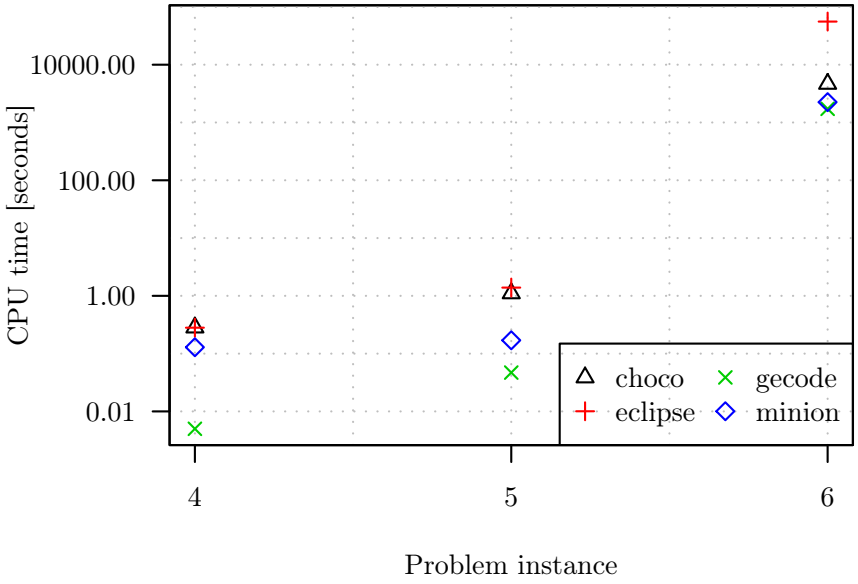


Figure 2.3. CPU time comparison for Magic Square.

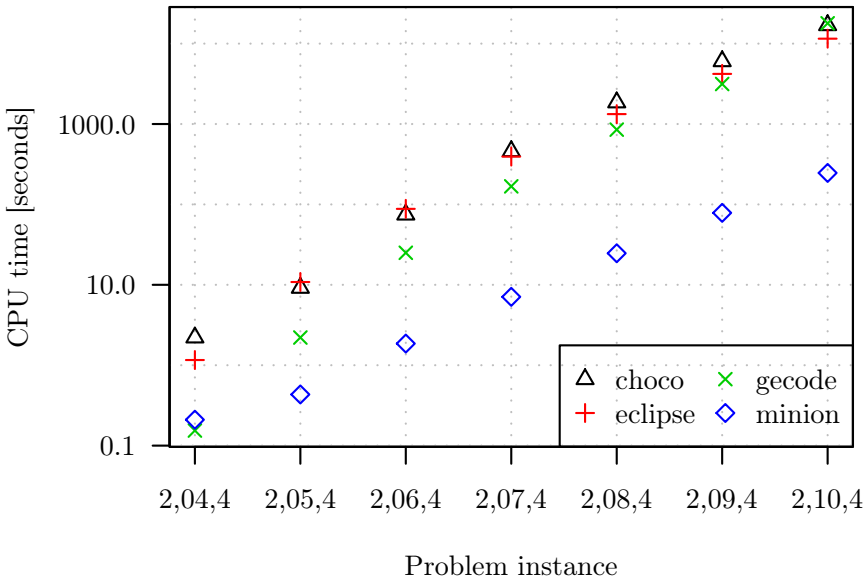


Figure 2.4. CPU time comparison for Social Golfers.

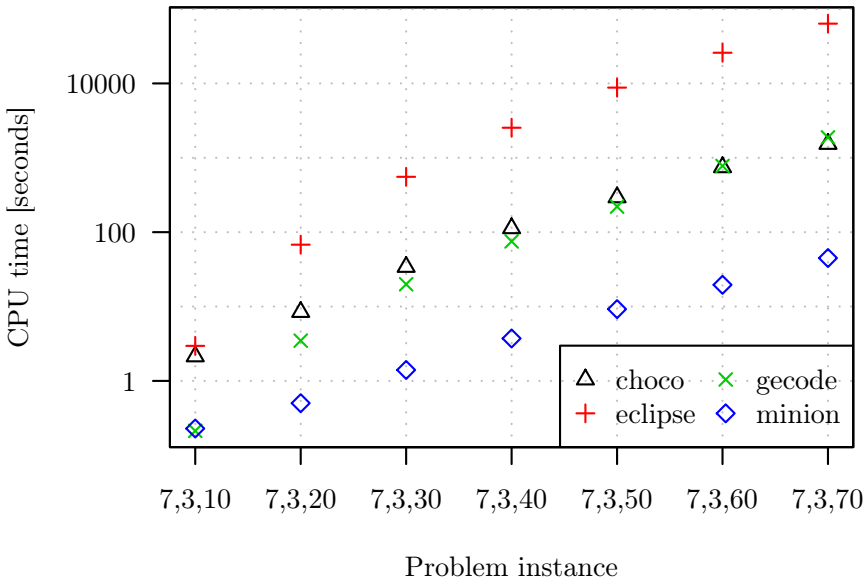


Figure 2.5. CPU time comparison for Balanced Incomplete Block Design.

small problems where the setup cost contributes the largest share of the CPU time (cf. Section 2.5.1). The same effect is even stronger for the Golomb Ruler problem (Figure 2.2 on page 19), where the total CPU times are larger.

Figure 2.3 on the facing page suggests a slightly different behaviour for the Magic Square problem. However, there are not enough data points to draw definitive conclusions. This problem was only run up to instances of size 6 because instances of size 7 took too long.

The coefficient of variation between the five runs for the 2,10,4 Social Golfers instance for Gecode was about 20%. Even considering the large variation, the key point – ECLiPSe performs better than Gecode, which is roughly the same as Choco – remains valid.

2.5.1. Setup costs and scaling

In all of the experiments except the Golomb Ruler, Gecode is the fastest solver for the smallest problem instances in terms of solve time. For instances that take longer to solve, its relative position changes and other solvers are faster.

Both Choco and ECLiPSe run in abstract machines that incur some setup cost when starting up. Minion reads an input file, parses it and constructs the problem to solve from that. The overhead incurred because of these issues accounts for the difference to Gecode for the smallest problems. For the Golomb Ruler problem, the CPU time Gecode takes to solve the smallest problem is equal to the time Minion takes. This is because the CPU time required to solve this instance is large compared

to the CPU required for the smallest instances of the other problem classes – it takes roughly a second whereas for other problem classes the smallest instance is solved in a fraction of a second. The overhead Minion incurs for parsing the input file is small and only accounts for a small fraction of the total CPU time in this case.

Figure 2.4 on page 20 shows that for the Social Golfers problem, ECLiPSe scales better than the other solvers with respect to the increase in CPU time with increasing problem size. Starting with the 2,7,4 instance, it is faster than Choco and for the largest instance it is faster than Gecode as well. Extrapolating past the end of the graph, it is possible that for very large instances ECLiPSe could be faster than Minion. Figure 2.5 on the preceding page on the other hand shows a different picture. Here the increase in CPU time for ECLiPSe and Gecode with increasing problem size is significantly larger than that of Choco and Minion. For the 7,3,60 problem instance, Choco is faster than Gecode despite being slower before.

These factors constitute a further difficulty for choosing the best overall solver. Apart from different performance on different problem classes, the performance across different instances of the same problem class also varies. In particular, the performance of the solvers scales differently as the size of the problem instance to solve increases. If we chose Minion as the solver to use but only solve very small problems, Gecode would have given a much better performance. Similarly, if we only solve very large problems, it is possible that another solver would have been a better choice.

2.5.2. Recomputation versus copying in Gecode

In addition to choosing a solver, there are also parameters for each solver which may increase or decrease performance on a particular problem instance. Choosing the best values for those parameters is similar to choosing the solver with the best performance – it is relatively easy to choose a default that will give good performance on almost all problems, but there is space for improvement by adapting the parameter values to particular problems.

Gecode provides parameters to tune the ratio of copying to recomputation. Setting this parameter will serve as an example for the difficulty of setting the value and to demonstrate that a default setting will not give the best performance on all problem instances. The n-Queens problem, the Social Golfers problem and the BIBD problem were rerun with recomputation distances of 1 (full copying – the same behaviour as Minion), 8 (the default), 16 and 32. The adaptive recomputation distance was left at the default value of 2 (cf. Schulte (2002)). These particular problem classes were chosen because both Magic Square and Golomb Ruler have a relatively small number of variables. Therefore the search tree is comparatively shallow and the effects of changing this parameter setting are not as pronounced as for the chosen problems.

The results were compared with the Kruskal-Wallis one-way analysis of variance test. The differences are not statistically significant because of the large variation among the CPU times for the problem instances; however when comparing the dif-

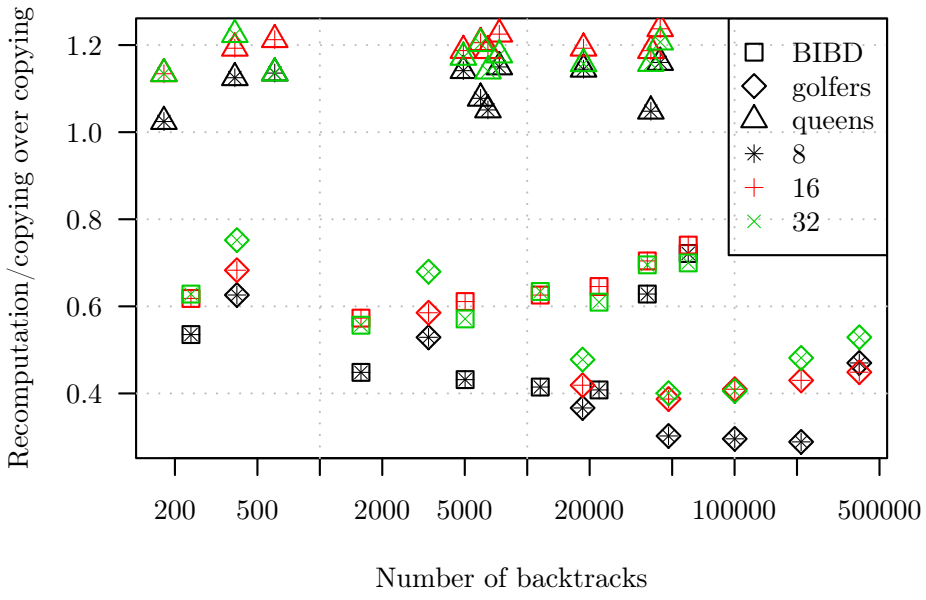


Figure 2.6. CPU time for different levels of recomputation and copying over CPU time for copying for Gecode. Shapes denote problem classes, crosshairs denote recomputation distances. Values less than 1 denote that copying and recomputation is faster than copying at every node.

ferences between doing a full copy at each node (recomputation distance 1) and the other recomputation distances with the Wilcoxon test, the differences were statistically significant at the 0.05 level.

Figure 2.6 shows the results for all the problems and recomputation distances. Note that the default recomputation distance in Gecode is 8, i.e. the results shown in Figures 2.1 on page 19, 2.4 on page 20, and 2.5 on page 21 are not the CPU times that the other CPU times are divided by.

For all instances and recomputation distances of the n-Queens problem, making a full copy at every node of the search tree performs better than a recomputation distance > 1. For problems with only few variables, it might be better always to copy. The performance improvement is only up to about 22% though.

The default value for the recomputation distance achieves the best performance in most of the cases. The recomputation distance appears to be proportional to the number of backtracks – for the lowest number of backtracks, copying at every node (i.e. a recomputation distance of 1) is fastest, whereas for the largest number of backtracks 16 is better than 8.

2.6. Summary

The results in this chapter show that the performance differences between different solvers can be significant. If performance is something to be considered, there is a clear need to evaluate different solvers and choose based on results such as the ones presented in this chapter.

But even when making such a decision based on empirical evidence, there will be room for performance improvements in some cases. Choosing a single solver will give suboptimal performance for some problems because of the nature of that problem and assumptions that the designers of the chosen solver made that may not necessarily be true in this particular case.

Algorithm Selection provides a way of improving on this situation. Instead of manually selecting a solver to tackle all problems with, it allows us to choose the best solver from the list of candidates to achieve better performance on individual problems. But making this decision is not straightforward. In the results presented here, the decision of which solver to choose often depends on the size of the problem – how long will it take to solve it or how many backtracks will the solver need?

These are of course questions that we can only answer once the problem has been solved. At that point the information we could get from the answers is useless however – we already solved the problem, there is no need to do that again, even if we knew how to do it faster. We need a way of choosing the best solver *before* solving the problem.

This is exactly what Algorithm Selection is concerned with. Given a problem, examine it and, depending on its characteristics, make the decision of what solver to use. Being able to do so efficiently does not only make it easier for humans to solve problems because the right way of solving it is selected automatically, but also more efficient because the most appropriate solver is selected on a case-by-cases basis instead of relying on a default choice giving good performance in general.

3

Background

Algorithm Selection is not a new problem. Researchers recognised long ago that a single algorithm will not give the best performance across all problems one may want to solve and that selecting the most appropriate method is likely to improve the overall performance. Empirical evaluations like the one in the previous chapter have provided compelling evidence for this (e.g. Aha (1992)).

The Algorithm Selection Problem has, in many forms and under different names, cropped up in many areas of research in the last few decades. Today there exists a large amount of literature on it. Most publications are concerned with new ways of tackling this problem and solving it efficiently in practice. This chapter surveys the available literature and describes how research has progressed.

3.1. The Algorithm Selection Problem

The original paper describing the Algorithm Selection Problem was published in Rice (1976). The basic model described in the paper is very simple – given a space of problems and a space of algorithms, map each problem-algorithm pair to its performance. This mapping can then be used to select the best algorithm for a given problem. The original figure illustrating the model is reproduced in Figure 3.1 on the next page. As Rice states,

> “The objective is to determine $S(x)$ [the mapping of algorithms to problems] so as to have high algorithm performance.”

He identifies the following four criteria for the selection process.

1. Best selection for all mappings $S(x)$ and problems x. For every problem, an algorithm is chosen to give maximum performance.

2. Best selection for a subclass of problems $x' \subseteq x$. A single algorithm is chosen to apply to each of a subclass of problems such that the performance degradation compared to choosing from all algorithms is minimised.

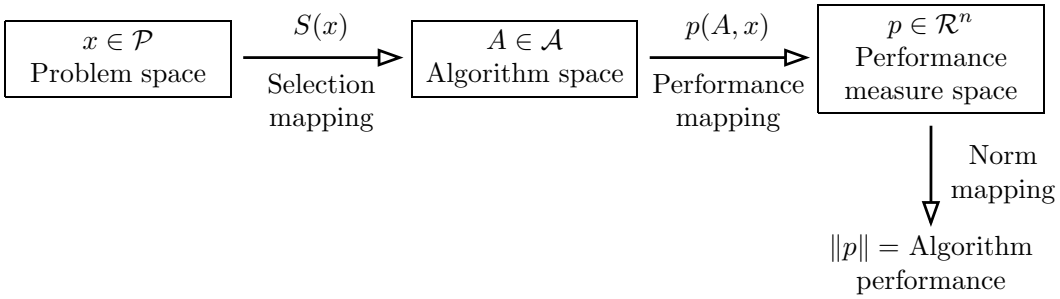


Figure 3.1. Basic model for the Algorithm Selection Problem as published by Rice (1976).

3. Best selection from a subclass of mappings $S' \subseteq S$. Choose the selection mapping from a subset of all mappings from problems to algorithms such that the performance degradation is minimised.

4. Best selection from a subclass of mappings and problems $S'(x')$. Choose a single algorithm from a subset of all algorithms to apply to each of a subclass of problems such that the performance degradation is minimised.

The first case is clearly the most desirable one. In practice however, the other cases are more common – we might not have enough data about individual problems or algorithms to select the best mapping for everything.

Rice (1976) lists five main steps for solving the problem.

> **Formulation** Determination of the subclasses of problems and mappings to be used.
>
> **Existence** Does a best selection mapping exist?
>
> **Uniqueness** Is there a unique best selection mapping?
>
> **Characterization** What properties characterize the best selection mapping and serve to identify it?
>
> **Computation** What methods can be used to actually obtain the best selection mapping?

This framework is taken from the theory of approximation of functions. The questions for existence and uniqueness of a best selection mapping are usually irrelevant in practice. As long as a *good* performance mapping is found and improves upon the current state of the art, the question of whether there is a different mapping with the same performance or an even better mapping is secondary. While it is easy to determine the theoretically best selection mapping on a set of given problems, casting this mapping into a *generalisable* form that will give good performance on

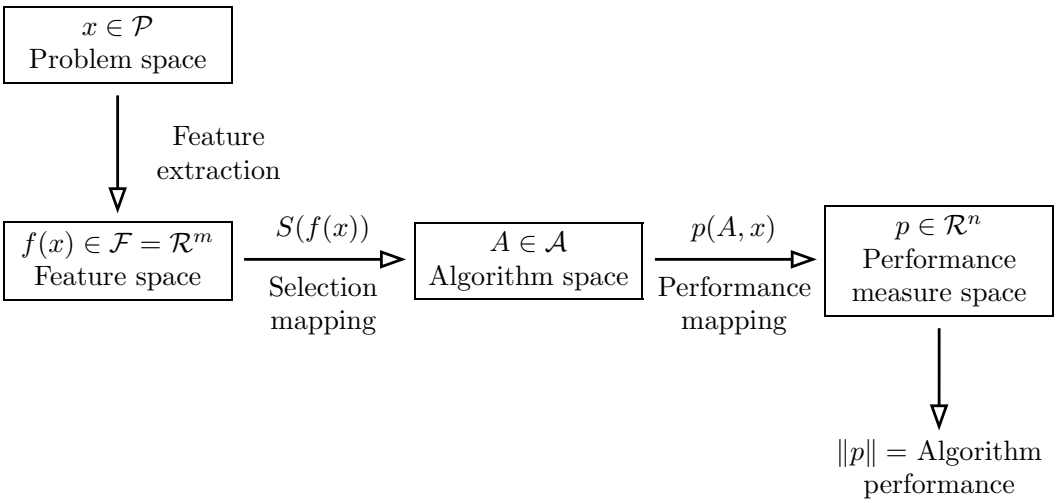


Figure 3.2. Refined model for the Algorithm Selection Problem with problem features (Rice, 1976).

new problems or even into a form that can be used in practice is hard. Guo and Hsu (2004) and Cook and Varnell (1997) compare different Algorithm selection models and select not the one with the best performance, but one that is easy to understand, for example. Vrakas et al. (2003) select their method of choice for the same reason. Similarly, Xu et al. (2008) choose a model that is cheap to compute instead of the one with the best performance. They note that,

> “ All of these techniques are computationally more expensive than ridge regression, and in our previous experiments we found that they did not improve predictive performance enough to justify this additional cost. ”

Rice continues by giving practical examples of where his model applies. He refines the original model to include features of problems that can be used to identify the selection mapping. The original figure depicting the refined model is given in Figure 3.2.

This model already contains all the elements we need for this dissertation. Features of each problem in a given set are extracted. The aim is to use these features to produce the mapping that selects the algorithm with the best performance for each problem. The actual performance mapping for each problem-algorithm pair is usually of less interest as long as the individual best algorithm can be identified.

Rice poses additional questions about the determination of features.

- What are the best features for predicting the performance of a specific algorithm?

- What are the best features for predicting the performance of a specific class of algorithms?
- What are the best features for predicting the performance of a subclass of selection mappings?

He also states that,

> The determination of the best (or even good) features is one of the most important, yet nebulous, aspects of the algorithm selection problem.

He refers to the difficulty of knowing the problem space. Many problem spaces are not well known and often a sample of problems is drawn from them to evaluate experimentally the performance of the given set of algorithms. If the sample is not representative, or the features do not achieve a good separation of the problems in the feature space, there is little hope of finding the best or even a good selection mapping.

Tsang et al. (1995) perform one of the earliest experimental investigations into Algorithm Selection. The main point of the paper is to show that there is no algorithm that is universally the best when solving constraint problems. The authors also demonstrate that the best algorithm-heuristic combination is not what one might expect for some of the surveyed problems. This provides an important motivation for research into Algorithm Selection. They close by noting that,

> …research should focus on how to retrieve the most efficient [algorithm-heuristic] combinations for a problem.

In order to learn the performance mapping from problems to algorithms, we use experimentally obtained training data. The accordingly modified version of the model of the Algorithm Selection Problem that will be used in the remainder of this dissertation is shown in Figure 3.3 on the facing page.

The original model from Rice (1976) is modified to take a *training phase* into account, where the selection mapping is learned as a *model* of the performance space. For a sample of the problem space, the features are extracted and the performance of the algorithm space is evaluated experimentally. Using this training data, the model is created by Machine Learning. The term *model* is used only in the loosest sense here; it can be as simple as a representation of all the training data without any further analysis.

3.1.1. Terminology

Algorithm Selection is a very general concept and as such has cropped up frequently in various lines of research. Often however a different terminology is used.

Borrett et al. (1996) use the term *algorithm chaining* to mean switching from one algorithm to another while the problem is being solved. This is an instance of online

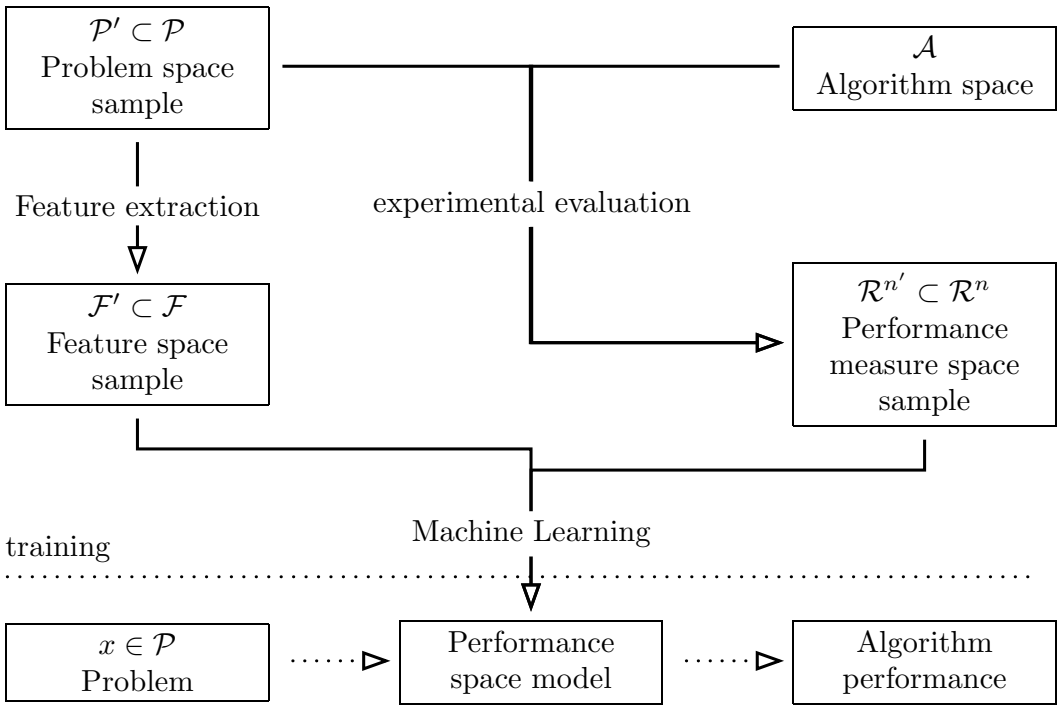


Figure 3.3. Modified Algorithm Selection model used in this dissertation.

Algorithm Selection. Lobjois and Lemaître (1998) call Algorithm Selection *selection by performance prediction*. Vassilevska et al. (2006) use the term *hybrid algorithm* for the combination of a set of algorithms and an Algorithm Selection model (what they term *selector*).

In Machine Learning today, Algorithm Selection is usually referred to as *meta-learning*. This is because Algorithm Selection models learn when to use which method of Machine Learning. The earliest approaches however also spoke of *hybrid approaches*, e.g. Utgoff (1988). Aha (1992) proposes rules for selecting a Machine Learning algorithm that take the characteristics of a data set into account. He uses the term *meta-learning*. Brodley (1993) introduces the notion of *selective superiority*. This concept refers to a particular algorithm being best on some, but not all tasks.

In heuristics research, Algorithm Selection is called *meta-heuristic* or *hyper-heuristic*. An Algorithm Selection model can be seen as a heuristic that decides when to use one of a set of heuristics. The term hyper-heuristic was first used by Cowling et al. (2001). The term meta-heuristic is part of Artificial Intelligence folklore and it is hard to trace its exact origins. The first mention of the term was probably in the paper that proposed Tabu search (Glover, 1986).

In addition to the many terms used for the process of Algorithm Selection, researchers have also used different terminology for the models of what Rice calls the performance measure space. Allen and Minton (1996) call them *runtime performance predictors*. Leyton-Brown et al. (2002), Hutter et al. (2006), Xu et al. (2007a), Leyton-Brown et al. (2009) coined the term *Empirical Hardness model*. This stresses the reliance on empirical data to create these models and introduces the notion of "hardness" of a problem. The concept of hardness takes into account all performance considerations and does not restrict itself to runtime performance for example. In practice however, the described empirical hardness models only take runtime performance into account. In all cases, the predicted measures are used to select an algorithm.

3.2. Search problems

Most of the research on Algorithm Selection focuses on combinatorial search problems. The terminology associated with this kind of problem will be used throughout this dissertation. This section reviews and explains the necessary concepts.

A combinatorial search problem is one where an initial state is to be transformed into a goal state by application of a series of operators or assignment of values to variables. The space of possible states is exponential in the size of the input and finding a solution is $\mathcal{NP}$-hard. Applying an operator to a state or is called expansion of the state. The space of possible states forms a tree; the children of a node are the states that are reachable from it by applying one of the operators. Similarly, assigning a value to a variable changes the state.

During the solution process, the search tree is built by expanding the nodes. At certain points, it may become necessary to unapply operators to revert to an earlier

state. This is usually the case when no operators can be applied to the current search state and it is not the goal state. This process is referred to as backtracking.

A heuristic is a strategy that determines which operators to apply to which nodes. Heuristics are not necessarily complete or deterministic, i.e. they are not guaranteed to find a solution if it exists or to always make the same decision in the same circumstances.

A large part of the literature relevant to this dissertation is concerned with search problems in different domains. Explanations of all the relevant techniques and concepts are beyond the scope of and not necessary for this dissertation. An overview of the field of satisfiability (SAT) is given by Biere et al. (2009). An introduction to constraint programming is given by Dechter (2003) and the reference handbook (Rossi et al., 2006). Automated planning is described by Ghallab et al. (2004). A general overview of Artificial Intelligence search can be found in Russell and Norvig (2009). An introduction to and overview of the relevant Machine Learning techniques is given by Bishop (2007), Witten et al. (2011).

3.3. Expert systems

The earliest Algorithm Selection systems appeared in the context of so-called expert systems. The idea behind them was to make the power of complex libraries available to the non-expert user. As such, the problem domain that those systems deal with usually require a lot of expert knowledge, such as differential equations in Mathematics. The user would set out to solve such a problem and, with the help of the expert system, select the appropriate steps for doing so. In this context, Algorithm Selection is not only used to improve the performance of the system, but also to make solving the problem possible at all.

Algorithm Selection is only a part of the systems described in this section. They are nevertheless mentioned here because they set the context for later systems. Even though the Algorithm Selection Problem was described several decades ago, it did not emerge as an independent area of research until relatively recently.

The level of user interaction during the solving process varies. Some systems only assist the user while others require the user to only specify the problem to solve. All systems incorporate some kind of expert knowledge; either in terms of explicitly given rules and transformations to apply to a problem or implicitly learned operations.

The ODEXPERT system (Kamel et al., 1993) helps the user with the selection of a numerical solver for initial value ordinary differential equations. It uses both explicitly given expert knowledge in the form of rules and decision trees and implicit knowledge derived from past decisions. It relies on a mixture of user-specified and automatically determined characteristics of input problems.

Similarly, PYTHIA (Dyksen and Gritter, 1989, Weerawarana et al., 1996, Joshi et al., 1996) automatically selects from a portfolio of algorithms to solve elliptic partial differential equations from the //ELLPACK system (Houstis et al., 1990). It contains human expert knowledge and maintains performance profiles of the al-

gorithms. In addition to the problem to solve, the user can specify requirements on the time to find the solution and quality of the solution as well.

The system ALEX (Neves, 1985) (Algebra example learner) on the other hand is mostly autonomous and does not require the expert knowledge to be specified explicitly. It learns rules to apply to algebraic equations and the context in which to apply a learned rule from examples of solving algebraic equations.

3.4. Algorithm portfolios

The idea behind expert systems was taken further under the topic of algorithm portfolios. The idea of having a portfolio was taken from Economics, where portfolios are used to maximise utility while minimising the associated risk as described by Huberman et al. (1997). Applied to algorithms, the utility to maximise is the performance of the algorithm or the quality of the solution to a problem.

Algorithm portfolios and expert systems share some of their aims. Expert systems put more emphasis on assisting the user to make a difficult decision while algorithm portfolios focus on improving the performance of the solving process. In that sense, they are two ways of looking at the same thing. Hough and Williams (2006) investigate selecting from a portfolio of optimisation algorithms for example. They stress that existing optimisation software is hard to use for non-expert users because of the number of algorithms and options available. Their approach misses most of the components usually found in expert systems, but is very similar to other portfolio systems described in the literature.

For stochastic algorithms and hard combinatorial problems, the idea of algorithm portfolios was first explored in Gomes and Selman (1997a,b) and formalised and investigated further in subsequent publications, e.g. Gomes and Selman (2001). The technique itself however had been described under different names by other authors at about the same time in different contexts, e.g. Tsang et al. (1995), Allen and Minton (1996), Borrett et al. (1996), Lobjois and Lemaître (1998).

3.4.1. Static portfolios

The most common kind of algorithm portfolio is a static portfolio. It contains a fixed number of algorithms or systems, each with their own performance characteristics. The algorithms and their parameters are not modified. This approach is used for example in SATzilla (Nudelman et al., 2004, Xu et al., 2007b, 2008), AQME (Pulina and Tacchella, 2007, 2009), CPhydra (O'Mahony et al., 2008), ArgoSmArT (Nikolić et al., 2009) and BUS (Howe et al., 1999).

As the algorithms in the portfolio do not change, their selection is crucial for its success. Ideally, the algorithms will complement each other such that good performance can be achieved on a wide range of different problems. Samulowitz and Memisevic (2007) use a portfolio of heuristics for solving quantified Boolean formu-

lae problems that have specifically been crafted to be orthogonal to each other. Wu and van Beek (2007) approach the problem from another direction – they evaluate all possible portfolios and select the one with the best performance and largest distance from a badly-performing portfolio. In most cases however, this is made less explicit. The systems mentioned in the previous paragraph use portfolios of solvers that have performed well in solver competitions with the implicit assumption that they have complementing strengths and weaknesses.

3.4.2. Dynamic portfolios

Rather than relying on a priori properties of the algorithms in the portfolio, dynamic portfolios adapt the algorithms depending on the problem to be solved. One approach is to have a portfolio of algorithmic building blocks that are put together. An example of this is the Adaptive Constraint Engine (ACE) (Epstein and Freuder, 2001, Epstein et al., 2002). The building blocks are so-called advisors, which characterise variables of constraint problem and give recommendations as to which one to process next. ACE combines these advisors into more complex ones. Fukunaga (2002, 2008) proposes a similar system, CLASS, which combines heuristic building blocks to form composite heuristics for solving SAT problems.

Closely related is the concept of specialising generic building blocks for the problem to solve. This approach is taken in the SAGE system (Strategy Acquisition Governed by Experimentation) (Langley, 1983b,a). It starts with a set of general operators that can be applied to a search state. These operators are refined by making the preconditions more specific based on their utility for finding a solution. The MULTI-TAC (Multi-tactic Analytic Compiler) system (Minton, 1993b,a, 1996) specialises a set of generic heuristics for the constraint problem to solve.

There can be complex restrictions on how the building blocks can be combined. RT-Syn (Smith and Setliff, 1992) for example uses a preprocessing step to determine the possible combinations of algorithms and data structures to solve a software specification problem and then selects the most appropriate combination using simulated annealing.

Another approach is to modify the parameters of parameterised algorithms in the portfolio. This is usually referred to as automatic tuning and not only applicable in the context of algorithm portfolios, but also for single algorithms. The HAP system (Vrakas et al., 2003) automatically tunes the parameters of a planning system depending on the problem to solve. Horvitz et al. (2001) dynamically modify algorithm parameters during search based on statistics collected during the solving process.

Automatic tuning

Automatic tuning and portfolio selection can be treated separately, as done in the Hydra portfolio builder (Xu et al., 2010). Hydra uses ParamILS (Hutter et al., 2007, 2009b) to automatically tune algorithms in a SATzilla portfolio. ISAC (Kadioglu

et al., 2010) uses GGA (Ansótegui et al., 2009) to automatically tune algorithms for clusters of problem instances.

The area of automatic parameter tuning has attracted a lot of attention in recent years. This is because algorithms have an increasing number of parameters that are difficult to tune even for experts and also because of research into dynamic algorithm portfolios that benefits from automatic tuning.

Minton (1996) automatically selects the most promising configuration of heuristics to solve constraint problems. All possible configurations are enumerated and the most promising one is selected using a hill climbing technique. Coy et al. (2001) take a more structured approach and systematically explore the space of possible configurations while avoiding running all of those configurations. They fit regression models to parameter values to interpolate between the extreme values. Terashima-Marín et al. (1999), Fukunaga (2002), Gagliolo et al. (2004), Ansótegui et al. (2009) use genetic algorithms to evolve promising configurations. Birattari et al. (2002) use a racing approach that uses statistical significance tests to determine whether a current best configuration is improved by a candidate. Racing is an idea developed in the Machine Learning community and described in more detail by Maron and Moore (1997).

Adenso-Diaz and Laguna (2006) propose the CALIBRA system. It uses factorial experimental design and local search techniques to find good configurations, but is not guaranteed to do so. The number of parameters that can be tuned is limited to five and interactions between parameters are not taken into account. Sequential parameter optimisation (Preuss and Bartz-Beielstein, 2007, Hutter et al., 2009a) also uses factorial experimental design techniques and fits a response surface model and uses this to determine the next promising parameter configuration to explore. Its main limitation is that parameters can only be optimised for a single problem.

The systems described so far are only of limited suitability for dynamic algorithm portfolios. They either take a long time to find good configurations or are restricted in the number or type of parameters. Few of the approaches mentioned above take interaction between the parameters into account. More recent approaches have focused on overcoming these limitations.

The ParamILS system (Hutter et al., 2007, 2009b) uses techniques based on local search to identify parameter configurations with good performance. The authors address over-confidence (overestimating the performance of a parameter configuration on a test set) and over-tuning (determining a parameter configuration that is too specific). Ansótegui et al. (2009) use genetic algorithms to discover favourable parameter configurations for the algorithms being tuned. The authors use a racing approach to avoid having to run all generated configurations to completion. They also note that one of the advantages of the genetic algorithm approach is that it is inherently parallel.

Both of these approaches are capable of tuning algorithms with a large number of parameters and possible values as well as taking interactions between parameters into account. They are used in practice in the Algorithm Selection systems Hydra

and ISAC, respectively.

Dynamic portfolios are in general a more fruitful area for Algorithm Selection research because of the large space of possible decisions. Static portfolios are usually relatively small and the decision space is amenable for human exploration. This is not a feasible approach for dynamic portfolios though. Minton (1996) notes that

> “MULTI-TAC turned out to have an unexpected advantage in this arena, due to the complexity of the task. Unlike our human subjects, MULTI-TAC experimented with a wide variety of combinations of heuristics. Our human subjects rarely had the inclination or patience to try many alternatives, and on at least one occasion incorrectly evaluated alternatives that they did try.”

3.5. Problem solving with portfolios

There are different ways in which a portfolio can be used to solve a given problem. All of the algorithms can be used or just a subset; they can be switched during the solving process or alternated.

A common case is to select the best algorithm from a portfolio and use it to solve the problem completely. This approach is used for example in SATzilla (Nudelman et al., 2004, Xu et al., 2007b, 2008), ARGOSMART (Nikolić et al., 2009), SALSA (Demmel et al., 2005) and EUREKA (Cook and Varnell, 1997).

In contrast to this approach, Arbelaez et al. (2009) select the best search strategy at every checkpoint in the search tree. Similarly, Brodley (1993) recursively partitions the problem to be solved and potentially selects different algorithms for each partition. In this approach, a lower-level decision can lead to changing the decision at the level above. More fine-grained approaches select the best way to proceed at every node of the search tree. The PRODIGY system (Carbonell et al., 1991) for example selects the next operator to apply in order to reach the goal state of a planning problem at each node. Similarly, Langley (1983a) learn weights for operators that can be applied at each search state and select from among them accordingly.

Closely related is the work by Lagoudakis and Littman (2000, 2001), which partitions the search space into recursive subtrees and selects the best candidate from the portfolio for every subtree. Samulowitz and Memisevic (2007) also select heuristics for solving sub-problems.

Other approaches monitor the performance of an algorithm after it has been selected. Fink (1998) investigates setting a time bound for the algorithm that has been selected. More sophisticated systems furthermore adjust their selection if such a bound is exceeded. Borrett et al. (1996) try to detect behaviour during search that indicates that the algorithm is performing badly, for example visiting nodes in a subtree of the search that clearly do not lead to a solution. If such behaviour is detected, they propose switching the currently running algorithm according to a

fixed replacement strategy. Sakkout et al. (1996) explore the same basic idea. They switch between two algorithms for solving constraint problems that achieve different levels of consistency and show that their approach achieves the same level of consistency as the more expensive algorithm. The cost is significantly lower however as they switch to the cheaper algorithm when it achieves the same level of consistency. Stergiou (2009) also investigates switching propagation methods during solving.

A different line of research computes explicit schedules for running (a subset of) the algorithms in the portfolio. Roberts and Howe (2006) rank algorithms in order of expected performance and allocate time according to this ranking. Pulina and Tacchella (2009) determine a schedule according to predefined strategies. O'Mahony et al. (2008) try to optimise the computed schedule instead of following predefined strategies. Petrik (2005) also computes an optimal schedule. Kadioglu et al. (2011) propose an efficient approach for computing optimal schedules.

Howe et al. (1999) propose a round-robin schedule that contains all algorithms in the portfolio, but varies their order and how much time is allocated to each one. Gerevini et al. (2009) compute round-robin schedules for subsets of all algorithms and choose the best one based on the results of performance simulations. Roberts and Howe (2007) explore different strategies for allocating time to algorithms. Streeter et al. (2007) compute a schedule with the aim of improving the average-case performance. In later work, they compute theoretical guarantees for the performance of their schedule (Streeter and Smith, 2008).

Cicirello and Smith (2005) investigate a model that allocates resources to an algorithm proportional to the number of times it has been successful. In particular, they note that the allocated resources should grow doubly exponentially in the number of successes. Wu and van Beek (2007) approach scheduling the chosen algorithms in a different way and assume a fixed limit on the amount of resources an algorithm can consume while solving a problem. They then compute the optimal schedule based on this.

Petrik and Zilberstein (2006) propose to run the selected algorithms in parallel instead of according to a sequential schedule in what they call a *parallel portfolio*. Despite the name, the authors assume that the portfolio will be run on a single processor and compute shares of compute time to allocate to each one of the algorithms.

In one of the original papers investigating algorithm portfolios, Gomes and Selman (2001) examine the effects of running the portfolio algorithms sequentially and in parallel without computing schedules that govern the resource allocation to each one.

3.5.1. Offline and online approaches

In addition to whether they choose a single algorithm or compute a schedule, existing approaches can also be distinguished by whether they operate before the problem is being solved (offline) or while the problem is being solved (online).

Examples of the approaches that only make offline decisions include Xu et al.

(2008), Minton (1996), Smith and Setliff (1992), O'Mahony et al. (2008). One of the problems is that if the chosen algorithm turns out to have bad performance, there is no way of mitigating this. Purely offline approaches are inherently vulnerable to bad predictions.

The approaches that make decisions during the search (e.g. at every node of the search tree) are necessarily online systems. Examples include Langley (1983a), Arbelaez et al. (2009), Carbonell et al. (1991), Lagoudakis and Littman (2000), Samulowitz and Memisevic (2007). A different class of online systems monitors the performance of a chosen method and makes a new decision if the actual performance does not match the expected performance. Examples of this approach include Borrett et al. (1996), Sakkout et al. (1996), Stergiou (2009).

A third class of systems not only dynamically updates the decisions made, but also adjusts the Algorithm Selection model based on the observed performance. Pulina and Tacchella (2009) retrain the prediction engine if the actual run time is much longer than the predicted runtime. Gagliolo et al. (2004), Gagliolo and Schmidhuber (2005, 2006b) take this approach to the extreme and learn the Algorithm Selection model only dynamically while the problem is being solved. Armstrong et al. (2006) also rely exclusively on an online selection model.

There are a number of approaches that combine an offline selection step and an online monitoring/adaption step. An example for this is Pulina and Tacchella (2009), but also Horvitz et al. (2001) where the offline selection of an algorithm is combined with the online adjusting of its parameters. Carchrae and Beck (2004) train different models for selecting the best algorithm to start with offline and predicting whether to switch the algorithm online.

3.6. Portfolio selectors

The key component of an algorithm portfolio is the mechanism to select a subset of the algorithms for solving a particular problem. Apart from accuracy, one of the main requirements for such a selector is that it is relatively cheap to run – if selecting an algorithm for solving a problem is more expensive than solving the problem, there is no point in doing so. Vassilevska et al. (2006) explicitly define the selector as "an efficient (polynomial time) procedure". They also note that,

> "While it seems that restricting a heuristic to a special case would likely improve its performance, we feel that the ability to partition the problem space of some $\mathcal{NP}$-hard problems by efficient selectors is mildly surprising."

There are technical challenges associated with making selectors efficient as well. Algorithm Selection systems that analyse the problem to be solved, such as SATzilla, need to take steps to ensure that the analysis does not become too expensive. Two such measures are the running of a pre-solver and the prediction of the time required

to analyse a problem (Xu et al., 2008). The idea behind the pre-solver is to choose an algorithm with reasonable general performance from the portfolio and use it to start solving the problem before starting to analyse it. If the problem happens to be very easy, it will be solved even before the results of the analysis are available. After a fixed time, the pre-solver is terminated and the results of the Algorithm Selection system are used. Predicting the analysis time is a closely related idea designed to lift the limitation of having a fixed time for running the pre-solver. If the predicted required analysis time is too high, a default algorithm with reasonable performance is chosen and run on the problem. This technique is particularly important in cases where the problem is hard to analyse, but easy to solve.

There are many different approaches to how such selectors operate. It is not necessarily an explicit part of the system. Minton (1996) compiles the Algorithm Selection system into a LISP programme for solving the original constraint problem. The selection rules are part of the programme logic. Fukunaga (2008), Garrido and Riff (2010) evolve selectors and combinators of heuristic building blocks using genetic algorithms.

3.6.1. Performance models

The way the selector operates is closely linked to the way the performance model of the algorithms in the portfolio is built. In early approaches, the performance model was usually not learnt but given in the form of human expert knowledge. Borrett et al. (1996), Sakkout et al. (1996) use hand-crafted rules to determine whether to switch the solution method during solving. Allen and Minton (1996) also have hand-crafted rules for estimating the runtime performance of an algorithm. Modern approaches sometimes use only human knowledge as well. Tolpin and Shimony (2011) for example model the performance space using statistical methods and use this hand-crafted model to select a heuristic for solving constraint problems.

A more common approach today is to automatically learn performance models using Machine Learning on training data. The portfolio algorithms are run on a set of representative problems and based on these experimental results, performance models are built. This approach is used by Xu et al. (2008), Pulina and Tacchella (2007), O'Mahony et al. (2008), Kadioglu et al. (2010), Guerri and Milano (2004), to name but a few examples. A drawback of this approach is that the training time is usually large. Gagliolo and Schmidhuber (2006a) investigate ways of mitigating this problem by using censored sampling, which introduces an upper bound on the runtime of each experiment.

In some cases, no explicit performance models are used at all. Caseau et al. (1999) and Minton (1996) run the candidate heuristics on a set of test problems and select the one with the best performance that way for example.

Per-portfolio models

Automated approaches learn a performance model of the entire portfolio based on training data. This is used for example by O'Mahony et al. (2008), Cook and Varnell (1997), Pulina and Tacchella (2007), Nikolić et al. (2009), Guerri and Milano (2004). Again there are different ways of doing this. Lazy approaches do not learn an explicit model, but use the set of training examples as a case base. For new problems, the closest problem in the case base is determined and decisions made accordingly. Wilson et al. (2000), Pulina and Tacchella (2007), O'Mahony et al. (2008), Nikolić et al. (2009), Gebruers et al. (2004) for example use nearest-neighbour classifiers to achieve this. Explicitly-learned models try to identify the concepts that affect performance for a given problem. This acquired knowledge can be made explicit to improve the understanding of the researchers of the problem domain. Carbonell et al. (1991), Gratch and DeJong (1992), Brodley (1993), Vrakas et al. (2003) learn classification rules that guide the selector. Vrakas et al. (2003) note that the decision to use a classification rule learner was not so much guided by the performance of the approach, but the easy interpretability of the result. Langley (1983a), Epstein et al. (2002), Nareyek (2001) learn weights for decision rules. Cook and Varnell (1997), Guerri and Milano (2004), Guo and Hsu (2004), Roberts and Howe (2006), Bhowmick et al. (2006) go one step further and learn decision trees. Guo and Hsu (2004) again note that the reason for choosing decision trees was not primarily the performance, but the understandability of the result.

Some approaches learn probabilistic models that take uncertainty and variability into account. Gratch and DeJong (1992) use a probabilistic model to learn control rules. Demmel et al. (2005) learn multivariate Bayesian decision rules. Carchrae and Beck (2004) learn a Bayesian classifier to predict the best algorithm after a certain amount of time. Stern et al. (2010) consider algorithm portfolios in a framework of Bayesian models. Domshlak et al. (2010) learn decision rules using naïve Bayes classifiers. Lagoudakis and Littman (2000), Petrik (2005) learn performance models based on Markov Decision Processes.

Other approaches include support vector machines (Hough and Williams, 2006, Arbelaez et al., 2009), reinforcement learning (Armstrong et al., 2006), neural networks (Gagliolo and Schmidhuber, 2005), decision tree ensembles (Hough and Williams, 2006), boosting (Bhowmick et al., 2006), multinomial logistic regression (Samulowitz and Memisevic, 2007) and clustering (Stamatatos and Stergiou, 2009, Stergiou, 2009, Kadioglu et al., 2010). Streeter et al. (2007) compute schedules for running the algorithms in the portfolio based on a statistical model of the problem instance distribution.

Per-algorithm models

A different approach is to learn performance models for the individual algorithms of the portfolio. The performance predicted on a problem can then be compared and the selector can proceed based on this. Models for each algorithm in the portfolio are

used for example by Xu et al. (2008), Howe et al. (1999), Allen and Minton (1996), Lobjois and Lemaître (1998), Gagliolo and Schmidhuber (2006b).

A common way of doing this is to use regression to predict the performance of each algorithm. This is used by Xu et al. (2008), Howe et al. (1999), Leyton-Brown et al. (2002), Haim and Walsh (2009), Roberts and Howe (2007). Silverthorn and Miikkulainen (2010) learn latent class models of unobserved variables. Weerawarana et al. (1996) use Bayesian belief propagation and neural nets to predict the runtime of a particular algorithm on a particular problem. Sillito (2000) uses sampling methods to estimate the cost of solving constraint problems. Watson (2003) models the behaviour of local search algorithms with Markov chains.

Another common approach is to build statistical models of an algorithm's performance based on past observations. Fink (1998) computes the expected gain for time bounds based on past success times. The computed values are used to choose the algorithm and the time bound for running it. Brazdil and Soares (2000) compare algorithm rankings based on different past performance statistics. Similarly, Leite et al. (2010) maintain a ranking based on past performance. Cicirello and Smith (2005) propose a bandit problem model that governs the allocation of resources to each algorithm in the portfolio. Gerevini et al. (2009) use the past performance of algorithms to simulate the performance of different algorithm schedules and use statistical tests to select one of the schedules.

Hierarchical models

There are some approaches that combine several models into a hierarchical performance model. Xu et al. (2007a) use sparse multinomial logistic regression to predict whether a SAT problem instance is satisfiable and, based on that prediction, use a logistic regression model to predict the runtime of each algorithm in the portfolio. Haim and Walsh (2009) take the same approach with a portfolio of algorithms of different types.

Hierarchical models are only applicable in a limited number of scenarios, which explains the comparatively small amount of research into them. For many application domains, only a single property needs to be predicted.

Selection of model learner

As described in the previous sections, there are many different ways of learning algorithm performance models. Some of the research mentioned compared different methods of doing so.

Xu et al. (2008) mention that, in addition to the chosen ridge regression for predicting the runtime, they explored using lasso regression, support vector machines and Gaussian processes. Cook and Varnell (1997) compare different decision tree learners, a Bayesian classifier, a nearest neighbour approach and a neural network. Leyton-Brown et al. (2002) compare several versions of linear and non-linear regression. Guo and Hsu (2004) explore using decision trees, naïve Bayes rules, Bayesian networks

and meta-learning techniques. Gebruers et al. (2005) compare nearest neighbour classifiers, decision trees and statistical models. Hough and Williams (2006) use decision tree ensembles and support vector machines. Bhowmick et al. (2006) investigate alternating decision trees and various forms of boosting, while Pulina and Tacchella (2007) use decision trees, decision rules, logistic regression and nearest neighbour approaches. Roberts and Howe (2007) use 32 different Machine Learning algorithms to predict the runtime of algorithms and probability of success. They attempt to provide explanations for the performance of the methods they have chosen in Roberts et al. (2008). Silverthorn and Miikkulainen (2010) compare the performance of different latent class models.

However, only Guo and Hsu (2004), Gebruers et al. (2005), Hough and Williams (2006), Pulina and Tacchella (2007), Silverthorn and Miikkulainen (2010) quantify the differences in performance of the methods they used. The other comparisons give only qualitative evidence.

3.6.2. Selector predictions

Apart from many different ways of creating performance models of the portfolio, there are also different predictions the performance model can make to inform the decision of the selector of a subset of the portfolio algorithms. The prediction can be a single categorical value – the algorithm to choose. This type of prediction is made for example by O'Mahony et al. (2008), Cook and Varnell (1997), Pulina and Tacchella (2007), Nikolić et al. (2009), Guerri and Milano (2004).

A different approach is to predict the runtime of the individual algorithms in the portfolio. For example Horvitz et al. (2001), Petrik (2005), Silverthorn and Miikkulainen (2010) do this. Xu et al. (2008) do not predict the runtime itself, but the logarithm of the runtime. They note that,

> “In our experience, we have found this log transformation of runtime to be very important due to the large variation in runtimes for hard combinatorial problems.”

Allen and Minton (1996) estimate the runtime by proxy by predicting the number of constraint checks. Lobjois and Lemaître (1998) estimate the runtime by predicting the number of search nodes to explore and the time per node. Lagoudakis and Littman (2000) talk of the “cost” of selecting a particular algorithm, Nareyek (2001) of the “utility” and Tolpin and Shimony (2011) of the “value of information” of selecting an algorithm. Xu et al. (2009) predict the penalized average runtime score, a measure that combines runtime and possible timeouts.

Borrett et al. (1996), Sakkout et al. (1996), Carchrae and Beck (2004) predict when to switch the algorithm used to solve a problem. Brazdil and Soares (2000), Soares et al. (2004), Leite et al. (2010) produce rankings of the portfolio algorithms. Howe et al. (1999), Gagliolo et al. (2004), Gagliolo and Schmidhuber (2006b), Roberts and Howe (2006), O'Mahony et al. (2008) predict schedules. The primary selection

criteria for Soares et al. (2004) and Leite et al. (2010) is the quality of the solution an algorithm produces.

In addition to the primary selection criteria, a number of approaches predict secondary criteria as well. Howe et al. (1999), Fink (1998), Roberts and Howe (2007) predict the probability of success for each algorithm. Weerawarana et al. (1996) predict the quality of a solution. Another kind of prediction is not concerned with the selection of a subset of the portfolio algorithms, but their runtime behaviour. Horvitz et al. (2001) and Hutter et al. (2006) predict parameter values to use for the chosen algorithm. Predictions can be made for other things in the context of Algorithm Selection. Gebruers et al. (2005), Little et al. (2002), Borrett and Tsang (2001) consider selecting the most appropriate model of a constraint problem. Smith and Setliff (1992), Brewer (1995), Wilson et al. (2000) predict algorithms and data structures to be used in a software system.

3.7. Features

Algorithm Selection systems employ problem features to inform the decision which subset of the portfolio to use for solving a given problem. The most general feature is the performance of an algorithm observed on a set of past problems. Cicirello and Smith (2005), Streeter et al. (2007), Silverthorn and Miikkulainen (2010) use only this feature. Other approaches use more fine-grained measures of past performance, for example Langley (1983b), Minton (1996).

Most approaches learn models for the performance on particular problems and do not use past performance as a feature, but to inform the prediction to be made. They consider features of the problem to be solved, for example O'Mahony et al. (2008), Pulina and Tacchella (2007), Weerawarana et al. (1996), Howe et al. (1999), Xu et al. (2008). Other sources of features include the domain of the problem to be solved (Carbonell et al., 1991), the generator that produced the problem to be solved (Horvitz et al., 2001), the runtime environment (Armstrong et al., 2006), structures derived from the problem such as the primal graph of a constraint problem (Gebruers et al., 2004, Guerri and Milano, 2004), specific parts of the model such as variables (Epstein and Freuder, 2001) or the algorithms in the portfolio themselves (Hough and Williams, 2006). Gerevini et al. (2009) rely on the problem domain as the only problem-specific feature and select based on past performance data for that domain. Beck and Fox (2000) consider not only the values of features of a problem, but the changes of those values while the problem is being solved. Smith and Setliff (1992) consider features of abstract representations of the algorithms.

The features based on the problem to be solved and similar things can be computed *statically*, i.e. before an algorithm is selected. Another common approach is to probe the search space and derive features from the observations made. Examples for this *semi-static* approach are Allen and Minton (1996), Cook and Varnell (1997), Lobjois and Lemaître (1998), Beck and Freuder (2004), Stamatatos and Stergiou (2009). The probing usually requires a preprocessing phase where parts of the problem are being

explored by one or more algorithms.

A third class of features is computed *dynamically* after an algorithm has been selected during solving. Approaches that rely purely on such features are for example Borrett et al. (1996), Sakkout et al. (1996), Nareyek (2001), Stergiou (2009), Domshlak et al. (2010).

The type of feature used determines how the portfolio selector is used for solving problems (cf. Section 3.5 on page 35). Many Algorithm Selection systems use several different types of features, for an overview see Table A.1 on page 124.

The features used for learning the Algorithm Selection model are crucial to its success. Uninformative features might prevent the model learner from recognising the real correlation between problem and performance or the most important feature might be missing. Many researchers have recognised this problem.

Howe et al. (1999) manually select the most important features. They furthermore take the unique approach of learning one model per feature for predicting the probability of success and combine the predictions of the models. Leyton-Brown et al. (2002), Xu et al. (2008) perform automatic feature selection by greedily adding features to an initially empty set. In addition to the basic features, they also use the pairwise products of the features. Pulina and Tacchella (2007) also perform automatic greedy feature selection, but do not add the pairwise products. Wilson et al. (2000) use genetic algorithms to determine the importance of the individual features. Petrovic and Qu (2002) evaluate subsets of the features they use and learn weights for each of them. Roberts et al. (2008) consider using a single feature and automatic selection of a subset of all features. Guo and Hsu (2004) and Kroer and Malitsky (2011) also use techniques for determining the most predictive subset of features.

Beck and Freuder (2004), Carchrae and Beck (2004, 2005) explicitly focus on features that do not require a lot of domain knowledge. Beck and Freuder (2004) note that,

> “While existing algorithm selection techniques have shown impressive results, their knowledge-intensive nature means that domain and algorithm expertise is necessary to develop the models. The overall requirement for expertise has not been reduced: it has been shifted from algorithm selection to predictive model building.”

Their approach uses a number of features that are applicable across a wide range of problems and require no expert knowledge of the specific domain.

It is not only important to use informative features, but also features that are cheap to compute. If the cost of computing the features and making the decision is too high, the performance improvement from selecting the best algorithm might be eroded. Xu et al. (2009) predict the feature computation time for a given problem and fall back to a default selection if it is too high to avoid this problem. Bhowmick et al. (2009) consider the computational complexity of calculating problem features when selecting the features to use. They show that while achieving comparable accuracy to the full set of features, their method is significantly cheaper.

3.8. Application domains

Over the years, Algorithm Selection systems have been used in many different application domains. These range from Mathematics, e.g. differential equations (Kamel et al., 1993, Weerawarana et al., 1996), linear algebra (Demmel et al., 2005) and linear systems (Bhowmick et al., 2006, Kuefler and Chen, 2008), to the selection of algorithms and data structures in software design (Smith and Setliff, 1992, Brewer, 1995, Wilson et al., 2000). The most common application domain by far however is search problems such as SAT (Xu et al., 2008, Lagoudakis and Littman, 2001, Silverthorn and Miikkulainen, 2010), constraints (Minton, 1996, Epstein et al., 2002, O'Mahony et al., 2008), quantified Boolean formulae (Pulina and Tacchella, 2009, Stern et al., 2010), planning (Carbonell et al., 1991, Howe et al., 1999, Vrakas et al., 2003), scheduling (Beck and Fox, 2000, Beck and Freuder, 2004, Cicirello and Smith, 2005), combinatorial auctions (Leyton-Brown et al., 2002, Gebruers et al., 2004, Gagliolo and Schmidhuber, 2006b) and general search algorithms (Langley, 1983b, Cook and Varnell, 1997, Lobjois and Lemaître, 1998).

Less common applications include Machine Learning (Soares et al., 2004, Leite et al., 2010), genetic algorithms (Gagliolo et al., 2004, Gagliolo and Schmidhuber, 2005) and the most probable explanation problem (Guo and Hsu, 2004).

Most of the techniques presented in papers are not limited to the application domains that they are evaluated on. In some cases however, researchers have tailored their approach to the domain they applied it to; for example in the case of hierarchical models for SAT (cf. Section 3.6.1 on page 40).

3.9. Methodology example – SATzilla

There is a lot of literature and different approaches to solving the Algorithm Selection Problem. One of the most prominent ones is SATzilla (Xu et al., 2008, 2009). To illustrate the basic process, SATzilla's approach is recast in the framework for solving Algorithm Selection problems described by Rice (1976) below (cf. Section 3.1 on page 26).

3.9.1. Formulation

Determination of the subclasses of problems and mappings to be used.

SATzilla is an algorithm portfolio selector for SAT problems. The space of all SAT problems is further manually partitioned into random, hand-crafted and industrial problems. The mapping is from SAT solver in the static portfolio to performance score on a particular problem instance. There are 19 solvers in the portfolio, but smaller subsets are chosen manually for each category of problems.

3.9.2. Existence

Does a best selection mapping exist?

There is a best selection mapping that always chooses the portfolio solver that has the best performance on a given problem. Given the runtime performance of each solver on each problem, this mapping can be determined easily. The runtime performance can be determined easily as well as SATzilla uses a static portfolio. For unseen problems, the best mapping cannot be determined without running each solver, but nevertheless this mapping obviously exists.

3.9.3. Uniqueness

Is there a unique best selection mapping?

Not necessarily. Several solvers could have the same performance on a problem and that performance could be the best. However, this question is irrelevant for SATzilla – if two solvers have the same performance on a problem, both are equally valid choices.

3.9.4. Characterisation

What properties characterise the best selection mapping and serve to identify it?

Like many other approaches, SATzilla uses Machine Learning to identify the mapping. It employs 91 features to characterise a problem to solve. The number of features is increased by quadratic feature expansion, where new features are formed by multiplying pairs of feature values for all possible pairs. The most predictive subset of this large set of features is selected by greedily adding features to an initially empty set until the predictive accuracy does not improve any more.

3.9.5. Computation

What methods can be used to actually obtain the best selection mapping?

SATzilla learns a ridge regression model of the solver performance. The model focuses on hard problems and disregards problems that are solved in less than a second. A pre-solver has the task of solving those problems. The rationale behind this approach is that feature extraction and performance prediction would take more time than actually solving the problem for these small problems. To mitigate this problem, SATzilla also predicts the time required to compute the features of a given problem and falls back to a manually chosen solver from the portfolio if it is too high.

The ridge regression model for each solver is learned offline from training data

taken from SAT competitions, where SAT solvers try to solve as many problems from a given set as quickly as possible. At runtime, the learned model is used to predict the performance score of each candidate solver. A single solver to run on the problem is chosen based on these predictions. The predictions are not checked for correctness and the model is not adjusted online.

3.10. Summary

A survey of the literature shows that over the years there have been many approaches to solving the Algorithm Selection Problem. Most of the time, these involve some kind of Machine Learning. This is not a surprise, as the relationship between an algorithm and its performance is often complex and hard to describe formally. In many cases, even the designer of an algorithm does not have a general model of its performance.

Smith-Miles (2009) presents a comprehensive survey on Machine Learning efforts to tackle the Algorithm Selection Problem. She takes a different perspective on Algorithm Selection and the literature relevant to this dissertation is mentioned in this chapter. Nevertheless the survey presents interesting additional information, such as an overview of the number of algorithms and the size of data sets of many approaches.

Table A.1 on page 124 presents more detail for each individual paper surveyed in this chapter. The table summarises and classifies the papers according to the criteria used in the previous sections. It serves as an overview as well as a reference for the literature on Algorithm Selection.

A researcher who wants to do Algorithm Selection for their purposes is faced with a plethora of different ways and methods that have been explored previously. It is not clear what the most suitable one or the one with the best performance is. There is also a lack of detailed evaluation of the different approaches – how likely is a system to make costly selection mistakes?

In many practical applications, it may not even be clear whether effective and efficient Algorithm Selection can be performed at all. Perhaps the underlying relationship between the characteristics of the problem and performance of an algorithm is too complicated to be learned or represented. The features of a problem may be so expensive to compute that then being able to determine the best algorithm might not yield a performance benefit at all.

The next two chapters look at case studies that explore exactly this problem. Can we perform Algorithm Selection for a problem in a specific domain effectively and efficiently and if so, how?

4

Learning when to use lazy learning in constraint solving

After surveying the Algorithm Selection literature in the previous chapter, we select a subset of the proposed techniques and apply them to a specific scenario – selecting whether to use lazy learning in constraint solving or not. This is a simple, binary decision that should be relatively easy to make. This case study serves as an entry point into the investigations into Algorithm Selection presented in this dissertation and touches on the challenges that are tackled.

4.1. Introduction and background

In constraint programming, *propagation* is used as a means of reducing the search space and finding a solution faster. Instead of exploring a search path, the solver can reason about it and decide that it is futile to explore because it cannot be part of a solution. Consider for example a set of three variables, each of which can be assigned one of two distinct values. If the constraints require each variable to have an assignment different from the other variables, propagation can rule out this part of the search tree as there cannot be such an assignment. It is not necessary to try assigning all possible values to each variable to see if a constraint is violated.

Search can be further improved by the use of a lazy learning algorithm (Katsirelos, 2009, Katsirelos and Bacchus, 2003, 2005, Gent et al., 2010), where previously unknown constraints are uncovered during search and used to speed up search subsequently. It is extremely efficient on some types of problems, but has a negative effect on others. Therefore, it is desirable to know beforehand whether or not lazy learning is expected to be useful.

Learning in constraints is a means of discovering new constraints during search whenever the solver reaches a state where it cannot proceed further. The decisions

The material in this chapter has been published previously in: Ian Gent, Chris Jefferson, Lars Kotthoff, Ian Miguel, Neil Moore, Peter Nightingale, and Karen Petrie. Learning When to Use Lazy Learning in Constraint Solving. In *19th European Conference on Artificial Intelligence*, pages 873–878, August 2010.
The contributions of the author of this dissertation are listed on page xix et seqq.

and propagation performed earlier in the search are analysed and a constraint that prohibits the variable assignments and disassignments that lead to the current failure are added. Lazy learning adds such constraints as late as possible during search instead of as soon as the failure occurs to improve the overall efficiency. The power of constraint learning comes from the combination of the learned constraints – while each individual constraint will only save a small amount of work in cases where a dead end similar to a previous one is encountered, the set of learned constraints can dramatically increase the amount of search that propagation can rule out. For more details on lazy learning in constraint solving, see Moore (2011).

This is a typical Algorithm Selection Problem. We have the choice between a standard constraint solver and a constraint solver that does lazy learning and, for each problem to be solved, are to select the most appropriate one. Our aim in this chapter is, apart from achieving performance improvements, to further our understanding of the relatively new technique of lazy learning. To this end, we employ a decision tree learner to distinguish between problems where lazy learning performs well and where not.

Decision trees are used in many approaches to the Algorithm Selection Problem because they are simple and easy to understand, e.g. by Guerri and Milano (2004) and Guo and Hsu (2004).

4.2. Evaluation problems

The training and test data for the decision tree to decide whether or not to use lazy learning comprises a set of 2028 constraint problem instances from 46 different problem classes. A complete list of problem classes can be found in Appendix E on page 149. The set has been chosen to include as many problems as possible, regardless of our expectations as to whether lazy learning will perform well.

For our experiments, we used the lazy learning variant of Minion, which we call Minion-lazy. The reference constraint solver used is Minion (Gent et al., 2006a) version 0.9. A comparison of performance between Minion and Minion-lazy is given in Figure 4.1 on the facing page. We used binaries compiled with g++ version 4.4.1 and Boost version 1.38.0. The experiments were run on machines with dual quad-core Intel E5430 2.66 GHz, 8 GB RAM running CentOS with Linux kernel 2.6.18-164.6.1.el5 64 Bit.

We imposed a time limit of 5000 seconds per problem for solving. On 11 problems, Minion-lazy ran out of memory (>4 GB) before it was able to solve the problem or reach the time limit. The total number of problems that neither solver could solve because of a time out or memory issues was 255. Both solvers took the same time on 4 problems that they could both solve.

The problems, the binaries to run them, and everything else required to reproduce our results is available at `http://www.cs.st-andrews.ac.uk/~larsko/ecai2010/learning.tar.bz2`.

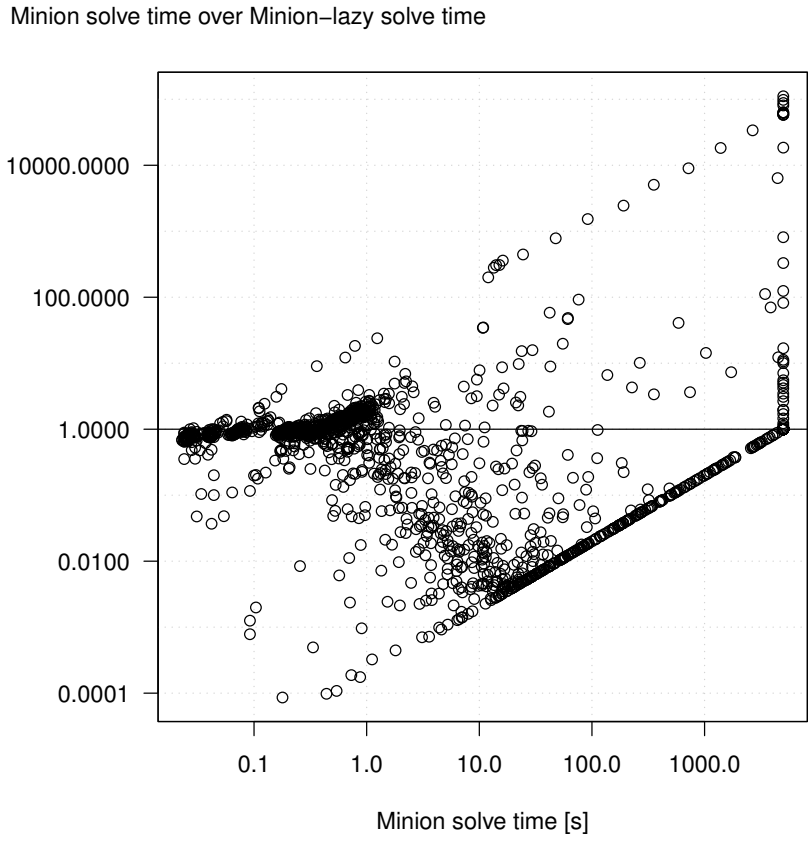


Figure 4.1. Runtime comparison for Minion-lazy vs Minion. Each point is a result for a single CSP. The x-axis is the solve time for Minion. The y-axis gives the speedup from using Minion-lazy instead of Minion. A ratio of 1 means they were the same, above 1 means Minion-lazy was faster and below 1 that Minion was faster.

4.3. Problem features and their measurement

We measured 85 features of the problem instances. They describe a wide range of properties, such as the number of constraints and variables used, a breakdown of the individual constraint and variable types and a number of features based on the primal graph. The primal graph $g = \langle V, E \rangle$ of a constraint problem has a vertex for every variable and two vertices are connected by an edge if and only if the two variables are in the scope of a constraint together.

Not all the features are likely to be useful for predicting what choice of solver to make. We describe the features that reflect the structure of a constraint problem below. In particular those features do not depend on particular solver properties or are specific to problem classes.

Edge density The number of edges in g divided by the number of pairs of distinct vertices.

Clustering coefficient For a vertex v, the set of neighbours of v is $n(v)$. The edge density among the vertices $n(v)$ is calculated. The clustering coefficient is the mean average of this local edge density for all v (Watts and Strogatz, 1998). It is intended to be a measure of the local cliqueness of the graph.

Normalised degree The normalised degree of a vertex is its degree divided by $|V|$. The mean and median normalised degree were used.

Normalised standard deviation of degree The standard deviation of vertex degree is normalised by dividing by $|V|$.

Width of ordering Each of our problems has an associated variable ordering. The width of a vertex v in an ordered graph is its number of *parents* (i.e. neighbours that precede v in the ordering). The width of the ordering is the maximum width over all vertices (Dechter (2003), Chapter 4). The width of the ordering and the width normalised by the number of vertices were used.

Width of graph The width of a graph is the minimum width over all possible orderings. The width of the graph and the width normalised by the number of vertices were used.

Multiple shared variables The proportion of pairs of constraints that share more than one variable.

Normalised mean constraints per variable For each variable, we count the number of constraints on the variable. The mean average is taken, and this is normalised by dividing by the number of constraints.

Normalised SAC literals The number of literals pruned by singleton arc consistency preprocessing (Debruyne and Bessière, 1997), as a proportion of all literals. Enforcing singleton arc consistency is a way of removing the variable assignments that cannot be part of a solution.

Ratio of auxiliary variables to other variables Auxiliary variables are introduced by decomposition of expressions in order to be able to express them in the language of the solver. We used the ratio of auxiliary variables to other variables.

Mean tightness The tightness of a constraint is the proportion of variable assignments that are not part of a solution to the total possible variable assignments. It is estimated by sampling 1000 random variable assignments (that are valid w.r.t. variable domains) and testing if they satisfy the constraint. The mean tightness over all constraints was used.

Literal tightness To measure the tightness of a literal (a variable-value pair) w.r.t. a particular constraint, we sample 100 random variable assignments containing the literal and test if it satisfies the constraint. The tightness of a literal is the mean of its tightness in all constraints on that literal. The mean literal tightness – the mean average of the tightness for each literal – and the standard deviation of the literal tightness divided by the mean literal tightness were used (a.k.a. the coefficient of variation).

Proportion of interchangeable variables In many CSPs, the variables form equivalence classes where the number and type of constraints a variable is in are the same. For example in the CSP $x_1 \times x_2 = x_3, x_4 \times x_5 = x_6$, x_1, x_2, x_4, x_5 are all interchangeable, as are x_3 and x_6. The first stage of the algorithm used by Nauty (McKay, 1981) detects this property. Given a partition of n variables generated by this algorithm, we transform this into a number between 0 and 1 by taking the proportion of all pairs of variables which are in the same part of the partition.

The features used in addition to the ones described in detail above are the number of variables, the number of Boolean variables, the number of discrete variables, the number of bound variables, the number of sparse bound variables, the mean value and the 0th, 25th, 50th, 75th and 100th percentile of the variable domains, the ratio of discrete to Boolean variables, the ratio of variables with domain size equal to 2 to variables with domain size not equal to 2, the number of auxiliary variables, the number of other variables, the number of constraints, the mean, the normalised mean and the 0th, 25th, 50th, 75th and 100th percentile of constraint arities, the absolute number and proportion of the individual constraints and the number and proportion of nullary, unary, binary and ternary constraints. Some of the additional features are based on the ones described in detail and include non-normalised versions or percentiles of those.

Wherever possible, we normalised features that would be specific to problem instances of a particular size, such as the number of variables. This is based on the intuition that similar problems of different sizes are likely to behave similarly with lazy learning. We also included non-normalised features to capture effects related to the size of a problem instance however.

Of the 2028 problems from the problem set, 93 problems could not be analysed because of insufficient memory or another error in the analyser.

4.4. Constructing a problem classifier

The training data was used to induce a decision tree offline for classifying a given problem. After having constructed and evaluated an initial decision tree, we proceeded to tune the inducer to learn simpler trees. While the performance of the classifier decreases, it is more likely to perform well on unseen problems and easier to understand. As one of the aims of this investigation is to gain a better understanding of the factors that affect the performance of lazy learning, this last point was crucial for us.

4.4.1. Methodology

The experimental data were post-processed in several ways to mitigate two main problems.

- Empirical runtime data is inherently noisy. If the observed difference between the two solvers is too small, we cannot be sure which solver was faster.
- On some problems, the difference between making the right and wrong decision affects performance very significantly. We want those problems to be more important when inducing the decision tree as the penalty for getting the decision wrong is high.

To achieve this, we did the following. First, we calculated the misclassification penalty as the absolute difference in solve time between the two solvers for each problem. Instances where both solvers timed out were not considered. For problems where one of the solvers timed out, we used the timeout (5000 seconds) as the time for that solver, and calculated the misclassification penalty as before. The resulting value is an underestimate of the true misclassification penalty.

Second, we determined which solver to use for each problem. If the difference between both solvers was less than 20% of the time that Minion took, we set the value to "don't know". This was simply to account for differences in run time that were caused by external effects. For the problems where Minion-lazy ran out of memory, we set the value to "use Minion". Based on this, we were able to choose one of the two solvers for 1012 problems.

Each problem was then weighted by the misclassification penalty. We did this to bias the Machine Learning algorithms towards the problems where we can gain or lose a lot – if the penalty for choosing the wrong solver is low, we do not care as much if we make the wrong decision. To achieve this kind of cost-sensitive classification, it is standard practice in Machine Learning to duplicate instances in the training data (Witten et al., 2011). We duplicated each problem $\lceil \log_2(\texttt{misclassification penalty}) \rceil$

times. This means that problems with a misclassification penalty of less than 2 seconds appear once and ones with a penalty of 5000 seconds (the maximum) appear 13 times. We chose this particular function because we did not want problems with a very high penalty to have too much effect and we did want each problem to appear at least once in the data set. The problems where Minion-lazy ran out of memory were not duplicated.

These data were used as input for the WEKA Machine Learning suite (Hall et al., 2009) to learn a classifier that, given a problem, tells us which solver to use. For problems where we did not have values for all features or some of them were unclear, for example when we were not able to make a decision between choosing Minion and Minion-lazy, we used a question mark. This designates an unknown value in WEKA.

The WEKA decision tree inducer we used was `J48`. It implements the well-established C4.5 algorithm (Quinlan, 1993). We did not tune the parameters of `J48` unless noted otherwise, as the default values learned a decision tree with good performance already. In all cases we trained the classifiers to predict the binary decision of whether to use Minion or Minion-lazy.

We evaluated the learned classifiers on the entire set of problems, making the decision which solver to use for each one. If we were unable to make a decision because of missing features, we selected Minion as the default solver. The results of this evaluation are different from the ones we get in WEKA because we use the data set that contains each problem only once – we train and test on the data set that is biased towards problems where a lot can be gained or lost and we evaluate on the unbiased original set. The results are summarised in Table 4.1 on page 55.

4.4.2. Selecting a feature set

Finding a small and appropriate set of features is crucial to the efficiency of classifying an instance. The values of all features must be computed. Some feature values are expensive to compute and may not contribute much to the predictive power of the whole set. We describe four classifiers built with progressively smaller sets of features. We demonstrate that the final classifier, learned with a set of only three features, is almost as accurate as the first one, learned with the full set of 85 features.

Classifier 1 – initial classifier

We built a decision tree using the whole data set with 85 features as described above. The tree had 61 nodes and we achieved 99.7% correctly classified problems with a precision of 99.7% and a recall of 99.7%. For comparison, always using standard Minion gives 86.3% correctly classified problems, a precision of 74.4% and a recall of 86.3%.

The performance of the learned classifier on the set of all problem instances is shown in Table 4.1 on page 55 in the row labelled “Classifier 1”. This demonstrates the feasibility of our approach for this Algorithm Selection Problem.

Classifier 2 – reduced number of features

Based on the encouraging results we achieved with classifier 1, we removed all but the 17 features we believe to reflect the structure of the underlying constraint problem described in detail in Section 4.3 on page 50 and reran the decision tree inducer. Computing a large number of features for each instance is expensive and might outweigh the benefit of having a classifier and being able to select the faster solver. We also want to eliminate the influence of features which are specific to Minion, a problem class, or a particular problem size.

The classifier learned from this data set showed a similar performance to the first one. The decision tree had 57 nodes and 99.6% of the problems were classified correctly. Precision and recall were 99.6% again as well. Table 4.1 on the facing page however shows that the overall performance is lower than the one of the previous classifier, although the difference is small. We concluded that we could reduce the number of features used in making the decision without a significant decrease in quality.

Classifier 3 – most predictive features

The `J48` decision tree learner does not necessarily use all the features in the learned decision tree. None of the previous classifiers used all the features. We decided to further reduce the number of features used in the decision tree by using the WEKA `AttributeSelectedClassifier` meta-learning algorithm with `CfsSubsetEval` and exhaustive search.

Each feature was assessed with respect to its predictive ability and the degree of redundancy within the current set of features. This evaluation was performed for all subsets of the set of 17 features used in classifier 2. After this step, only three features remained – normalised width of ordering (NWO), normalised mean constraints per variable (NMCV) and mean tightness (MT).

The decision tree built with these three features had 79 nodes, 99.6% correctly classified problems and precision and recall of 99.6%. As Table 4.1 on the next page shows, the performance is similar to classifier 3.

4.4.3. Towards a simple decision tree

The decision tree of classifier 3 has 79 nodes, even though it uses only three features. The interactions between the features and their effect on performance are not easy to understand. Furthermore, the decision tree appears to be overfitted. For most paths from the root to a leaf node, it switches several times on a single feature. On one branch, it matches the intervals $[0, 6.79\%](6.79\%, 6.81\%](6.81\%, 23.59\%](23.59\%, 100\%)$ of mean tightness (MT) to decisions *Minion-lazy*, *Minion*, *Minion-lazy*, *Minion* respectively. The interval $(6.79\%, 6.81\%]$ is a clear case of overfitting – it is very narrow and contains only one problem from the original data set. The problem occurs 4 times in the training set because of its misclassification penalty.

	solved	total time [s]	accuracy	> 20% difference misclas-sification	penalty [s]	compute features [s]
oracle	1,773	194,372	-	0	0	-
anti-oracle	1,485	1,805,732	-	1,046	1,611,360	-
Minion	1,736	360,949	86.3%	292	166,577	0
Minion-lazy	1,522	1,639,155	13.7%	754	1,444,783	0
Classifier 1	1,769	201,726	99.7%	30	7,354	126,917
Classifier 2	1,767	207,327	99.6%	50	12,955	124,918
Classifier 3	1,767	214,176	99.6%	75	19,804	11,285
Classifier 4	1,765	231,531	96.8%	97	37,159	11,285

Table 4.1. Summary of classifier performance. "accuracy" denotes the average percentage of correctly classified problems during ten-fold cross-validation. The oracle classifier always makes the right decision and the anti-oracle always the wrong decision. Minion always picks the standard solver and Minion-lazy always the lazy learning solver. The time to compute the features is the total time for all the 1,935 problems that could be analysed. The total time is the time taken to process (compute features, make decision and solve) all 1,773 problems which either solver can solve, including 5,000 seconds for each instance that a given classifier fails to solve. We exclude the 255 problems where both solvers time out, which would add 1,275,000 seconds to each classifier. An instance is misclassified if at least one solver solves it within the time limit and the solver that was not selected takes less time. The penalty is the number of seconds the specific classifier took longer than the oracle takes. All times are rounded to the nearest second.

Classifier 4 – final classifier

We adjust the parameters of the `J48` decision tree inducer to perform more pruning of the learned tree and eliminate the overfitting to the set of problems that we train on. Specifically, we reduced the confidence threshold for pruning from 25% to 1% and increased the minimum number of instances permissible at a leaf from 2 to 50. The decision tree generated with these parameters has only 13 nodes and is depicted in Figure 4.2 on the facing page. The classifier evaluated 96.8% of problems correctly and had a precision of 96.8% and a recall of 96.8%. As Table 4.1 on the previous page shows, the performance clearly suffers. However, as we discuss in detail in Section 4.5 on the facing page, the losses in practical performance are small compared to the win over using standard Minion, and with a classifier that is much less likely to be overfitted.

4.4.4. Evaluation on different data

The decision tree of classifier 4 is both easy to understand and the feature values it requires are cheap to compute. However, the question that remains is whether it is applicable for new problems. We have to consider two cases. The new problem could belong to a problem class that is contained in our data set and that classifier 4 was trained on. In this case, we are confident that our decision tree will give good results because of its high accuracy on the existing problem set. In the more interesting case, a new problem belongs to a problem class of which no representatives are contained in our set.

To evaluate the performance of the decision tree on unknown problem classes, we use the well-established technique of leave-one-out cross-validation and apply it to the set of problem classes. In n-fold cross-validation, the original data set is split into n parts of roughly equal size. Each of the n partitions is in turn used for testing. The remaining $n - 1$ partitions are used for training. In the end, every instance in the original data set will have been used for both training and testing in different runs (Kohavi, 1995). Leave-one-out cross-validation is n-fold cross-validation where n is the size of the data set – each part contains only a single datum.

Each part of the original data set that we use has one particular problem class removed. The idea is that if the `J48` decision tree inducer, using the same methodology as described above, learns the same decision tree as shown in Figure 4.2 on the next page for subsets of the problem classes, then it is likely to be problem class-independent. Unfortunately, not all of the problem classes contain problem instances where lazy learning is both faster and slower. Leaving one of these problem classes out may not have any effect on the learned decision tree, as they may not contribute to the knowledge how to distinguish between the two solvers. On the other hand, problems from an unknown problem class may have the same characteristic. For 24 out of 46 problem classes, the problems can be split into ones where lazy learning is faster and ones where it is slower.

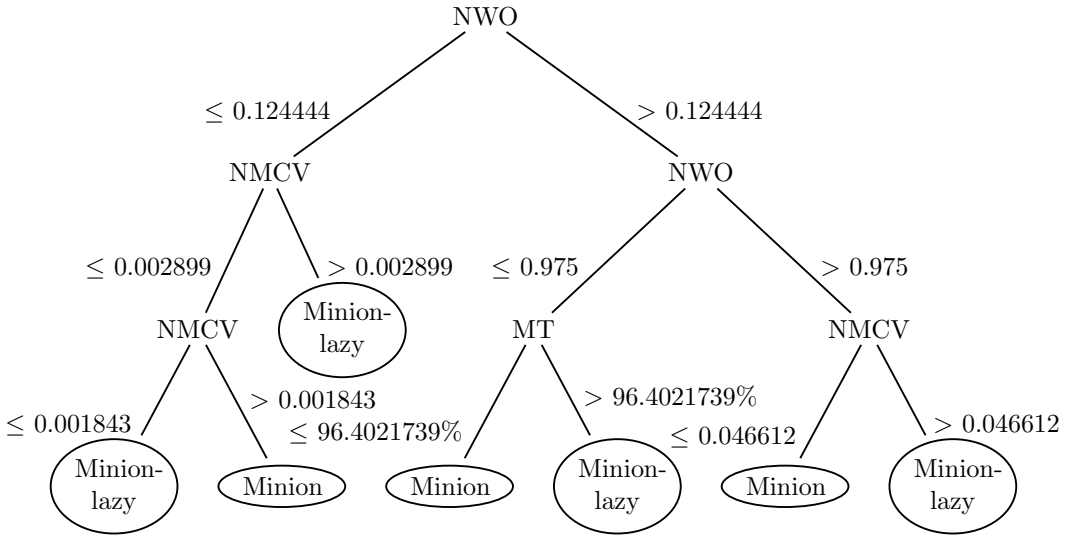


Figure 4.2. Final decision tree (classifier 4) to predict the faster solver for a given problem. NWO stands for normalised width of ordering, NMCV for normalised mean constraints per variable and MT for mean tightness.

Out of the 46 generated parts, exactly the same tree as shown in Figure 4.2 was learned for 24. 18 of those parts were created by removing a problem class that has no instances where Minion-lazy is faster. For all of the decision trees, the feature at the root node of the tree was the same as in Figure 4.2 and for all but one of them the value was the same as well. For 15 out of the 22 decision trees that were different, the only modification was the addition or deletion of a single subtree. Typical trees that were generated several times are shown in Figures 4.3 on the next page and 4.4 on page 59. They are still very similar to the decision tree in Figure 4.2; the differences are highlighted with dashed lined. For all but two of the generated trees, the proportion of correctly classified problems was higher than 95% and higher than 90% for all of them.

4.5. Classification performance

The performance of the final decision tree shown in Figure 4.2 improves significantly over simply running standard Minion on all of the problems, as shown in Table 4.1 on page 55. In particular, it achieves 29 more problems solved in 129,418 seconds less, compared to a maximum possible gain of 37 more problems in 166,577 seconds less for the perfect oracle classifier.

Calculating the features the classifier requires takes approximately 6 seconds per instance and we gain more than 70 seconds on average by using the classifier. The decision tree itself is very simple and the time required to make the decision which

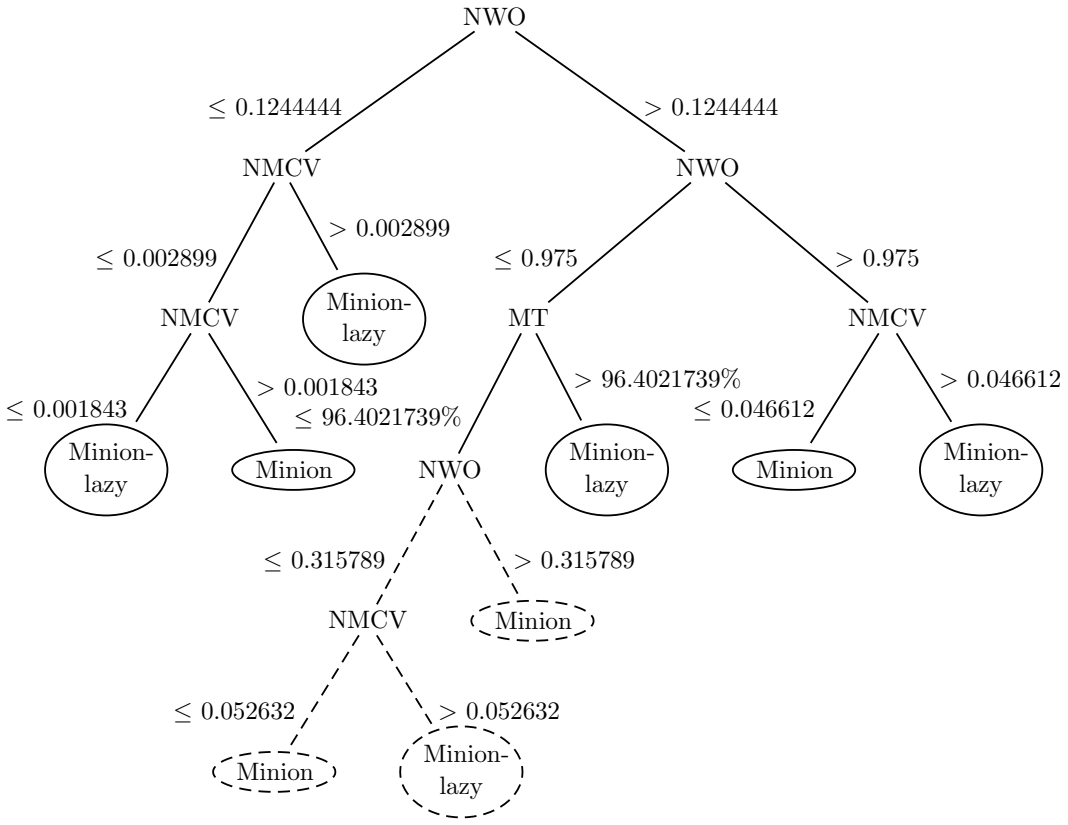


Figure 4.3. Typical decision tree with an additional subtree that was generated during cross-validation with data sets with one problem class left out. The added subtree is highlighted with dashed lines.

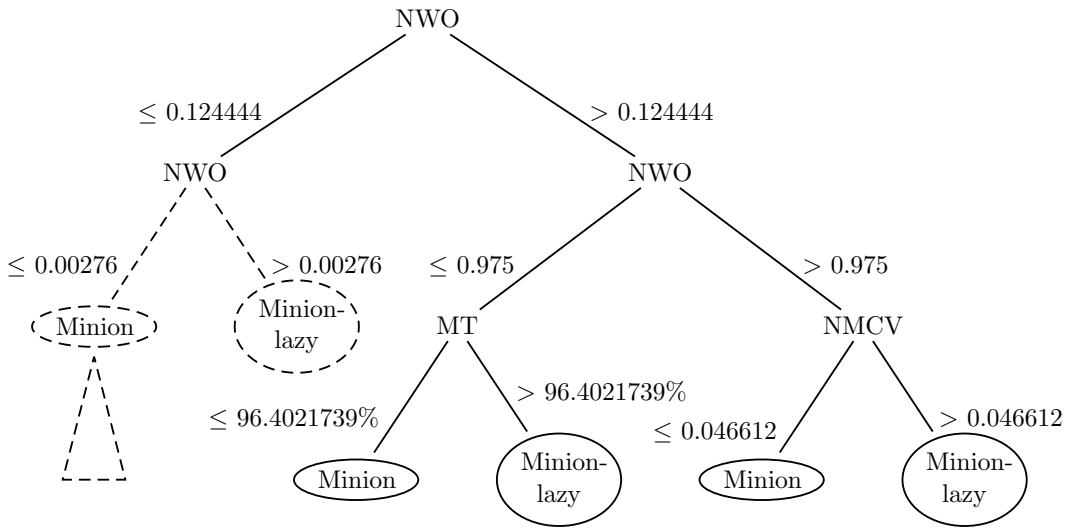


Figure 4.4. Typical decision tree with a subtree pruned that was generated during cross-validation with data sets with one problem class left out. The missing subtree is highlighted with dashed lines. Note that also the attribute that is switched on above the pruned subtree is different.

solver to use once all features values are known is negligible. We are therefore left with a net win over the whole problem set of approximately 39 hours for a total run time of approximately 422 hours. This is almost 85% of the best possible improvement – the oracle classifier gives a win of about 46 hours, assuming that it required no features and made its decision in zero time.

4.6. Understanding the problem domain

The features used in the decision tree of classifier 4 provide some insight into the performance characteristics of lazy learning. One of them is the mean tightness of the constraints. If it is above a certain value, Minion-lazy is faster. The mean tightness is a measure of the proportion of possible assignments to the constrained variables that the constraint disallows. Its use in the decision tree indicates that lazy learning is successful in identifying such disallowed assignments and pruning the search tree accordingly. The threshold value is quite high and indicates that the propagators for the existing constraints are unsurprisingly able to prune large parts of the search already and that the overhead of lazy learning is high compared to pure propagation.

Another feature used in the decision tree is the normalised mean constraints per variable. Minion-lazy is chosen when the value is either particularly low or particularly high. This indicates that for few constraints per variable, not enough propagation is achieved, while for many constraints per variable the interactions between

those constraints are not sufficiently captured by propagating individual constraints. In both cases, lazy learning is able to mitigate the problem.

It is unclear why the normalised width of ordering affects the performance of lazy learning. A possible explanation for its importance is that it serves as a proxy for a closely related feature that we did not consider but that has a strong effect on the performance of Minion-lazy.

4.7. Summary and contributions

This chapter showed how to use a decision tree learner to decide whether to use lazy learning in constraint solving. It demonstrated that we can successfully apply techniques for solving the Algorithm Selection Problem in this scenario. We cannot only improve the performance of constraint solving significantly, but also improve our understanding of the problem domain by inspecting the learned decision tree.

Through a series of decision trees, we showed that the number of features used in the tree can be reduced significantly while maintaining good classification performance. There are several benefits to such a reduction. First, fewer feature values need to be computed when classifying a problem and thus the overhead cost is reduced. Second, the learned decision tree is smaller and easier to understand.

We demonstrated the general applicability of the final decision tree through cross-validation across many different classes of constraint problems and are confident that it can be used to solve the Algorithm Selection Problem for this application domain in general. We hope that this will assist with the adoption of the powerful technique of lazy learning in constraint solving.

The main contributions of this chapter are as follows.

- The application of Machine Learning techniques to solve the Algorithm Selection Problem in a new domain.
- The use of decision trees to further the understanding of the factors that affect the performance of algorithms in the application domain.
- The use of cross-validation across problem classes to demonstrate the general applicability of a learned decision tree.

We selected the Machine Learning technique to use by hand and manually refined the learned decision tree by tweaking the used features and parameters. In general, it is desirable to do this automatically. The next chapter presents a case study for a different scenario and takes steps towards solving the Algorithm Selection Problem automatically.

5

Case study for the `alldifferent` constraint

This chapter looks at a case study for a different scenario. Instead of making a simple binary decision, there are now more options to choose from and some of these options depend on each other. Furthermore, we will try to automate the manual decisions that lead to the final decision tree in the previous chapter.

As mentioned before, selecting the Algorithm Selection technique to use is a problem for researchers without a strong background in the area because of the vast number of different approaches in the literature (cf. Chapter 3). This chapter looks at a number of different ways.

5.1. Introduction

The `alldifferent` constraint is a so-called *global* constraint that operates on at least two variables at the same time. It requires each variable in its scope to have a value that is different from the values of all other variables in its scope. The power of global constraints comes from the ability to consider larger parts of the problem together rather than small parts individually.

Consider for example a problem with five variables that are constrained by the same `alldifferent` constraint. If the potential values for each of the set of variables are the same four numbers, the `alldifferent` constraint can deduce that there is no satisfying assignment without any search – there are only four distinct values for five variables that have to be different. This requirement could also be enforced by a set of constraints that requires each pair of variables to be different. In this case however, additional search is necessary, as for each individual constraint there is an assignment of values to the two variables in its scope such that they are different.

The implementation and how to propagate the `alldifferent` constraint repre-

---

The material in this chapter has been published previously in: Lars Kotthoff, Ian Gent, and Ian Miguel. Using machine learning to make constraint solver implementation decisions. In *SICSA PhD conference*, 2010.
and: Ian Gent, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Machine learning for constraint solver design -– A case study for the `alldifferent` constraint. In *3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS)*, pages 13–25, 2010.
The contributions of the author of this dissertation are listed on page xix et seqq.

sents only one of many decisions to make when implementing a constraint solver. It is a much more fine-grained decision than the one in the case study in the previous chapter, where only one high-level binary decision needs to be made. In addition to having more choices, the decision can be made for every single instance of the constraint instead of globally for all instances.

In this chapter, we investigate choosing the most suitable implementation for all instances of the `alldifferent` constraint in a problem to solve. The potential performance improvement is smaller than that for making the decision per instance of the constraint, but the problem becomes much easier to solve. We use standard Machine Learning techniques to learn classifiers on a set of training problems.

We demonstrate that even in this case we can achieve significant performance improvements without tuning of the Machine Learning techniques. In particular, we make a series of multi-level decisions to choose the implementation of the `alldifferent` constraint for a particular problem.

5.2. Background

The `alldifferent` constraint requires all pairs of variables from the set of variables that it is imposed on to be different. For example `alldiff(`x_1, x_2, x_3`)` enforces $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$.

There are many different ways to implement the `alldifferent` constraint. First, there is a choice of whether to decompose the constraint or consider it as a whole. The decomposition enforces disequality on each pair of variables in the `alldifferent` constraint. More sophisticated versions (e.g. by Régin (1994)) consider the constraint as a whole and are able to do more propagation. Further variants are discussed by van Hoeve (2001). This decision is the first in the sequence.

The second decision to be made determines the actual implementation of the chosen version of the constraint. It depends on the previous decision of whether to decompose the constraint or not. While the implementation of disequality constraints required for the decomposition is trivial, there are many different ways to implement the holistic `alldifferent` constraint. For an in-depth survey of the different versions, see Gent et al. (2008).

We choose from nine different versions of the `alldifferent` constraint. They are,

- the version equivalent to the decomposition into binary not-equal constraints,
- the standard version implemented in Minion,
- the standard version without staged propagation,
- the standard version without incremental matching,
- the standard version with the Hopcroft-Karp algorithm (Hopcroft and Karp, 1973) to compute a matching instead of the Ford-Fulkerson algorithm (Ford and Fulkerson, 1956),

- the standard version without exploitation of strongly connected components,
- the standard version without a priority queue,
- the standard version with domain size checks and
- the standard version without assignment optimisation.

The features turned on and off in the individual versions are described in detail in Gent et al. (2008). The default choice reflected in the standard implementation achieves good performance on a wide range of problems.

The idea of using Machine Learning to make a series of decisions instead of a single one in Algorithm Selection has been used by Xu et al. (2007a) and Haim and Walsh (2009). They decide whether a SAT problem is satisfiable or not before, based on that decision, making the choice as to the most suitable algorithm. The crucial difference from the approach in this chapter is that they predict a feature of the problem that cannot be computed easily while we are predicting a partial solution to the Algorithm Selection Problem.

Our approach represents an important step towards making a series of algorithm selections that depend on each other. The problem in this form has not been considered before in the literature. While approaches that evolve a basic algorithm (e.g. Minton (1996)) implicitly take dependencies into account, this information is not used in the performance models. Research more relevant to this has been done in the automatic tuning community, where parameters often depend on each other and their values cannot be selected without taking the interactions into account or some parameters have to be set before the possible values for others are known (e.g. Hutter et al. (2009b)). The crucial difference however is that we make such decisions in the context of Algorithm Selection systems instead of tuning. Earlier decisions not only restrict the space of possible later decisions, but also affect the way we are choosing from among those.

5.3. Evaluation problems

We evaluated the performance of the different versions of the `alldifferent` constraint on two different sets of problems. The first one was used for learning classifiers, the second one only for the evaluation of the learned classifiers.

The training set consisted of 277 problems from 14 different problem classes. The set to evaluate the learned classifiers consisted of 1036 problems from 2 different problem classes that were not present in the set we used for Machine Learning. We chose this set for evaluation because the low number of different problem classes makes it unsuitable for training. A complete list of problem classes can be found in Appendix E on page 149.

The reference constraint solver used is Minion version 0.9 and its default implementation of the `alldifferent` constraint `gacalldiff`. The experiments were run

with binaries compiled with g++ version 4.4.3 and Boost version 1.40.0 on machines with 8 core Intel E5430 2.66GHz, 8GB RAM running CentOS with Linux kernel 2.6.18-164.6.1.el5 64Bit.

We imposed a time limit of 3600 seconds for each problem. The total number of problems that no solver could solve within that limit was 66 for the first set and 26 for the second set. We took the median CPU time of 3 runs for each problem to mitigate noise inherent in empirical data.

As Figure 5.1 on the next page shows, adapting the implementation decision to the problem instead of always choosing a standard implementation has the potential to achieve significant speedups on some problems of the first set of problems and still noticeable speedups on the second set.

We ran the problems with 9 different versions of the `alldifferent` constraint – the naïve version which is operationally equivalent to the binary decomposition and the 8 different implementations of the more sophisticated version described in Section 5.2 on page 62. The amount of search done by the 8 versions which implement the more sophisticated algorithm is the same. The variables and values were searched in the order they were specified in in the model of the problems.

5.4. Problem features and their measurement

We measured 37 features of the problems. They describe a wide range of features such as constraint and variable statistics and a number of features based on the primal graph. The features edge density, clustering coefficient, normalised standard deviation of degree, multiple shared variables, normalised mean constraints per variable, ratio of auxiliary variables and proportion of symmetric variables were used in exactly the same way as in the previous chapter. Features used in a different way and additional features are described below. Detailed descriptions of the features used before can be found in Section 4.3 on page 50.

Normalised degree The minimum, maximum, mean and median normalised degree were used.

Width of ordering The width of the ordering normalised by the number of vertices was used.

Width of graph The width of the graph normalised by the number of vertices was used.

Variable domains The quartiles and the mean value over the domains of all variables.

Constraint arity The quartiles and the mean of the arity of all constraints (the number of variables constrained by it), normalised by the number of constraints.

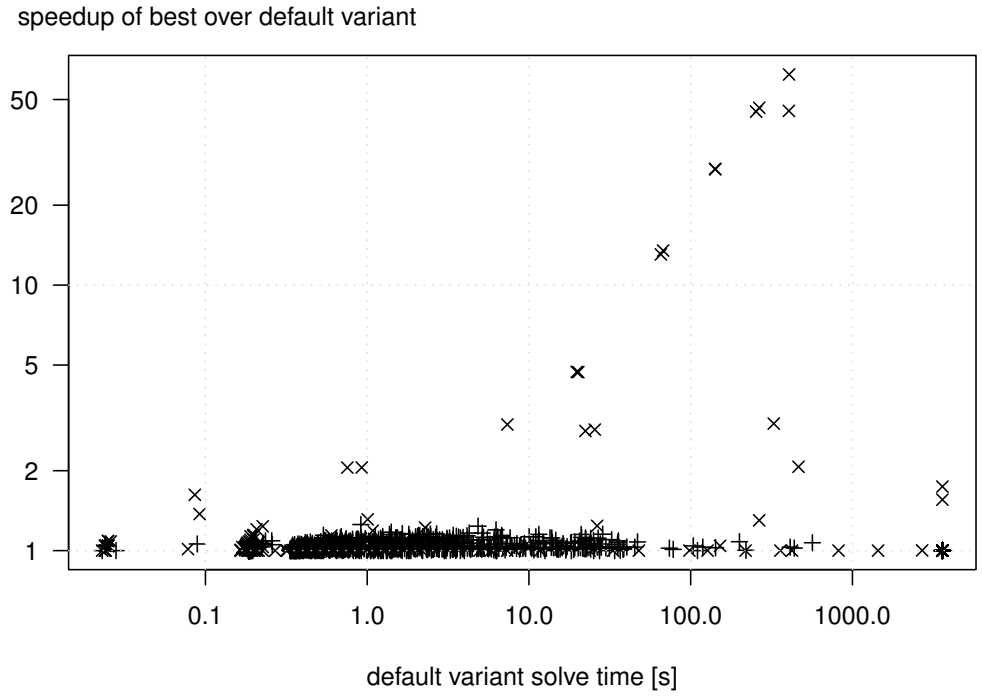


Figure 5.1. Potential speedup a problem-specific implementation could achieve over always using the default implementation. The crosses represent the problems of the first data set, the pluses the problems of the second data set. A speedup of one means that the default version of `alldifferent` is the fastest version, a speedup of two means that the fastest version of `alldifferent` is twice as fast as the default version. Overall, the performance improvement we can achieve on the first set is much bigger than what we can achieve on the second set.

Tightness The tightness quartiles and the mean tightness over all constraints were used.

Alldifferent statistics The size of the union of all variable domains in an `alldifferent` constraint divided by the number of variables. This is a measure of the ratio of possible to satisfying assignments. We used the quartiles and the mean over all `alldifferent` constraints.

Computing the features took 27 seconds per problem on average.

5.5. Learning a problem classifier

Before using Machine Learning on the set of training problems, we annotated each problem with the `alldifferent` implementation that had the best performance on it according to the following criteria. If the naïve `alldifferent` implementation took less CPU time than all the other ones, it was chosen, else the implementation that had the best performance in terms of search nodes per second was chosen. All implementations except the naïve one explore the same search space. If no solver was able to solve the problem, we assigned a "don't know" annotation.

We used the WEKA (Hall et al., 2009) Machine Learning software through the R (Team, 2011) interface to learn classifiers. We used a wide range of the WEKA classifiers that were applicable to our problem – algorithms which generate decision rules, decision trees, Bayesian classifiers, nearest neighbour and neural networks. Our selection is broad and includes most major Machine Learning methodologies. We used a Bayesian classifier (`BayesNet`), a nearest neighbour classifier (`IBk`), a neural network classifier (`MultilayerPerceptron`), rule-based classifiers (`ConjunctiveRule`, `DecisionTable`, `JRip`, `OneR` and `PART`), decision tree learners (`FT`, `LADTree`, `J48`, `J48graft`, `BFTree`, `NBTree`, `RandomForest`, `RandomTree` and `REPTree`) and the `HyperPipes` classifier. All of the implementations are described in Witten et al. (2011). For all of these algorithms, we used the default parameters provided by WEKA.

The performance of the learned classifiers was measured in terms of misclassification penalty. The misclassification penalty is the additional CPU time we require to solve a problem when choosing to solve it with a solver that is not the fastest one. If the selected solver was not able to solve the problem, we assumed the timeout of 3600 seconds minus the CPU time the fastest solver took to be the misclassification penalty. This only gives a lower bound for the true misclassification penalty.

5.5.1. Cost model

We again decided to assign the maximum misclassification penalty, or the maximum possible gain (cf. Figure 5.1 on the preceding page), as a cost to each problem to bias the WEKA learners towards the problems we care about most. We used the

classifier learner	misclassification penalty [s] all equal	cost model
default decision	2304	2304
BayesNet	1493	106
BFTree	249	1
ConjunctiveRule	1433	1580
DecisionTable	396	1.4
FT	250	8
HyperPipes	863	863
IBk	1	1
J48	244	1.3
J48graft	244	1.3
JRip	249	1.2
LADTree	13	1.3
MultilayerPerceptron	1045	8.5
NBTree	396	1.2
OneR	61	4.3
PART	4	1.1
RandomForest	1.2	1
RandomTree	1	1
REPTree	402	1.4

Table 5.1. Misclassification penalty for all classifiers with and without cost model on the first data set. All numbers are rounded.

common technique of duplicating instances (Witten et al., 2011), similar to the methodology used in the previous chapter. Each problem appeared in the new data set $1 + \lceil \log_2(\texttt{cost}) \rceil$ times. The particular formula to determine how often each problem occurs was chosen empirically such that problems with a low cost are not disregarded completely, but problems with a high cost are much more important. Each problem will be in the data set used for training the Machine Learning classifiers at least once and at most 13 times for a theoretic maximum cost of 3600.

First, we make the decision whether to use the `alldifferent` version equivalent to the binary decomposition or the holistic one. Then, based on the previous decision, we decide which specific version of the `alldifferent` constraint to use. Each individual classifier below is a combination of classifiers that make this series of decisions.

Table 5.1 shows the total misclassification penalty for all classifiers with and without instance duplication on the first data set. It clearly shows that our cost model improves the performance significantly in terms of misclassification penalty for almost all classifiers.

The performance of each classifier was evaluated using stratified three-fold cross-validation. The original data set is split into three parts of roughly equal size. Each of the three partitions is in turn used for testing. The remaining two partitions are used for training. In the end, every problem will have been used for both training and testing in different runs (Kohavi, 1995). Stratified cross-validation ensures that the ratio of the different classification categories in each subset is roughly equal to the ratio in the whole set. If, for example, about 50% of all problem in the whole data are solved fastest with the naïve implementation, it will be about 50% of the problems in each subset as well.

5.5.2. Evolving the feature set

The time required to compute the features was 27 seconds per problem instance on average, and it took 0.2 seconds per problem instance on average to run the classifiers and combine their decisions. This overhead would make our system slower than always using the default implementation. This is mostly because of the cost of computing all the features required to make the decision. Most of the time required to make the decision is spent computing the features that the classifiers need. We removed the most expensive features – all the properties of the primal graph described in Section 5.4 on page 64 apart from edge density.

The problem with using cross-validation to estimate the performance of a classifier is that instead of a single classifier that can easily be applied to another data set, there are several classifiers. One possible solution is to discard the classifiers used to estimate the performance and train a new one on the entire data set. However, this carries the risk of overfitting to the training data set. Instead, we decided to keep all classifiers created during cross-validation and use them as an ensemble. We combine the decisions of the individual classifiers by majority vote. The technique of combining the decisions of several classifiers was introduced by Freund and Schapire (1995) and formalised by Dietterich (2000). Each individual classifier is again a combination that makes the series of decisions required, but now also for each individual decision an ensemble of the three classifiers trained during cross-validation.

The results for both feature and data sets are shown in Table 5.2 on the next page. The performance with only the features that are cheap to compute is not significantly worse and sometimes even better. The time required to compute all those features is only about 3 seconds per problem. On the first set of problems, we solve each problem on average 8 seconds faster using our system (misclassification penalty of default decision minus that of our system divided by the number of problems in the set). We are therefore left with a performance improvement of an average of 5 seconds per problem. On the second set, we cannot reasonably expect a performance improvement – the perfect oracle classifier only achieves about 0.2 seconds per problem on average.

The results also show that the classifiers we have learned on a data set that contains problems from many problem classes can be applied to a different data set with

	misclassification penalty [s]			
	problem set 1		problem set 2	
classifier	all features	cheap features	all features	cheap features
anti-oracle	19993	19993	47144	47144
default decision	2304	2304	223	223
BayesNet	106	62	284	186
BFTree	1	1	220	220
ConjunctiveRule	1580	1580	218	218
DecisionTable	1.4	1.5	223	221
FT	8	8.5	575	220
HyperPipes	863	863	468	468
IBk	1	1	131	505
J48	1.3	1.2	604	569
J48graft	1.3	1.2	603	603
JRip	1.2	1	607	607
LADTree	1.3	1.4	621	610
MultilayerPerceptron	8.5	8.5	236	220
NBTree	1.2	1.2	516	228
OneR	4.3	4.3	219	219
PART	1.1	1.2	602	605
RandomForest	1	1	223	221
RandomTree	1	1	605	506
REPTree	1.4	1.4	234	234

Table 5.2. Summary of classifier performance on both sets of problems in terms of total misclassification penalty in seconds. We first evaluated the performance using the full set of features described in Section 5.4 on page 64, then using only the cheap features. The anti-oracle always makes the worst possible decision. The "default decision" classifier always makes the same decision. Three-fold cross-validation was used. All numbers are rounded.

	data set 1		data set 2	
	all features	cheap features	all features	cheap features
best	IBk	BFTree	IBk	BayesNet
worst	ConjunctiveRule	ConjunctiveRule	LADTree	LADTree

Table 5.3. Individual best and worst classifiers for the different data and feature sets for the numbers presented in Table 5.2 on the previous page.

problems from different classes and still achieve a performance improvement in terms of selecting the best algorithm. Unfortunately, the overhead of doing Algorithm Selection on the second set of problems devours the improvement. Note also that the classifier that performs best on one data set is not necessarily the best performer on the other data set. The same observation can be made for the classifier with the worst performance on one data set. This means that we cannot simply choose "the best" classifier or discard "the worst" for a given set of training problems. Table 5.3 summarises this result. The individual best and worst classifiers vary not only with the data set, but also with the set of features used.

5.6. Summary and contributions

This chapter presented the selection of the implementation of the `alldifferent` constraint as a case study for using Machine Learning to solve the Algorithm Selection Problem. In addition to using techniques described in the literature, we made a series of decisions that depend on each other to arrive at the most suitable implementation for solving a given problem. We presented empirical data that showed the effectiveness of a cost model that assigns a weight to each problem and that using a reduced feature set does not affect performance negatively in general.

During the evaluation of the performance of the learned classifiers, we noticed that the performance varies by several orders of magnitude. It is furthermore not clear which technique is the best – the ones with good performance in one scenario exhibit bad performance in others and vice versa. While the cost model is a way of improving the performance in general, it does not affect the performance of some of the classification learners and is even detrimental in one case.

Our system achieved performance improvements even taking into account the time it takes to compute the features and run the learned classifiers. For atypical sets of problems, where always making the default decision is the right choice in almost all of the cases, we were not able to compensate for this overhead, but we are confident that we can achieve a real speedup on average in general.

We showed conclusively that a series of decisions that depend on each other can be made in practice and improve the performance over always making a default

decision. This is an important contribution because the total prediction error is the product of the prediction errors of all the individual decisions. Even small errors at each step can lead to a large overall error that would make the system useless in practice.

The main contributions of this chapter are as follows.

- Providing quantitative evidence for the effectiveness of a cost model to bias the classification learners towards important instances as used in the previous chapter.
- The demonstration of the feasibility of making multi-level decisions that depend on each other with accuracy sufficient to achieve performance improvements.
- The demonstration of the variability of classification learner performance across different data and feature sets.

This chapter established that selecting a good classification learner is crucial to the effectiveness of an Algorithm Selection system. We are now faced with a dilemma – in order to solve the Algorithm Selection Problem effectively, we have to solve another Algorithm Selection Problem, namely the selection of the best classification learner. We could of course use one of the many techniques described in the literature to do that, but then the question arises of whether this chosen technique is the best one.

Selecting the best way of solving the Algorithm Selection Problem is clearly as hard as solving the Algorithm Selection Problem itself. The next chapter focuses on ensemble classification, a technique that provides an elegant solution to this dilemma. Ensemble classification was introduced earlier in this chapter as a means of combining the classifiers learned during different stages of a cross-validation. It enables us to perform Algorithm Selection effectively, even without having to decide directly on a best technique.

6

Ensemble classification for Algorithm Selection

The results of the previous chapter demonstrate that there is a high level of uncertainty attached to the performance of an individual technique for solving the Algorithm Selection Problem. Good performance in one scenario does not imply good performance in another scenario and similarly for bad performance.

Choosing the best technique can be crucial for the feasibility of the system, but it is not straightforward to do this. In fact, this is yet another instance of the Algorithm Selection Problem. This chapter applies a technique borrowed from Machine Learning, ensemble classification, to this problem.

6.1. Introduction

A review of the Algorithm Selection literature (cf. Chapter 3) shows that there is no general consensus as to the best Machine Learning techniques. The choice of a particular Machine Learning technique is usually justified by the performance of the system. Only in a few cases do the authors compare different methods and choose the one with the best performance. But has the technique that will provide the best performance in general been chosen?

This chapter investigates a means by which we can avoid having to choose the best technique. Instead of choosing a single best Machine Learning algorithm, we use an ensemble of several Machine Learning algorithms. We show that the performance of the ensemble is as good as and sometimes better than the performance of the best individual technique in a given scenario while at the same time being more predictable and stable.

We used ensemble classification in the previous chapter already as a means of combining the different classifiers that are learned during the stages of a cross-validation. This chapter promotes the technique from merely addressing a technical issue to making Algorithm Selection more accessible and providing a more robust

The material in this chapter has been published previously in: Lars Kotthoff, Ian Miguel, and Peter Nightingale. Ensemble classification for constraint solver configuration. In *16th International Conference on Principles and Practices of Constraint Programming*, pages 321–329, September 2010.

The contributions of the author of this dissertation are listed on page xix et seqq.

performance.

6.2. Background

In Machine Learning, combining several classifiers is a well-established technique. It was first introduced by Freund and Schapire (1995) and formalised by Dietterich (2000). An ensemble is a set of classifiers that have all been trained to make the same kind of decision. The ensemble makes a decision for a given problem by getting decisions from all of its classifiers or a subset and combining those decisions.

Consider for example an ensemble of three classifiers. The first classifier was learned using a decision tree inducer, the second is a naïve Bayes classifier and the third a neural net. All three were trained using the same data, but the models they learn from that data are quite different. Apart from the different representations, the learned concept could be different as well. The strengths and weaknesses of these classifiers potentially lie in different areas of the training data, such that for example the decision tree might classify an example correctly that the neural net does not.

Given new data, all three classifiers are asked for their decision. The returned classifications are not necessarily going to be the same. An intuitive way of selecting one of those classifications is to choose the one that occurs most often. If two or more classifications occur with the same frequency, a way of breaking this tie is needed.

There are many different methods for creating different classifiers and combining their predictions. The ensemble can consist of classifiers that have been trained independently or it can be engineered to contain classifiers that complement each other – in situations where one classifier is known to have bad performance, other classifiers are specifically trained to compensate. One way to achieve this is to use different subsets of the training data for each classifier in the ensemble. Another technique is to train the same classification learner on different feature subsets. More details can be found in e.g. Dietterich (2000), Witten et al. (2011).

An obvious advantage of ensemble classification is that there is no need to choose a single best classifier. Apart from that, the ensemble is more likely to be robust with respect to different input data. Another observation however is that in some cases the performance of the ensemble is better than the performance of its best constituent classifier (Dietterich, 2000).

The problem of selecting the most suitable Machine Learning technique for a given problem is an area of active research in the Machine Learning community (cf. Chapter 3). In addition to selecting the best algorithm, it has also been recognised that the difference between different parameter settings for the same algorithms can have a more profound effect than choosing a different algorithm (Lavesson and Davidsson, 2006). Chen and Jin (2007) propose a general framework for solving this problem in the context of Machine Learning. Ensemble classification is an effective means of mitigating this problem.

6.3. Evaluation data sets and features

The data sets are taken from the previous two chapters. For one, we select between two different versions of the Minion constraint solver – with and without lazy learning. For the other data set, we select one of nine different implementations of the `alldifferent` constraint, also in the context of the Minion constraint solver.

The problem features we used are described in Section 5.4 in the previous chapter. For the first data set, we did not use the statistics for the `alldifferent` constraint as they are not applicable. For the second data set, we did not use the number of literals pruned by singleton arc consistency preprocessing (cf. Section 4.3 on page 50) for the same reason.

6.4. Learning classifiers and ensemble

The methodology we used in this chapter is the same as described in the previous two chapters. In contrast to the previous chapter however, we do not make a series of decisions for the implementation of the `alldifferent` constraint, but flatten the tree of decisions – the set of leaves constitutes the set of possible decisions. This simplifies the evaluation and makes the results comparable with the ones achieved on the other data set.

We used the WEKA (Hall et al., 2009) Machine Learning software through the R (Team, 2011) interface to learn classifiers. The specific classifiers we used are the same as in the previous chapter; `BayesNet`, `BFTree`, `ConjunctiveRule`, `DecisionTable`, `FT`, `HyperPipes`, `IBk`, `J48`, `J48graft`, `JRip`, `LADTree`, `MultilayerPerceptron`, `NBTree`, `OneR`, `PART`, `RandomForest`, `RandomTree` and `REPTree`.

Again we used the misclassification penalty as a performance measure of the learned classifiers and attached the maximum classification penalty as a weight to the individual problems.

We aggregate the decisions of the individual classifiers in the ensemble by majority vote; we take the predictions of each classifier for an individual problem and choose the one that occurs most often. Ties are broken by selecting a label according to the alphanumeric ordering. A thorough investigation by Bauer and Kohavi (1999) showed that majority voting performs better in general than other techniques for combining decisions in classifier ensembles.

For both data sets, we generated separate partitions for training and test data as follows. First, we removed the instances of a randomly selected problem class. Then we removed about 33% of the remaining instances at random. The remaining data was used for training and the removed instances for testing. We generated 10 different partitions of approximately equal size for each data set in this manner.

We ran each Machine Learning algorithm on each training partition and evaluated its performance in terms of misclassification penalty through stratified 10-fold cross-validation (Kohavi, 1995). The median of the 10 folds is our performance estimate

for unseen data. The performance on the corresponding test set is evaluated by using all of the 10 classifiers trained during cross-validation as an ensemble. The absolute misclassification penalties are normalised and scaled to be comparable across the different data sets.

The most important issue we are addressing is the generality of the learned classifier – given its performance on the data set we are using for testing, will it perform equally well on unknown data? There are two different cases. The unknown data could contain new instances from a problem class that the classifier has seen before or the data could consist of unknown instances from unknown problem classes. We address both scenarios by removing individual problem classes and random problems from the original data set. Using this method, we test for overfitted classifiers at the same time.

6.5. Results

Figure 6.1 on the facing page shows the performance on the different partitions. It becomes clear that the performance on one set of data, even when using cross-validation, is not a good predictor of the performance on another set of data in this scenario. This is demonstrated by the length of the arrows. In only one of twenty cases, the classifier that has the best performance on the training set also exhibits the best performance on the test set. In two cases, the best classifier becomes the worst on new data. In absolute terms, the difference between the best and the worst individual classifiers is up to several orders of magnitude.

It also becomes clear that this effect is attenuated by using the ensemble classifier – in almost all cases, the performance differences on different sets of data are less pronounced, as denoted by the lengths of the arrows starting at the crosses (ensemble classifier) versus the ones starting at the circles (single classifier). The ensemble classifier is more robust in that its performance does not vary as much. The algorithm with the best average performance over all the data was `BFTree`. The ensemble classifier was within 1% of its performance.

On eight data sets, the performance of the ensemble classifier is better on the test data than on the training data relative to the other classifiers. While the single best classifier is always better than the ensemble on the training data, there are seven cases where the ensemble performs better than the single best classifier on the test data. This observation provides further evidence for the robustness of the ensemble approach.

The ensemble classifier does not have to contain a large number of classifiers to achieve the performance shown in Figure 6.1. Figure 6.2 on page 78 shows the results for just three classifiers – `BayesNet`, `MultilayerPerceptron` (a neural network algorithm) and `J48` (an implementation of the well-known C4.5 algorithm (Quinlan, 1993)). The improvements over using a single classifier are comparable to the ones shown in Figure 6.1. In this case, the performance of the ensemble is sometimes better than that of the single best classifier on both training and test data.

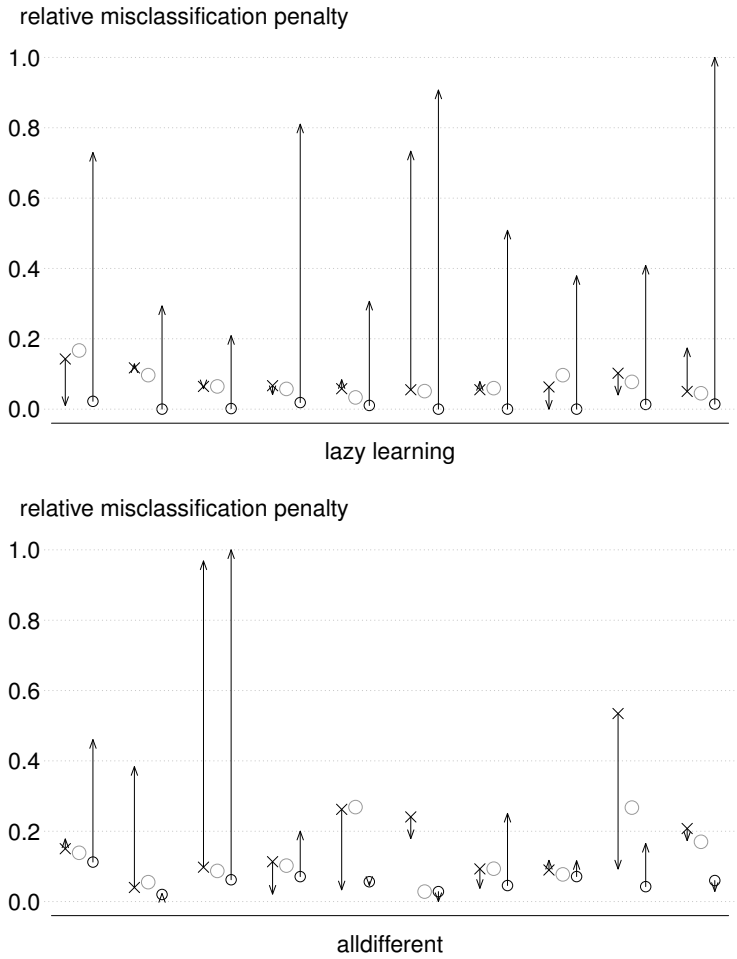


Figure 6.1. Classifier performance on the different partitions. The misclassification penalties were normalised by that of the best classifier across all partitions and scaled between best and worst classifier to make the different data sets comparable (i.e. the best classifier is now 0 and the worst 1). The black circles show the performance of the classifier that performed best on the training set. The larger, grey circles show the performance of the classifier that was the best one on the test partition. The cross denotes the performance of the ensemble classifier on the training set. The ends of the arrows denote the performance on the test partition for best individual and ensemble classifiers. The length of the arrow is a measure of the uncertainty of the prediction of the performance of the classifier in general from its cross-validation performance during the training phase.

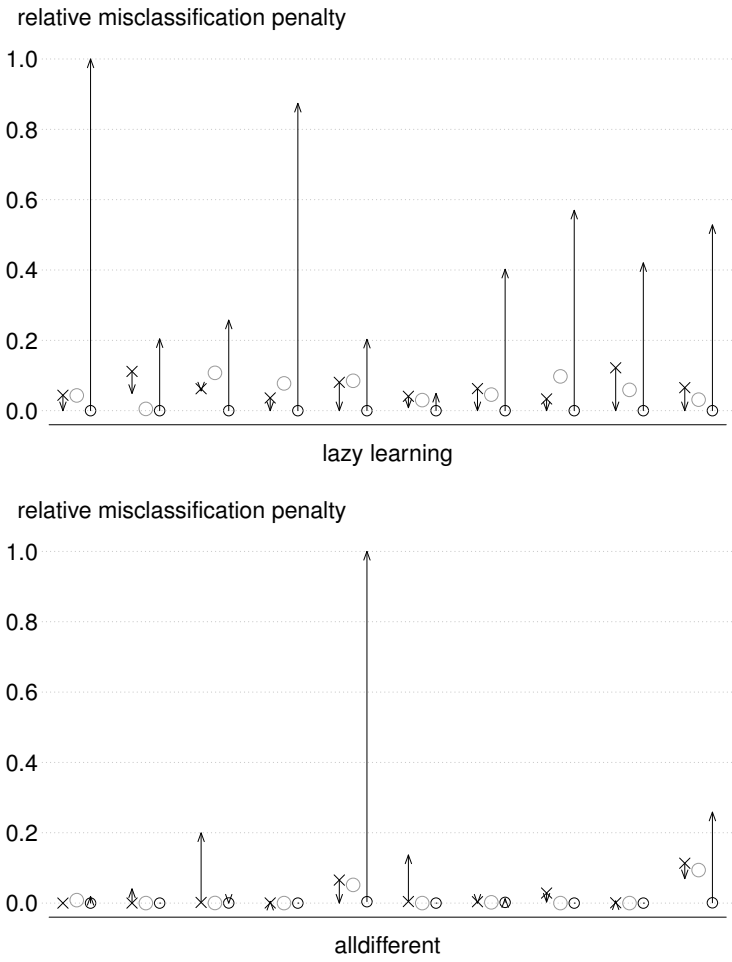


Figure 6.2. Classifier performance for three different classifiers. Note that in five cases the performance of the ensemble classifier was better than that of an individual best one on both training and test data. That is, both the cross and the tip of the arrow originating at the cross are lower than the black circle and the larger grey circle, respectively.

The performance of the ensemble of course depends on the performance of its constituent classifiers. The approach is more robust with respect to classifiers with bad performance though – if one of the classifiers in the ensemble performs badly, the other ones can compensate for this. The three Machine Learning algorithms we chose for the smaller ensemble were selected because they represent different Machine Learning methodologies and not because of their performance on our Algorithm Selection data. Most notably, we did not include the classifier with the best overall performance, `BFTree`.

In terms of solve time, the ensemble classification approach improves over always making a default decision. The improvement is substantial for lazy learning and marginal for alldifferent, similar to the results reported in the previous two chapters.

6.6. Summary and contributions

This chapter presented an investigation into the variability of the performance of different Machine Learning techniques on two Algorithm Selection Problems. Applying ensemble classification, a first in the context of Algorithm Selection, is an efficient means of reducing this variability and ensuring good performance across a range of different scenarios.

We based our investigation on experimental results for a large number of diverse problems and a large number of different Machine Learning techniques. Our results demonstrated that the performance of a Machine Learning algorithm varies a lot across different data sets. Predictions of its performance on new data cannot be made without investing significant effort into substantiating these claims. In particular, a classifier that exhibits good performance may perform badly on other data and a classifier with bad performance that would appear unsuitable may exhibit much better performance on new data.

We showed conclusively that ensemble classification, a technique borrowed from Machine Learning, is an effective way of mitigating this problem. An ensemble of classifiers whose decisions are combined by majority vote exhibits not only much more robust and predictable performance, but also performance that is equal to that of well-performing individual classifiers. We furthermore showed that the performance of the ensemble can even exceed the performance of the best individual classifier. While these results have been known in the Machine Learning community for supervised learning problems, we have shown that they also apply in the context of the Algorithm Selection Problem. We used explicit ensembles of classifiers for Algorithm Selection for the first time.

While combining several classifiers adds significant overhead in the offline phase when more classifiers need to be learned, the overhead in the online phase when new problems are classified was negligible in our experiments. The time required to compute the problem features was much higher than the time to run the additional classifiers and combine their predictions in all cases. In particular, running an individual classifier on a single problem only takes a few milliseconds on average.

We provided convincing evidence that even with an ensemble consisting of only a small number of classifiers, the advantages of ensembles still apply. The performance of the ensemble improved over the performance of an individual classifier even more often in this case. A smaller ensemble can reduce both the offline training time as well as the time spent classifying a given problem significantly.

One major advantage of ensemble classification is that individual Machine Learning algorithms can be combined without intrinsic knowledge of each individual one. The level of Machine Learning expertise required is reduced significantly without affecting the results negatively.

The main contributions of this chapter are as follows.

- The application of Machine Learning ensembles to the Algorithm Selection Problem.
- The demonstration of the improvements in robustness and performance the ensemble achieves over a single classifier.
- The demonstration of the effectiveness of even an ensemble with a small number of classifiers.
- The reduction of the Machine Learning knowledge required to tackle Algorithm Selection through the use of ensembles.

Ensemble classification is an efficient means of avoiding having to select the best technique for solving the Algorithm Selection Problem. This does not mean however that solving Algorithm Selection problems is now a straightforward matter – if all the classifiers in the ensemble exhibit bad performance, so will the ensemble. While this chapter presented a way of alleviating the difficulty of selecting the individual best technique, there is still a need to identify techniques with at least good performance.

This can be done with relatively little experience in Algorithm Selection, but researchers with no background in the area will still struggle to achieve good performance without significant trial and error. The next chapter addresses this final problem by identifying Machine Learning techniques that exhibit good performance across a representative sample of Algorithm Selection problems.

7

What Machine Learning technique to use?

The previous chapter presented the technique of Machine Learning ensembles, which can be used to avoid the problem of having to decide on a particular Machine Learning technique to use for Algorithm Selection. While this is an effective way of achieving good performance without having to worry too much about the Machine Learning algorithms, it does not address the fundamental problem – which Machine Learning algorithms and techniques should be used for Algorithm Selection?

7.1. Introduction

Approaches in the literature usually justify their choice of a Machine Learning methodology (or a combination of several) with the performance improvement they achieve over a system that does not perform Algorithm Selection. The majority of researchers do not compare with other Machine Learning techniques. A large percentage also does not critically assess the real performance – could we do as well or even better by using just a single algorithm instead of having to deal with portfolios and complex Machine Learning?

Researchers wishing to use Algorithm Selection techniques and Machine Learning are faced with several dilemmas. First, the sheer number of different Machine Learning approaches in the literature (cf. Chapter 3) makes it very hard to decide which one to choose. Second, presented approaches may not be evaluated critically when they are proposed, thus creating a misleading impression of their real performance. In particular, even though a system achieves significant performance improvements, there may be another Machine Learning technique that achieves even greater improvements, but that has not been considered for comparison.

This chapter presents a comprehensive comparison of Machine Learning paradigms

---

Most of the material in this chapter has been published previously in: Lars Kotthoff, Ian P. Gent, and Ian Miguel. A Preliminary Evaluation of Machine Learning in Algorithm Selection for Search Problems. In *Fourth Annual Symposium on Combinatorial Search*, pages 84–91, July 2011.
and: Lars Kotthoff, Ian P. Gent, and Ian Miguel. An Evaluation of Machine Learning in Algorithm Selection for Search Problems. *Submitted to AI Communications*.
The first paper won the Best Student Paper Award.
The contributions of the author of this dissertation are listed on page xix et seqq.

and techniques for tackling the Algorithm Selection Problem. It evaluates the performance of a large number of different techniques on data sets used in systems from the literature. It also compares the results with existing systems and to a simple "winner-takes-all" approach where the best overall algorithm is always selected – an approach that performs surprisingly well in practice. The chapter closes by giving recommendations as to which Machine Learning techniques should be considered when performing Algorithm Selection.

7.2. Algorithm Selection methodologies

In an ideal world, we would know enough about the algorithms in a portfolio to formulate rules to select a particular one based on certain characteristics of the problem to solve. In practice, this is not possible except in trivial cases. For complex algorithms and systems, we do not understand the factors that affect the performance of a specific algorithm on a specific problem well enough to make the decisions the Algorithm Selection Problem requires with confidence.

As outlined before, a common approach to overcoming these difficulties is to use Machine Learning. Several Machine Learning methodologies are applicable here. We present the most prevalent ones below. In addition to these and like in the evaluations in previous chapters, we use a simple majority predictor that always predicts the algorithm from the portfolio that has the best performance most often on the set of training instances ("winner-takes-all" approach) for comparison purposes. This provides an evaluation of the real performance improvement over manually picking the best algorithm from the portfolio. We use the WEKA (Hall et al., 2009) `ZeroR` classifier implementation for this purpose.

7.2.1. Case-based reasoning

Case-based reasoning informs decisions for unseen problems with knowledge about past problems. An introduction to the field can be found in Riesbeck and Schank (1989). The idea behind case-based reasoning is that instead of trying to construct a theory of what characteristics affect the performance, examples of past performance are used to infer performance on new problems.

The main part of a case-based reasoning system is the case base. We use the WEKA `IBk` nearest-neighbour classifier with 1, 3, 5 and 10 nearest neighbours as our case-based reasoning algorithms. The case base consists of the problem instances we have encountered in the past and the best algorithm from the portfolio for each of them – the set of training instances and labels. Each case is a point in n-dimensional space, where n is the number of attributes each problem has. The nearest neighbours are determined by calculating the Euclidean distance. While this is a very weak form of case-based reasoning, we simply do not have more information about the problems and algorithms from the portfolio that we could encode in the reasoner.

The attraction of case-based reasoning, apart from its conceptual simplicity, is the fact that the true relationship between attributes and performance can be arbitrarily complex. As long as the training data is representative (i.e. the case base contains problems similar to the ones we want to tackle with it), the approach will achieve good performance – all it needs to do is match new data with old data.

We use the AQME system (Pulina and Tacchella, 2009) as a reference system that uses case-based reasoning to compare with.

7.2.2. Classification

Intuitively, Algorithm Selection is a simple classification problem – label each problem instance with the algorithm from the portfolio that should be used to solve it. We can solve this classification problem by learning a classifier that discriminates between the algorithms in the portfolio based on the characteristics of the problem.

We use the WEKA classifiers

- `AdaBoostM1`,
- `BayesNet`,
- `BFTree`,
- `ConjunctiveRule`,
- `DecisionTable`,
- `FT`,
- `HyperPipes`,
- `J48`,
- `J48graft`,
- `JRip`,
- `LADTree`,
- `LibSVM` (with radial basis and sigmoid function kernels),
- `MultilayerPerceptron`,
- `OneR`,
- `PART`,
- `RandomForest`,
- `RandomTree` and

- `REPTree`.

Our selection is large and inclusive and contains classifiers that learn all major types of classification models. In addition to the WEKA classifiers, we used a custom classifier that assumes that the distribution of the class labels for the test set is the same as for the training set and samples from this distribution without taking features into account.

We consider the classifier presented in Chapter 4 as a reference system to compare with.

7.2.3. Regression

Instead of considering all algorithms from the portfolio together and selecting the one with the best performance, we can also try to predict the performance of each algorithm on a given problem independently and then select the best one based on the predicted performance measures. The downside is that instead of running the Machine Learning once per problem, we need to run it for each algorithm in the portfolio for a single problem.

The advantage of this approach is that instead of trying to learn a model of a particular portfolio, the learned models only apply to individual algorithms. This means that changing the portfolio (i.e. adding or removing algorithms) can be done without having to retrain the models for the other algorithms. Furthermore, the performance model for a single algorithm might be not as complex and easier to learn than the performance model of a portfolio.

Regression is usually performed on the runtime of an algorithm on a problem. Xu et al. (2008) predict the logarithm of the runtime because they

> “…have found this log transformation of runtime to be very important due to the large variation in runtimes for hard combinatorial problems.”

We use the WEKA

- `GaussianProcesses`,
- `LibSVM` (ε and ν),
- `LinearRegression`,
- `REPTree` and
- `SMOreg`

learners to predict both the runtime and the logarithm of the runtime. Again we have tried to be inclusive and add as many different regression learners as possible regardless of our expectations as to their suitability or performance.

We use a modified version of SATzilla (Xu et al., 2008) (denoted SATzilla′) as a reference system to compare with.

7.2.4. Statistical relational learning

Statistical relational learning is a relatively new discipline of Machine Learning that attempts to predict complex structures instead of simple labels (classification) or values (regression) while also addressing uncertainty. An introduction can be found in Getoor and Taskar (2007). For Algorithm Selection, we try to predict the performance ranking of the algorithms from the portfolio on a particular problem.

We consider this approach promising because it attempts to learn the model that is conceptually closest to the model a human would create given perfect knowledge. In the context of algorithm portfolios, we do not care about the performance of individual algorithms, but the relative performance of the algorithms in the portfolio. While this is not relevant for selecting the single best algorithm, many approaches use predicted performance measures to compute schedules according to which to run the algorithms in the portfolio (e.g. Gerevini et al. (2009), Pulina and Tacchella (2009), O'Mahony et al. (2008)). We also expect a good model of this sort to be much more robust with respect to the inherent uncertainty of empirical performance measurements.

We use the support vector machine SVM^{rank} instantiation* of SVM^{struct} (Joachims, 2006). It was designed to predict ranking scores. Instances are labelled and grouped according to certain criteria. The labels are then ranked within each group. We can use the system unmodified for our purposes and predict the ranking score for each algorithm on each problem. We used a value of 0.1 for the tolerance ε that controls the accuracy of the approximation of the optimal solution where the learned model predicts the training data perfectly modulo a certain amount of slack. In cases where the model learner was not able to find such an approximation within one hour, we set $\varepsilon = 0.5$.

To the best of our knowledge, statistical relational learning has never before been applied to Algorithm Selection.

7.3. Evaluation data sets

We evaluate and compare the performance of the approaches mentioned above on five data sets of hard Algorithm Selection problems from the literature. We take three sets from the training data for SATzilla 2009 (Xu et al., 2009). These data come from the area of satisfiability (SAT), where the task is to find a set of assignments to the Boolean variables in a logic formula in conjunctive normal form such that the formula is true or prove that no such set can exist. A logic formula in conjunctive normal form consists only of conjunctions of disjunctions. They consist of SAT instances from three categories – hand-crafted, industrial and random and contain 1181, 1183 and 2308 instances, respectively. The authors use 91 attributes for each instance and

*http://www.cs.cornell.edu/People/tj/svm_light/svm_rank.html

select a SAT solver from a portfolio of 19 solvers[†]. We compare the performance of each of our methodologies with a modified version of SATzilla that only outputs the predictions for each problem without running a presolver or doing any of the other optimisations (SATzilla′). Some of the timeout values in the training data available on the website do not reflect the actual values used in the experiments and after consultation with the authors of SATzilla, we adjusted all timeout values to 3600 seconds.

The fourth data set comes from the Quantified Boolean Formulae (QBF) Solver Evaluation 2010[‡]. A quantified Boolean formula consists of a formula in propositional logic and quantifiers over this formula. The task is again to find a set of assignments that makes the formula true or prove that no such set can exist. The data set consists of 1368 QBF instances from the main, small hard, 2QBF and random tracks. 46 attributes are calculated for each instance and we select from a portfolio of 5 QBF solvers. Each solver was run on each instance for at most 3600 CPU seconds. If the solver ran out of memory or was unable to solve an instance, we assumed the timeout value for the runtime. The experiments were run on a machine with a dual 4 core Intel E5430 2.66 GHz processor and 16 GB RAM. We compare the performance to that of the AQME system, which was trained on parts of this data.

Our last data set is taken from Chapter 4 and selects from a portfolio of two solvers for a total of 2028 constraint problem instances from 46 problem classes with 17 attributes each. We compare our performance with the classifier described in Chapter 4.

For each data set, some of the attributes are cheap to compute while others are extremely expensive. In practice, steps are usually taken to avoid the expensive attributes, such as explicitly excluding them (cf. Chapter 4). Each attribute can be used for each methodology; there is no particular Machine Learning algorithm that requires expensive or allows cheap attributes to be used in particular. More details can be found in the publications that describe the systems with which we compare.

We chose the data sets because they represent Algorithm Selection problems from three areas where the technique of algorithm portfolios has attracted a lot of attention recently. For all sets, reference systems exist with which we can compare. Furthermore, the number of algorithms in the respective portfolios for the data sets is different.

It should be noted that the systems we are comparing against are given an unfair advantage. They have been trained on at least parts of the data that we are using for the evaluation, i.e. not all the data we are using is new to them. The Machine Learning algorithms we use however are given disjoint sets of training and test instances.

[†] http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/
[‡] http://www.qbflib.org/index_eval.php

7.4. Methodology

The focus of our evaluation is the performance of the Machine Learning algorithms. Additional factors that would impact the performance of an Algorithm Selection system in practice are not taken into account. These factors include the time to calculate problem features and additional considerations for selecting algorithms, such as memory requirements.

We measured the performance of the learned models in terms of misclassification penalty, as in the previous chapters. For the classification learners, we attached the maximum misclassification penalty as a weight to the respective problem instance during the training phase.

The handling of missing attribute values was left up to the specific Machine Learning system. We estimated the performance of the learned models using ten-fold stratified cross-validation (Kohavi, 1995). The performance on the whole data set was estimated by summing the misclassification penalties of the individual folds.

For each data set, we used two sets of features – the full set and the subset of the most predictive features. We used WEKA's `CfsSubsetEval` attribute selector with the `BestFirst` search method with default parameters to determine the most predictive features for the different Machine Learning methodologies. We treated SVM^{rank} as a black box algorithm and therefore did not determine the most predictive features for it.

We performed a full factorial set of experiments where we ran each Machine Learning algorithm of each methodology on each data set. We also evaluated the performance with thinned out training data. We randomly deleted 25%, 50% and 75% of the problem-algorithm pairs in the training set. We thus simulated partial training data where not all algorithms in the algorithm portfolio had been run on all problem instances.

To evaluate the performance of the Algorithm Selection systems we compare with, we ran them on the full, unpartitioned data set. The misclassification penalty was calculated in the same way as for the Machine Learning algorithms.

7.4.1. Machine Learning algorithm parameters

We tuned the parameters of all Machine Learning algorithms to achieve the best performance on the given data sets. Because of the very large space of possible parameter configurations, we focussed on the subset of the parameters that is likely to affect the generalisation error, for example parameters to control the pruning of a learned decision tree, the minimum amount of data that is still split in a decision rule learner or the number of folds in internal cross-validations. Tuning the values of all parameters would be prohibitively expensive. The total number of evaluated configurations was 19,032.

Our aim was to identify the parameter configuration with the best performance on *all* data sets. Configurations specific to a particular data set would prevent us

from drawing conclusions as to the performance of the particular Machine Learning algorithm in general. It is very likely that the performance on a particular data set can be improved significantly by carefully tuning a Machine Learning algorithm to it (cf. Chapter 4), but this requires significant effort to be invested in tuning for each data set.

Our intention for the results presented in this chapter is twofold. First, the algorithms that we demonstrate to have good performance can be used with their respective configurations as-is by researchers wishing to build an Algorithm Selection system for search problems. Second, these algorithm configurations can serve as a starting point for tuning them to achieve the best performance on a particular data set. The advantage of the former approach is that a Machine Learning algorithm can be chosen for a particular task with quantitative evidence for its performance already available.

In many approaches in the literature, Machine Learning algorithms are not tuned at all if the performance of the Algorithm Selection system is already sufficient with default parameters. Many researchers who use Machine Learning for Algorithm Selection are not Machine Learning experts.

We used the same methodology for tuning as for the other experiments. For each parameter configuration, the performance in terms of misclassification penalty with the full set of parameters on each data set was evaluated using ten-fold stratified cross-validation. We determined the best configurations by calculating the intersection of the set of best configurations on each individual data set. For four algorithms, this intersection was empty and we used the configurations with the best overall performance, but not necessarily the best performance on an individual data set. This was the case for the classification algorithms `BFTree`, `DecisionTable`, `JRip` and `PART`. For all other algorithms, there was at least one configuration that achieved the best performance on all data sets.

We found that for most of the Machine Learning algorithms that we used, the default parameter values already gave the best performance across all data sets. Furthermore, most of the parameters had very little or no effect; only a few made a noticeable difference. For SVM^{rank}, we found that only a very small number of parameter configurations were valid across all data sets – in the majority of cases, the configuration would produce an error because of invalid combinations of parameters or unimplemented functionality. We decided to change the parameter values from the default for the nine case-based reasoning and classification algorithms below.

AdaBoostM1 We used the `-Q` flag that enables resampling.

DecisionTable We used the `-E acc` flag that uses the accuracy of a table to evaluate its classification performance.

`IBk` with 1, 3, 5 and 10 neighbours We used the `-I` flag that weights the distance by its inverse.

J48 We used the flags `-R -N 3` for reduced error pruning.

JRip We used the `-U` flag to prevent pruning.

PART We used the `-P` flag to prevent pruning.

We were surprised that the use of pruning decreased the performance on unseen data. Pruning is a way of preventing a learned classifier from becoming too specific to the training data set and generalising poorly to other data. One possible explanation for this behaviour is that the concept that the classifier learns is sufficiently pronounced in even relatively small subsets of the original data and pruning over-generalises the learned model and causes a reduction in performance.

7.5. Experimental results

We first present and analyse the results for each Machine Learning methodology and then take a closer look at the individual Machine Learning algorithms and their performance.

The misclassification penalty in terms of the majority predictor for all methodologies and data sets is shown in Figure 7.1. The results range from a misclassification penalty of less than 10% of the majority predictor to almost 650%. In absolute terms, the difference from always picking the best overall algorithm can be from an improvement of more than 28 minutes per problem to a decrease in performance of more than 41 minutes per problem.

At first glance, no methodology seems to be inherently superior. The "No Free Lunch" theorems, in particular the one for supervised learning (Wolpert, 2001), suggest this result. We were surprised by the good performance of the majority predictor, which in particular delivers excellent performance on the industrial SAT data set. The SVM^{rank} relational approach is similar to the majority predictor when it delivers good performance.

Many approaches do not compare their performance with the majority predictor, thus creating a misleading impression of the true performance. As our results demonstrate, always choosing the best algorithm from a portfolio without any analysis or Machine Learning can *significantly outperform* more sophisticated approaches.

Figure 7.2 shows the misclassification penalty in terms of a classifier that learns a simple rule (`OneR` in WEKA) – the data is the same as in Figure 7.1, but the reference is different. This evaluation was inspired by Holte (1993), who reports good classification results even with simple rules. On the QBF and SAT-IND data sets, there is almost no difference. On the CSP data set, a simple rule is not able to capture the underlying performance characteristics adequately – it performs worse than the majority predictor, as demonstrated by the improved relative performance of the other approaches. On the remaining two SAT data sets, learning a simple classification rule improves over the performance of the majority predictor.

The reason for including this additional comparison was to show that there is no simple solution to the problem. In particular, there is no single attribute that

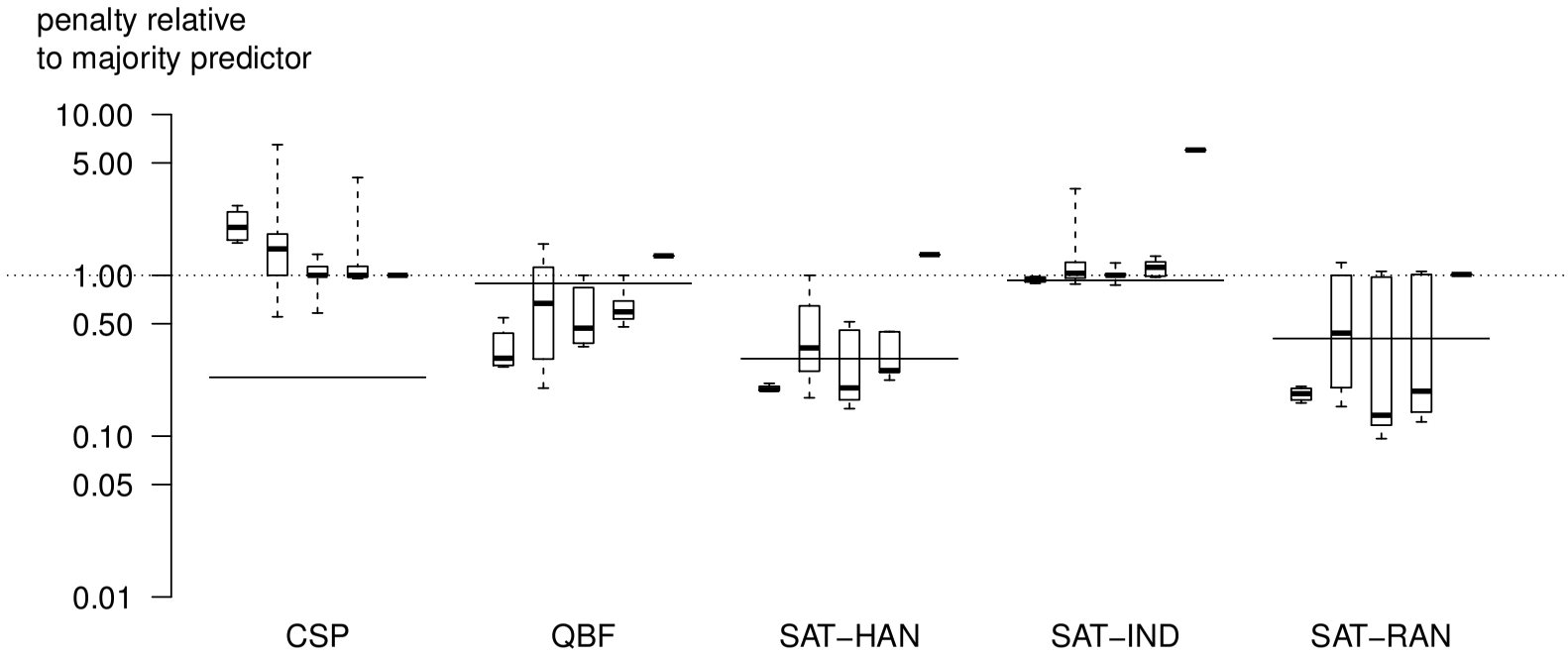


Figure 7.1. Experimental results with full feature sets and training data across all methodologies and data sets. The plots show the 0th (bottom line), 25th (lower edge of box), 50th (thick line inside box), 75th (upper edge of box) and 100th (top line) percentile of the performance of the Machine Learning algorithms for a particular methodology (4 Machine Learning algorithms for case-based reasoning, 19 for classification, 6 for regression and 1 for statistical relational learning). The boxes for each data set are, from left to right, case-based reasoning, classification, regression, regression on the log and statistical relational learning. The performance is shown as a factor of the simple majority predictor, which is shown as a dotted line. Numbers less than 1 indicate that the performance is better than that of the majority predictor. The solid lines for each data set show the performance of the systems we compare with (the classifier from Chapter 4 for the CSP data set, Pulina and Tacchella (2009) for the QBF data set and SATzilla′ for the SAT data sets).

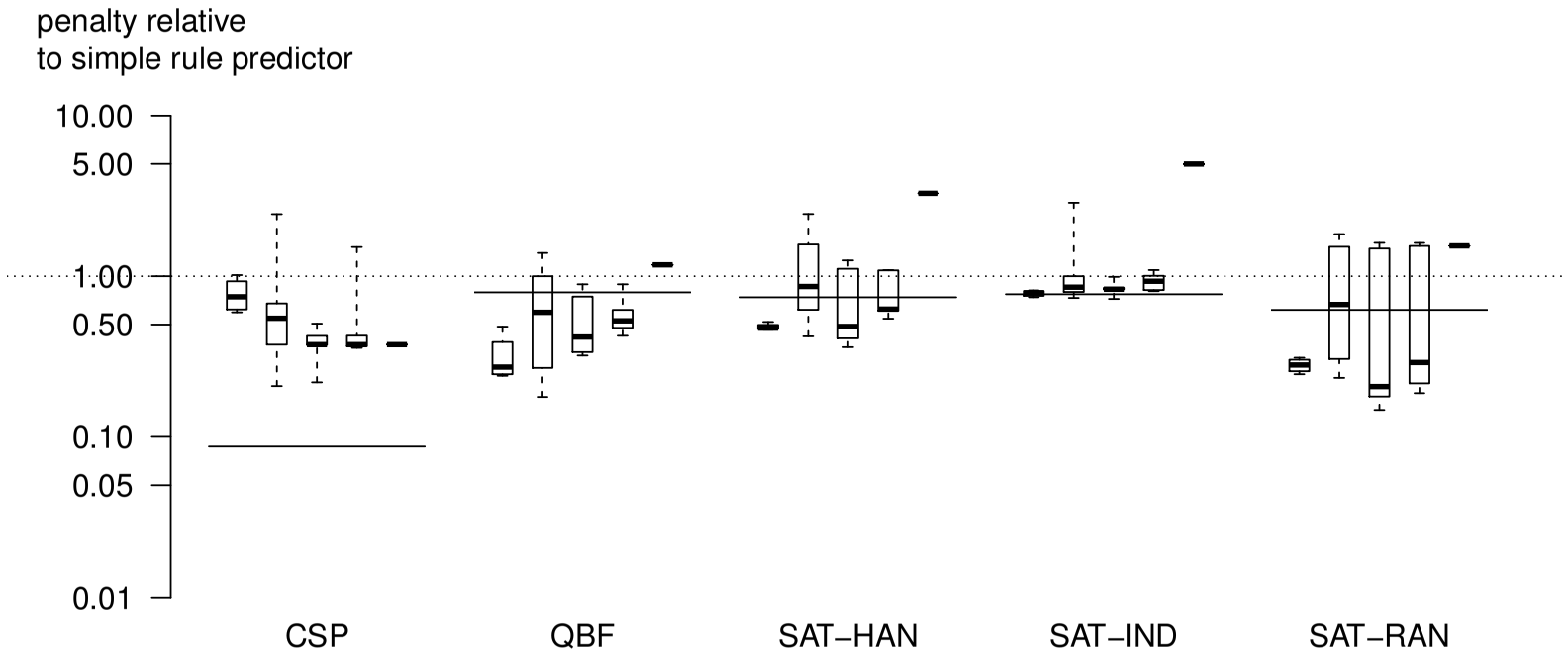


Figure 7.2. Experimental results with full feature sets and training data across all methodologies and data sets. The boxes for each data set are, from left to right, case-based reasoning, classification, regression, regression on the log and statistical relational learning. The performance is shown as a factor of a classifier that learns a simple rule (`OneR` in WEKA), which is shown as a dotted line. Numbers less than 1 indicate that the performance is better than that of the simple rule predictor.

adequately captures the performance characteristics and could be used in a simple rule to reliably predict the best solver to use. On the contrary, the results suggest that considering only a single attribute in a rule is an oversimplification that leads to a deterioration of overall performance. The decrease in performance compared to the majority predictor on some of the data sets bears witness to this.

To determine whether regression on the runtime or on the log of the runtime is better, we estimated the performance with different data by choosing 1000 bootstrap samples from the set of data sets and comparing the performance of each Machine Learning algorithm for both types of regression. Regression on the runtime has a higher chance of better performance – with a probability of ≈67% it will be better than regression on the log of the runtime on the full data set. With thinned out training data the picture is different however and regression on the log of the runtime delivers better performance. We therefore show results for both types of regression in the remainder of this chapter.

Figure 7.3 shows the results for the set of the most predictive features. The results are very similar to the ones with the full set of features. A bootstrapping estimate as described above indicated that the probability of the full feature set delivering results better than the set of the most important features is ≈69%. Therefore, we only consider the full set of features in the remainder of this chapter – it is better than the selected feature set with a high probability and does not require the additional feature selection step. In practice, most of the Machine Learning algorithms ignore features that do not provide relevant information anyway – either explicitly like `J48` by not including them in the generated decision tree, or implicitly like the regression techniques that set the factors for those features to zero.

The effects of thinning out the training data were different across the data sets and are shown in Figure 7.4. On the industrial and random SAT data sets, the performance varied seemingly at random; sometimes increasing with thinned out training data for one Machine Learning methodology while decreasing for another one on the same data set. On the hand-crafted SAT and QBF data sets, the performance decreased across all methodologies as the training data was thinned out while it increased on the CSP data set. Statistical relational learning was almost unaffected in most cases.

There is no clear conclusion to be drawn from these results as the effect differs across data sets and methodologies. They however suggest that, as we are dealing with inherently noisy data, deleting a proportion of the training data may reduce the noise and improve the performance of the Machine Learning algorithms. At the very least, not running all algorithms on all problems because of resource constraints is unlikely to have a large negative impact on performance as long as most algorithms are run on most problems.

The size of the algorithm portfolio did not have a significant effect on the performance of the different Machine Learning methodologies. Our intuition was that as the size of the portfolio increases, classification would perform less well because the learned model would be more complex. At the same time, we expected the perfor-

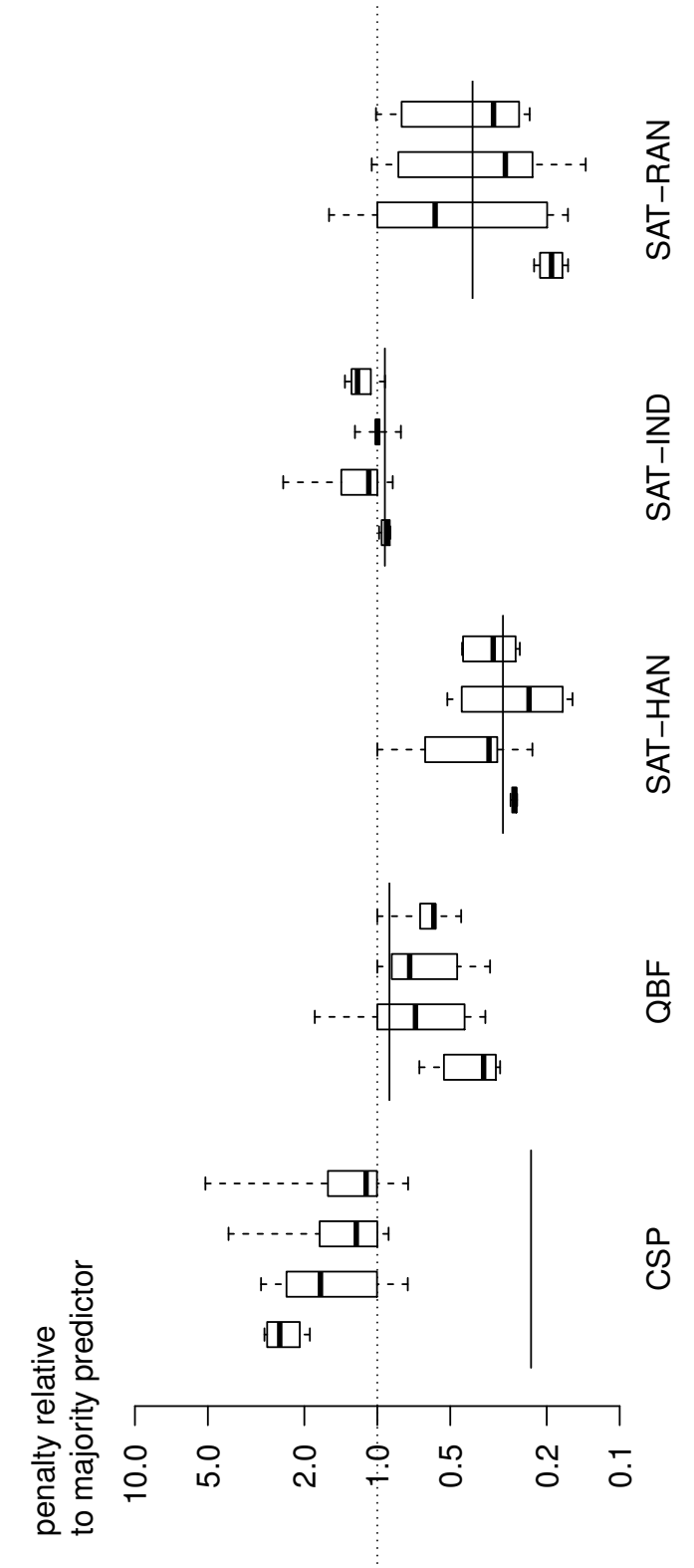


Figure 7.3. Experimental results with reduced feature sets across all data sets. The boxes for each data set are, from left to right, case-based reasoning, classification, regression and regression on the log. We did not determine the set of the most predictive features for statistical relational learning. The performance is shown as a factor of the simple majority predictor, which is shown as a dotted line. For each data set, the most predictive features were selected.

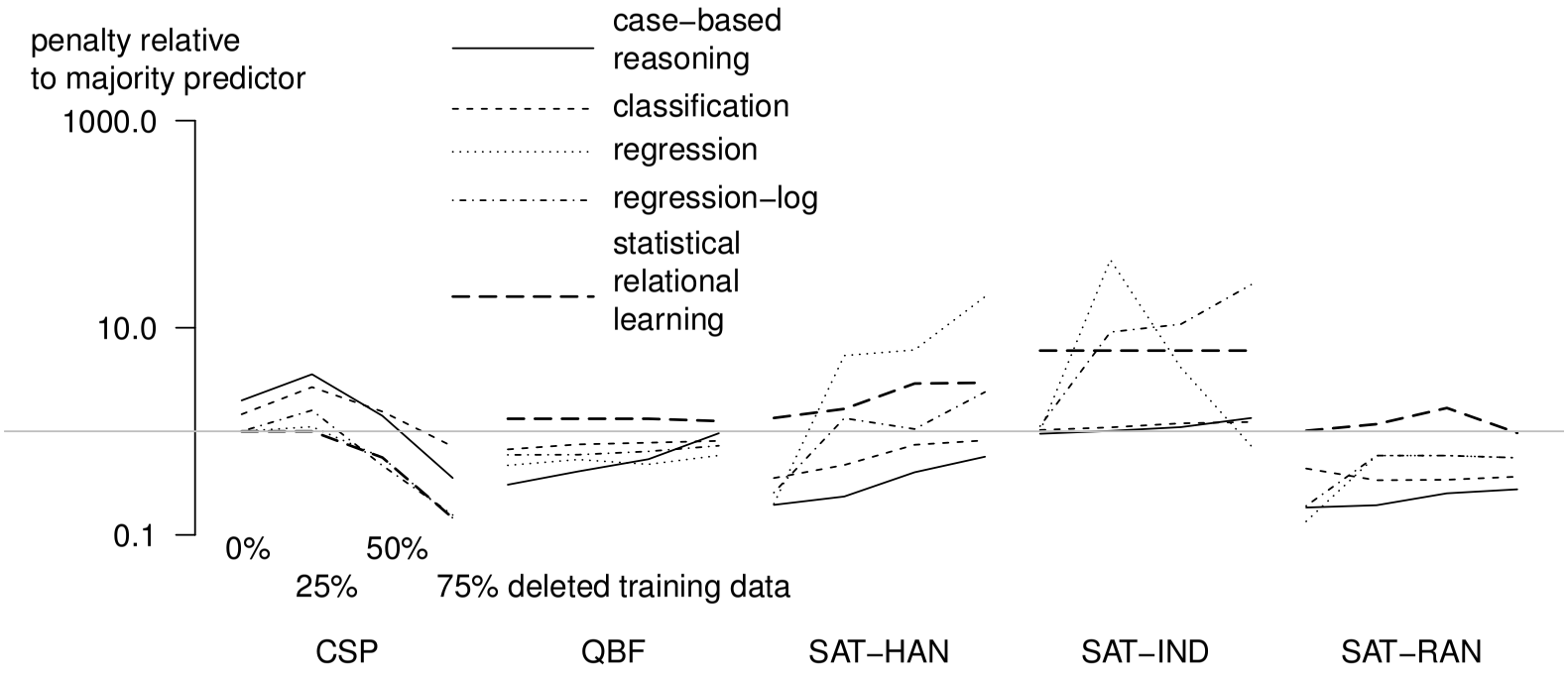


Figure 7.4. Experimental results with full feature sets and thinned out training data across all methodologies and data sets. The lines show the median penalty (thick line inside the box in the previous plots) for 0%, 25%, 50% and 75% of the training data deleted. The performance is shown as a factor of the simple majority predictor which is shown as a grey line. Numbers less than 1 indicate that the performance is better than that of the majority predictor.

methodology	rank with full training data 1	2	3	better than majority predictor	rank 1 with deleted training data 25%	50%	75%
case-based reasoning	**52%**	29%	25%	**80%**	**80%**	**70%**	**39%**
classification	2%	3%	5%	60%	6%	8%	14%
regression	33%	**32%**	**28%**	67%	3%	7%	35%
regression-log	8%	19%	24%	75%	1%	15%	6%
statistical relational learning	6%	16%	18%	0%	10%	0%	5%

Table 7.1. Probabilities for each methodology ranking at a specific place with respect to the median performance of its algorithms and probability that this performance will be better than that of the majority predictor. We also show the probabilities that the median performance of the algorithms of a methodology will be the best for thinned out training data. All probabilities are rounded to the nearest percent. The highest probabilities for each rank are in **bold**.

mance of regression to increase because the difficulty of the learned models does not necessarily increase. In practice however the opposite appears to be the case – on the CSP data set, where we select from only 2 solvers, classification and case-based reasoning perform worse compared with the other methodologies than on the other data sets. As we compared only three different portfolio sizes, there is not enough data from which to draw definitive conclusions.

As it is not obvious from the results which methodology is the best, we again used bootstrapping to estimate the probability of being the best performer for each one. We sampled, with replacement, from the set of data sets and for each methodology from the set of Machine Learning algorithms used and calculated the ranking of the median performances across the different methodologies. Repeated 1000 times, this gives us the likelihood of an average algorithm of each methodology being ranked 1st, 2nd and 3rd. We chose to compare the median performance because there was no Machine Learning algorithm with a clearly better performance than all of the others and algorithms with a good performance on one data set would perform much worse on different data. We used the same bootstrapping method to estimate the likelihood that an average Machine Learning algorithm of a certain methodology would perform better than the simple majority predictor. The probabilities are summarised in Table 7.1.

Based on the bootstrapping estimates, *case-based reasoning* is the methodology most likely to give the best performance and the methodology most likely to deliver good performance in terms of being better than the majority predictor. Regression on the log of the runtime is also very likely to perform better than the majority

predictor. In terms of the rankings however, regression on the runtime without the log transformation is more likely to be second to case-based reasoning.

We observe that the majority classifier still has a non-negligible chance of being as least as good as sophisticated Machine Learning approaches. Its advantages over all the other approaches are its simplicity and that no problem features need to be computed, a task that can further impact the overall performance negatively because of the introduced overhead.

7.5.1. Determining the best Machine Learning algorithm

When using Machine Learning for Algorithm Selection in practice, one has to decide on a specific Machine Learning algorithm rather than methodology. Looking at Figure 7.1, we notice that individual algorithms within the classification and regression methodologies have better performance than case-based reasoning. While having established case-based reasoning as the best overall Machine Learning *methodology*, the question remains whether an individual Machine Learning *algorithm* can improve on that performance.

The `GaussianProcesses` algorithm to predict the runtime has most wins. But how likely is it to perform well in general? Our aim is to identify Machine Learning algorithms that will perform well in general rather than concentrating on the top performer on one data set only to find that it exhibits bad performance on different data. It is unlikely that one of the best algorithms here will be the best one on new data, but an algorithm with good performance on all data sets is likely to exhibit good performance on unseen data. We performed a bootstrap estimate of the probability of an individual Machine Learning algorithm being better than the majority predictor by sampling from the set of data sets as described above. The results are summarised in Table 7.2.

The results confirm our intuition – the two algorithms that *always* perform better than the majority predictor are never the best algorithms while the algorithms that have the best performance on at least one data set – `GaussianProcesses` predicting the runtime and `RandomForest` and `LibSVM` with radial basis function for classification – have a significantly lower probability of performing better than the majority predictor.

Some of the Machine Learning algorithms within the classification and regression methodologies outperform the majority predictor with a probability of less than 50% and do not appear in Table 7.2 at all. It is likely that the bad performance of these algorithms contributed to the relatively low rankings of their respective methodologies compared with case-based reasoning, where all Machine Learning algorithms exhibit good performance. The fact that the individual algorithms with the best performance do not belong to the methodology with the highest chances of good performance indicate that choosing the best individual Machine Learning algorithm regardless of methodology is likely to result in better overall performance.

The good performance of the case-based reasoning algorithms was expected based

Machine Learning methodology	algorithm	better than majority predictor
classification	`LADTree`	**100%**
regression	`LinearRegression`	**100%**
case-based reasoning	`IBk` with 1 neighbour	81%
case-based reasoning	`IBk` with 5 neighbours	81%
case-based reasoning	`IBk` with 3 neighbours	80%
case-based reasoning	`IBk` with 10 neighbours	80%
classification	`DecisionTable`	80%
classification	`FT`	80%
classification	`J48`	80%
classification	`JRip`	80%
classification	`RandomForest`	80%
regression	`GaussianProcesses`	80%
regression-log	`GaussianProcesses`	80%
regression-log	`SMOreg`	80%
regression-log	`LibSVM` ε	80%
regression-log	`LibSVM` ν	80%
classification	`REPTree`	61%
classification	`LibSVM` radial basis function	61%
regression	`REPTree`	61%
regression	`SMOreg`	61%
classification	`AdaBoostM1`	60%
classification	`BFTree`	60%
classification	`ConjunctiveRule`	60%
classification	`PART`	60%
classification	`RandomTree`	60%
regression-log	`LinearRegression`	60%
regression-log	`REPTree`	60%
classification	`J48graft`	59%
regression	`LibSVM` ν	59%

Table 7.2. Probability that a particular Machine Learning algorithm will perform better than the majority predictor on the full training data with the full feature sets. We only show algorithms with a probability higher than 50%, sorted by probability. All probabilities are rounded to the nearest percent.

on the results presented in Table 7.1. All of the algorithms of this methodology have a very high chance of beating the majority predictor. The nearest-neighbour approach appears to be robust with respect to the number of neighbours considered.

The good results of the two best algorithms seem to contradict the expectations of the "No Free Lunch" theorems. It has been shown however that the theorems do not necessarily apply in real-world scenarios because the underlying assumptions may not be satisfied (Rao et al., 1995). In particular, the distribution of the best algorithms from the portfolio to problems is not random – it is certainly true that certain algorithms in the portfolio are the best on a much larger number of problems than others. Xu et al. (2008) for example explicitly exclude some of the algorithms in the portfolio from being selected in certain scenarios.

7.6. Ensemble classification

The results in Chapter 6 showed that using an ensemble of several classifiers is an effective means of increasing the robustness of an Algorithm Selection system while maintaining a high level of performance. The main question left unanswered in the investigation into ensemble classification was which classifiers to use in the ensemble.

The results presented in this chapter enable us to answer this question. Based on the probabilities given in Table 7.2 on the previous page, we can choose classifiers to use in an ensemble. As an example, we choose the three classifiers `LADTree`, `FT` and `IBk` with 5 neighbours. The decision to include `LADTree` was motivated by its performance while the other classifiers were chosen to have a variety of different Machine Learning techniques in the ensemble. Other classifiers with a similar probability of being better than the majority predictor would have been reasonable choices, too. We include only three algorithms in the ensemble to limit the overhead introduced by training and running additional classifiers, as in Chapter 6.

The performance the example ensemble classifier and its constituent Machine Learning algorithms achieve is shown in Figure 7.5 on the facing page. The ensemble is always better than the majority predictor and on two of five data sets outperforms its constituent classifiers. On the other hand, its performance is worse than that of any of the individual classifiers on two other data sets. Overall, it does provide better performance than the `LADTree` classifier however, the only Machine Learning algorithm in the ensemble that is always better than the majority predictor.

The results confirm those presented in Chapter 6. Using an ensemble of classifiers instead of a single one increases the robustness and achieves good performance in general. As mentioned before, the choice of classifiers in the ensemble is crucial to its success. The results presented in this chapter, namely the probabilities of being better than the majority predictor in Table 7.2 on the previous page, equip us with the empirical evidence to make an informed decision as to the composition of an ensemble and be confident of its performance in practice.

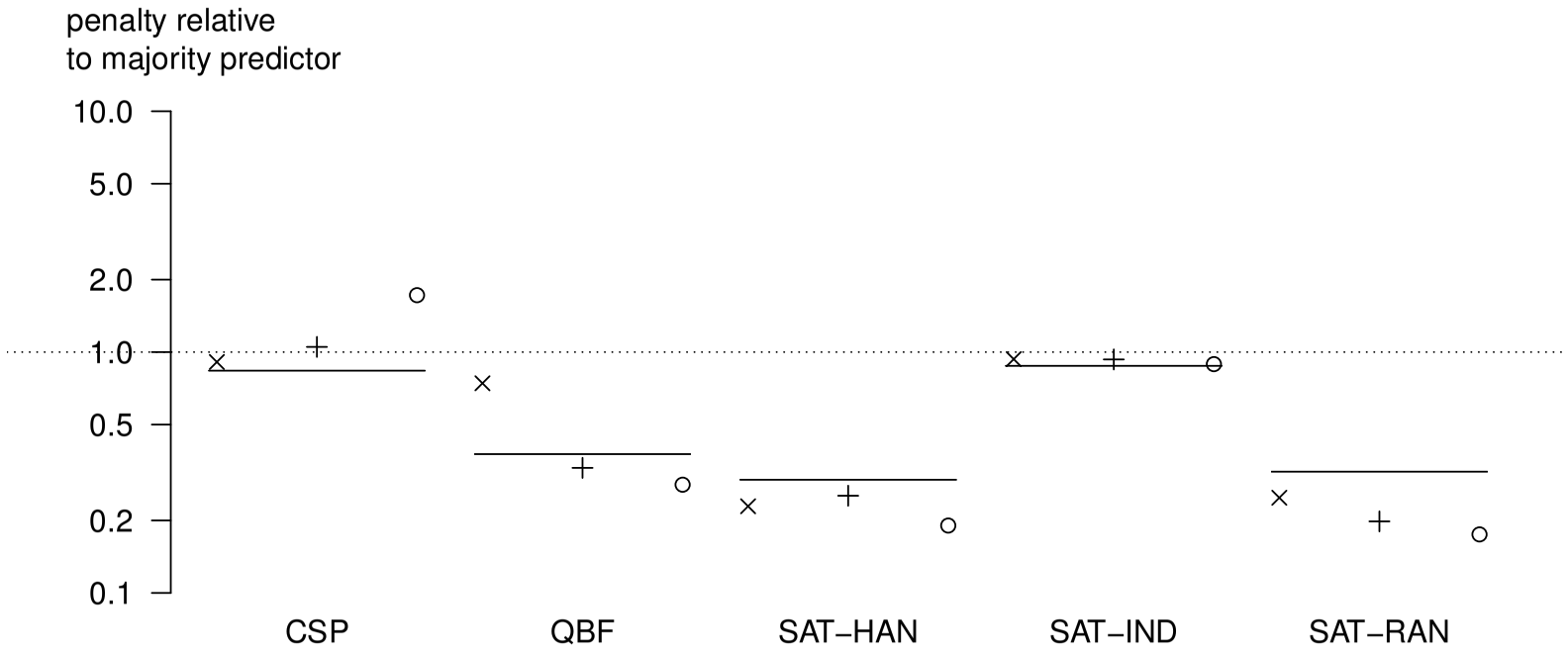


Figure 7.5. Experimental results with the ensemble classifier consisting of three classifiers with good performance. The crosses, pluses and circles show the performance of `LADTree`, `FT` and `IBk` with 5 neighbours, respectively. The solid line shows the performance of the ensemble. The performance is shown as a factor of the simple majority predictor which is shown as a dotted line. Numbers less than 1 indicate that the performance is better than that of the majority predictor.

7.7. Summary and contributions

In this chapter, we investigated the performance of five different Machine Learning methodologies and many different Machine Learning algorithms for Algorithm Selection on five data sets from the literature. We compared the performance not only among these methodologies and algorithms, but also with existing Algorithm Selection systems. To the best of our knowledge, we presented the first large-scale, quantitative comparison of Machine Learning methodologies and algorithms applicable to Algorithm Selection. We furthermore applied statistical relational learning to Algorithm Selection for the first time.

We used the performance of the simple majority predictor as a baseline and evaluated the performance of everything else in terms of it. This is a less favourable evaluation than found in many publications, but gives a better picture of the real performance improvement of algorithm portfolio techniques over just using a single algorithm. This method of evaluation clearly demonstrates that sophisticated (and computationally expensive) approaches can have inferior performance compared with simply choosing the best individual algorithm in all cases.

Our evaluation also

- showed the performance in terms of a simple rule learner,
- evaluated the effects of using only the set of the most predictive features instead of all features,
- looked at the influence the size of the algorithm portfolio has on the relative performance of the Machine Learning methodologies
- and quantified performance changes caused by thinning out the set of training data.

We demonstrated that methodologies and algorithms that have the best performance on one data set do not necessarily have good performance on all data sets. A non-intuitive result of our investigation is that deleting parts of the training data can help improve the overall performance, presumably by reducing some of the noise inherent in the empirical runtime data.

Based on a statistical simulation with bootstrapping, we gave recommendations as to which algorithms are likely to have good performance. We identified *linear regression* and *alternating decision trees* as implemented in WEKA as particularly promising types of Machine Learning algorithms. In the experiments done for this chapter, they always outperform the majority predictor. We focussed on identifying Machine Learning algorithms that deliver good performance in general. It should be noted that neither of these algorithms exhibited the best performance on any of the data sets, but the Machine Learning algorithms that did performed significantly worse on other data.

We furthermore demonstrated that *case-based reasoning* algorithms are very likely to achieve good performance and are robust with respect to the number of past cases

considered for any given new datum. Combined with the conceptual simplicity of nearest-neighbour approaches, it makes them a good starting point for researchers who want to use Machine Learning for Algorithm Selection, but are not Machine Learning experts themselves.

We showed that the default parameters of the Machine Learning algorithms in WEKA already achieve very good performance and in most cases no tuning is required. While we were able to improve the performance in a few cases, finding the better configuration carried a high computational cost. In practice, few researchers will be willing or able to expend lots of resources to achieve small improvements.

Finally, we confirmed the benefit of using an ensemble of classifiers as demonstrated in Chapter 6 by using the presented empirical evidence to compose an ensemble of three classifiers that achieves good performance on all data sets.

The main contributions of this chapter are as follows.

- The first large-scale performance comparison of Machine Learning algorithms on several Algorithm Selection data sets.
- An investigation of the effects different ways of preprocessing data (like determining the set of the most predictive features) have on the performance in practice.
- The application of statistical relational learning to Algorithm Selection.
- Demonstrating the relatively good performance of the simple majority predictor.
- The identification of linear regression, alternating decision trees and nearest-neighbour classifiers as Machine Learning algorithms that are likely to achieve good performance on Algorithm Selection problems.
- The confirmation of the benefits of ensemble classification and the demonstration of good performance of an ensemble consisting of Machine Learning algorithms recommended here.

8

Conclusions and future work

In this dissertation, we presented several contributions to the field of Algorithm Selection in general and Algorithm Selection for combinatorial search problems in particular. We have given some motivation for Algorithm Selection, presented a detailed survey of the literature on Algorithm Selection and immediately related fields. We have also presented case studies that apply Algorithm Selection techniques to new problem domains and bring out some of the difficulties often encountered when doing Algorithm Selection in practice.

The material presented throughout this dissertation is centred around the question posed in the introduction,

How should we do Algorithm Selection for combinatorial search problems in practice?

We have established the importance of this question and the difficulty of answering it. We have investigated it in detail in two specific contexts. We have proposed ways of answering or helping to answer it and shown the effectiveness of these ways.

Starting from the central question of this dissertation, the thesis defended can be formulated as follows. There are Machine Learning techniques that can be applied to Algorithm Selection to decrease the background knowledge required to perform it in practice and achieve good performance.

In this chapter, we will provide concrete answers to the central question. This chapter summarises the contents of the dissertation, reiterates the contributions, draws conclusions and presents an outlook on future work.

8.1. Summary

Chapter 2 presented an investigation that gave quantitative evidence for the need for Algorithm Selection. Comparing several state of the art constraint solvers, the differences in performance were found to be significant. Although there was a solver that had the best performance most of the time, there was clearly scope for improvement by selecting a different solver in some situations. However, it was not obvious how to make this decision.

Chapter 3 had a look at the vast amount of literature available on Algorithm Selection. It classified and analysed dozens of publications according to specific criteria. One of its contributions was to establish the diversity of approaches in the literature and that there is no approach that is agreed to be the best one, making it hard to choose in practice.

Chapter 4 presented a case study for a specific problem domain and used a decision tree learner to decide whether to use lazy learning in constraint solving. It demonstrated steps that can be taken to ensure that an Algorithm Selection system is effective and efficient. It also showed that we can not only improve the performance of constraint solving significantly in this context, but also improve our understanding of the problem domain by inspecting the learned decision tree.

Chapter 5 employs the selection of the implementation of the `alldifferent` constraint as another case study for using Machine Learning to solve the Algorithm Selection Problem. In addition to using techniques described in the literature, it made a series of decisions that depend on each other to arrive at the most suitable implementation for solving a given problem. It furthermore presented empirical data that shows the effectiveness of a cost model that assigns a weight to each problem and that using a reduced feature set does not affect performance negatively in general.

Chapter 6 showed that ensemble classification, a technique borrowed from Machine Learning, is an effective way of mitigating the problem of having to choose the best way to do Algorithm Selection. As alluded to earlier, this is one of the biggest obstacles to be overcome when using Algorithm Selection in practice. An ensemble of classifiers whose decisions are combined by majority vote exhibits not only much more robust and predictable performance, but also performance that is equal to that of well-performing individual classifiers. In some cases, the performance of the ensemble even exceeded the performance of the best individual classifier.

Chapter 7 investigated the performance of five different Machine Learning methodologies and many Machine Learning algorithms for Algorithm Selection on five data sets from the literature. It addressed the crucial lack of such a comparison in the available literature and complements Chapter 6, which shows ways of avoiding to select an individual best technique, but still requires the selection of a few. Chapter 7 also takes a look at several issues related to the selection of a Machine Learning technique. It gives recommendations as to which techniques should be considered for Algorithm Selection based on extensive empirical evidence. Finally, it presents empirical evidence that the proposed Machine Learning algorithms with good performance can be used in a classifier ensemble to achieve the benefits demonstrated in Chapter 6.

8.2. Contributions

The main contributions of this dissertation are as follows.

- The identification of Machine Learning techniques that are likely to perform well in the context of Algorithm Selection problems based on large-scale empirical evidence (Chapter 7). Either linear regression to predict the run time of an algorithm on a problem or alternating decision trees should be used, both as implemented in the WEKA Machine Learning toolkit.

- The identification and evaluation of ensemble classification as a promising technique for reducing the amount of Machine Learning expertise required to perform Algorithm Selection while maintaining a high level of performance improvements (Chapters 6 and 7). In addition, ensembles increase the robustness in the sense that bad performance of one constituent can be alleviated by the other constituents.

- The first large-scale survey and evaluation of Machine Learning techniques applicable for performing Algorithm Selection for combinatorial search problems. Chapter 3 shows the large variety of techniques in the literature on one hand and the almost complete lack of comparisons of different techniques on the other.

- The use of decision trees to improve our understanding of the issues underlying an Algorithm Selection problem (Chapter 4). Although decision trees have been used frequently in the literature, using them for this purpose has not been described.

- The demonstration of the feasibility of making multi-level decisions that depend on each other (Chapter 5). Even when combining the output of several Machine Learning stages, each with an associated uncertainty, the resulting system is still effective and efficient.

- Two case studies (Chapters 4 and 5) that apply Algorithm Selection techniques to new problem domains and raise issues that are addressed in other parts of this dissertation.

Apart from these main contributions, the following ones have been made.

- The establishing of a cost model that weights training examples according to their importance in Machine Learning for Algorithm Selection. Although similar models have been used in the literature, their effectiveness has never been evaluated in a publication.

- The evaluation of the generality of a decision tree through cross-validation across problem classes. This technique complements traditional cross-validation, which partitions at random. Partitioning by problem class enables us to estimate the generalisation error more realistically.

- The use of bootstrapping to determine the probability of a certain performance level of a technique used for Algorithm Selection. Bootstrapping leverages the empirical evidence to provide estimates of performance in general.
- The application of statistical relational learning to Algorithm Selection. There is no previous application of this relatively new field of Machine Learning in the literature. Although the results lag behind other techniques, statistical relational learning is, not least because of the higher information content of its predictions, a promising technique.

The focus of this dissertation has been not so much on presenting yet another technique that improves the state of the art in a specific Algorithm Selection scenario, but to address the fundamental problem that many researchers face.

We are now in a position to answer the central question of this dissertation, based on the research presented in the previous chapters.

- Systems that perform Algorithm Selection for combinatorial search problems should use linear regression to predict the performance of each algorithm on a problem and make the decision to choose which one based on that prediction or use alternating decision trees to directly predict the best algorithm. Nearest-neighbour classifiers are also a reasonable choice. The exact choice of method will depend on other factors such as what information is required to decide.
- These techniques or the other ones recommended in Chapter 7 can be combined in a Machine Learning ensemble to make the Algorithm Selection system more robust. While the performance of one of the constituents of the ensemble can be better than that of the ensemble in some cases, the empirical evidence presented in Chapters 6 and 7 suggests that over large sets of diverse data the performance of the ensemble is better than that of an individual constituent.

8.3. Scope and limitations

There are a lot of factors that affect the performance of Algorithm Selection systems and a lot of different methods that can be applied to tackle the Algorithm Selection Problem. Chapter 3 explores the vast amount of literature and reinforces this point. Not all of the different techniques can be investigated in this dissertation; its scope is necessarily limited to a selection of issues and techniques.

The selection of issues and techniques was motivated by their relevance to Algorithm Selection as demonstrated by the available literature. Combinatorial search problems are the dominant application domain for Algorithm Selection. Most of the techniques investigated here have been used in this or a similar form. They also cover a broad range of different Machine Learning techniques. The aim of the selection was to present research that will be useful in many different scenarios and contexts.

This dissertation focusses on the practical aspects of Algorithm Selection and empirical evidence for improvements. This is partly because this is where the focus

of research over the last decades has been, but mostly because there is no theoretical framework suitable for an evaluation of the performance of different techniques for Algorithm Selection. The main question of this dissertation was, as alluded to earlier, motivated by the fact that it is very difficult to select a technique to use in a specific scenario based on the available literature.

The automatic tuning of algorithms is relevant to Algorithm Selection and some of the literature has been described in Chapter 3. It is however not one of the foci of this dissertation. The concentration on algorithms that have not been tuned automatically does not necessarily incur a loss of generality for the results though. Taking the selection of the most appropriate implementation of the `alldifferent` constraint as an example (Chapter 5), the different versions could be seen as different settings for a parameter. Similarly, different parameter configurations could be seen as different algorithms to be selected from. An Algorithm Selection system would implicitly perform the tuning by selecting the algorithm with the most appropriate configuration.

The only case that is not covered is when the space of possible configurations is too large to be represented in this manner. There are however systems such as Hydra (Xu et al., 2010) and ISAC (Kadioglu et al., 2010) that deal with this scenario.

8.4. Future work

One of the main avenues for future work is the implementation and use of the techniques described in this dissertation in a real system. Appendix B describes a framework for the automatic generation of constraint solvers that are specialised for a given problem. There exists a preliminary implementation that already shows a lot of promise.

A major part of the described framework is an Algorithm Selection system to choose the most appropriate implementations of components of the target constraint solver. An implementation of this part can draw on the material presented in this dissertation – a series of decisions that depend on each other will have to be made (cf. Chapter 5), specific Machine Learning techniques will have to be used (Chapter 7) and the system should be robust (Chapter 6).

Generating a constraint solver is a difficult task that requires many interdependent decisions to be made with respect to the algorithms and data structures to use. It is possible that it is infeasible to apply traditional Algorithm Selection techniques in this context. If this problem arose, a possible solution would be to employ statistical relational learning, which has been investigated in Chapter 7.

Applying more sophisticated Machine Learning techniques to Algorithm Selection is another major research direction. This may be necessary in order to be able to perform Algorithm Selection in scenarios such as the one described above at all, or may be able to provide significantly higher performance than current state of the art systems.

The automatic generation of constraint solvers tailored to a specific problem will

be a major step forward in Algorithm Selection systems. The potential for practical performance improvements is much higher than in existing Algorithm Selection systems because of the much more fine-grained decision making. Efficiently and effectively exploring the very large space of possible solvers is novel and challenging. Analysing problems to solve with regards to a wide range of diverse implementation decisions will require the modification and extension of existing techniques.

Apart from this main extension of the present work, there are a number of other directions that could be explored. Even though this dissertation makes major contributions to basic Algorithm Selection research, there is still plenty of scope for additional work.

Chapter 7 establishes a number of Machine Learning algorithms likely to exhibit good performance in Algorithm Selection scenarios. There is however no explanation for this that could provide pointers as to what factors have an effect on the performance. While the results are based on extensive empirical evidence, investigating the underlying issues may provide pointers how to improve the performance even further or how to modify techniques for this purpose.

As established throughout this dissertation, there is plenty of empirical evidence for the difficulty of doing Algorithm Selection in practice. There is however no notion of the difficulty or "hardness" of an Algorithm Selection problem. Is it difficult because the problem domains are mostly drawn from applications with a high complexity class? Do the features we use for Machine Learning carry enough information for us to make a decision? Or is there another factor that we have not considered yet?

One of the major contributions of this dissertation was to show that ensemble classification is a technique whose application to Algorithm Selection has several benefits. There are other techniques in Machine Learning for which this could be true, for example boosting to complement ensemble classification or principal component analysis as a preprocessing step. Investigating to what extent such techniques can be applied to improve Algorithm Selection is another promising avenue for future research.

Another challenge is how to compose ensembles. Ideally, the constituent Machine Learning algorithms would complement each other such that in cases where one delivers poor performance, the ensemble can compensate for it. This is a problem similar to constructing portfolios for Algorithm Selection itself and some of the techniques used there may be applicable in this case as well.

Beyond this, the current work could be extended by applying Algorithm Selection to new problem domains. Similarly, new techniques could be developed to make Algorithm Selection decisions. Statistical relational learning in particular is a promising direction to pursue because of the richer prediction output that could enable new approaches.

In conclusion, there are many promising avenues for further research. The research presented in this dissertation has started exploring some of them as well as uncovering new directions. The results described in the previous chapters are immediately

applicable not only to practical Algorithm Selection systems, but also to researchers pushing the boundaries of the field.

A

Summary of relevant literature

The literature overview in the table below is sorted chronologically for lack of a more meaningful order. Cutting out the individual entries and rearranging them according to some other criteria would probably be informative, but is entirely unnecessary as the modern interwebs provide us with ways of doing so that do not involve scissors. A version of this table that can be sorted according to various criteria is available at http://www.cs.st-andrews.ac.uk/~larsko/thesis/.

citation	domain	features	predict what	predict how	type	portfolio
Langley (1983b,a)	search	past performance	heuristic	hand-crafted and learned rules	offline and online	dynamic
Carbonell et al. (1991)	planning	problem domain features and search statistics	control rules	explanation-based rule construction	online	dynamic
Gratch and DeJong (1992)	planning	problem domain features and search statistics	control rules	probabilistic rule construction	online	dynamic
Smith and Setliff (1992)	software design	features of abstract representation	algorithms and data structures	simulated annealing	offline	static
Aha (1992)	Machine Learning	problem features	best algorithm	learned rules	offline	static
Brodley (1993)	Machine Learning	problem and algorithm features	best algorithm for recursive subsets	rules	offline	static
Kamel et al. (1993)	differential equations	past performance and problem features	algorithm	hand-crafted rules	offline	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Minton (1993b,a, 1996)	constraints	runtime performance	heuristic	hand-crafted and learned rules	offline	semi-static
Cahill (1994)	software design	problem features	algorithms and data structures	frame-based knowledge base	offline	static
Tsang et al. (1995)	constraints	problem features	-	-	-	static
Brewer (1995)	software design	runtime data	algorithms, data structures and their parameters	statistical model	offline	static
Weerawarana et al. (1996), Joshi et al. (1996)	differential equations	problem features	runtime performance	Bayesian belief propagation and neural nets	offline	static
Borrett et al. (1996)	constraints	search statistics	switch algorithm?	hand-crafted rule	online	static, static order
Allen and Minton (1996)	constraints/SAT	probing	runtime performance	hand-crafted rule	online	static
Sakkout et al. (1996)	constraints	search statistics	switch algorithm?	hand-crafted rule	online	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Gomes and Selman (1997b,a)	constraints	past performance	best heuristic	statistical model	offline	semi-static
Cook and Varnell (1997)	parallel search	probing	set of search strategies	decision trees, Bayesian classifier, nearest neighbour and neural net	online	static
Fink (1997, 1998)	planning	statistics of past performance	time bounds in schedule	statistical model offline and online and regression on problem size	offline	static
Lobjois and Lemaître (1998)	branch and bound	probing	runtime performance	hand-crafted rule	online	static
Caseau et al. (1999)	vehicle routing problem	runtime performance	heuristic	genetic programming	offline	static
Howe et al. (1999)	planning	problem features	round-robin schedule based on runtime performance	linear regression	offline	static
Terashima-Marín et al. (1999)	scheduling	problem and search features	heuristic	genetic algorithm	offline	semi-static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Wilson et al. (2000)	software design for linear systems	problem features	data structures	nearest neighbour	offline	static
Beck and Fox (2000)	job shop scheduling	problem feature changes during search	algorithm scheduling policy	hand-crafted rule	online	static
Brazdil and Soares (2000)	classification	past performance	ranking	distribution model	offline	static
Lagoudakis and Littman (2000)	order selection and sorting	problem features	remaining cost for each sub-problem	MDP	online	static
Sillito (2000)	constraints	probing	cost of solving problem	statistical model	offline	static
Gomes and Selman (2001)	constraints and mixed integer programming	past performance	best heuristic	statistical model	offline	semi-static
Cowling et al. (2001)	scheduling	problem features	best heuristic	hand-crafted rule and learned weights	online	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Epstein and Freuder (2001), Epstein et al. (2002)	constraints	variable characteristics	heuristic	weights and hand-crafted rules	offline and online	semi-static
Lagoudakis and Littman (2001)	DPLL branching rules	problem features	remaining cost for each sub-problem	MDP	online	static
Nareyek (2001)	optimisation	search statistics	expected utility of heuristic	non-stationary reinforcement learning of weights	offline and online	static
Horvitz et al. (2001)	constraints	problem and problem generator features, search statistics	runtime performance and parameters	Bayesian models	offline and online	static
Borrett and Tsang (2001)	constraints	problem features and search statistics	redundant constraints to add	hand-crafted rule	offline	-

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Little et al. (2002)	logic puzzles	problem graph features	problem model transformations for runtime performance	nearest neighbour	offline	-
Petrovic and Qu (2002)	scheduling	problem features	heuristic	case-based reasoning	offline	static
Leyton-Brown et al. (2002)	winner determination problem	problem features	problem hardness	several forms of regression	offline	static
Fukunaga (2002, 2008)	SAT	variable characteristics	heuristic	genetic programming	offline	semi-static
Vrakas et al. (2003)	planning	problem features	parameters	classification association rules	offline	dynamic
Guo (2003)	sorting and probabilistic inference	problem features	best algorithm	decision tree, naïve Bayes, Bayesian network, meta-learning	offline	static
Watson (2003)	job-shop scheduling	problem features and search statistics	local search algorithm	statistical model	offline and online	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Gebruers et al. (2004)	bid evaluation problem	problem and problem graph features	solution method	nearest neighbour	offline	static
Guerri and Milano (2004)	bid evaluation problem	problem and problem graph features	solution method and algorithm	decision trees	offline	static
Beck and Freuder (2004)	scheduling	probing	best algorithm	hand-crafted rules	offline	static
Nudelman et al. (2004), Xu et al. (2007b, 2008)	SAT	problem features	runtime performance	ridge regression, lasso regression, SVMs, Gaussian processes	offline	static
Carchrae and Beck (2004, 2005)	job shop scheduling	probing and search statistics	length or exploration phase and switch algorithm?	Bayesian classifier and reinforcement learning	offline and online	static
Soares et al. (2004)	Machine Learning	problem features	ranking of SVM kernel widths	nearest neighbour	offline	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Guo and Hsu (2004)	most probable explanation problem	problem features	best algorithm	decision trees, naïve Bayes rules, Bayes networks, meta-learning techniques	offline	static
Gagliolo et al. (2004)	genetic algorithms	past performance	schedule	linear model	online	static
Demmel et al. (2005)	linear algebra	problem features	algorithm	multivariate Bayesian decision rule	offline	static
Gebruers et al. (2005)	constraints	problem features	problem model and best solution strategy	nearest neighbour, decision trees and statistical	offline	static
Petrik (2005)	SAT	past performance	schedule based on predicted runtime	analytic and MDP	offline and online	static
Cicirello and Smith (2005)	scheduling	past performance	best heuristic	max k-armed bandit model	online	static
Gagliolo and Schmidhuber (2005)	genetic algorithms	past performance	schedule	neural nets	online	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Armstrong et al. (2006)	procedure calls	runtime performance	switch algorithm?	reinforcement learning	online	static
Gagliolo and Schmidhuber (2006b)	SAT and auction winner determination problem	past performance	schedule	bandit problem model	online	static
Roberts and Howe (2006)	planning	problem features	round-robin schedule	decision trees	offline	static
Hough and Williams (2006)	optimisation	problem, algorithm and environment features	algorithm	ensembles of decision trees and SVMs	offline	static
Bhowmick et al. (2006)	linear systems	problem features	algorithm	boosting and alternating decision trees	offline	static
Hutter et al. (2006)	stochastic local search in SAT	problem features	runtime performance and parameters	ridge regression	offline	semi-static
Xu et al. (2007a)	SAT	problem features	satisfiability and runtime performance	sparse multinomial logistic regression and ridge regression	offline	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Pulina and Tacchella (2007, 2009)	QBF	problem features	schedule	decision trees, decision rules, logistic regression and nearest neighbour	offline and online	static
Samulowitz and Memisevic (2007)	QBF	problem features	best heuristic and confidence values	multinomial logistic regression	offline and online	static
Wu and van Beek (2007)	scheduling	-	portfolio of backtracking algorithms with deadline	case-based reasoning	offline	dynamic
Streeter et al. (2007)	planning	past performance	schedule	statistical models	offline and online	static
Roberts and Howe (2007), Roberts et al. (2008)	planning	problem features	runtime and probability of success	32 different algorithms	offline	static
Streeter and Smith (2008)	SAT, integer programming, planning	problem features	schedule	statistical models	offline and online	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
O'Mahony et al. (2008)	constraints	problem features	schedule	nearest neighbour	offline	static
Kuefler and Chen (2008)	linear systems	problem features and search statistics	algorithm	reinforcement learning	online	static
Wei et al. (2008)	SAT	search statistics	heuristic	hand-crafted rule	online	static
Nikolić et al. (2009)	SAT	problem features	search policy	nearest neighbour	offline	static
Stamatatos and Stergiou (2009)	constraints	probing	propagation method	clustering	offline	static
Stergiou (2009)	constraints	search statistics	propagation method	clustering	online	static
Arbelaez et al. (2009)	constraints	problem features and search statistics	search strategy	SVMs	online	static
Haim and Walsh (2009)	SAT	problem features	restart strategy and satisfiability	ridge regression and logistic regression	offline	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Bhowmick et al. (2009)	linear systems	problem features	algorithm	nearest-neighbour, alternating decision trees, naïve Bayes, SVM	offline	static
Gerevini et al. (2009)	planning	past performance	macro actions and round-robin schedule	performance simulations for different schedules	offline	static
Leite et al. (2010)	Machine Learning	past performance and probing	ranking of classification algorithms	statistical model	offline and online	static
Silverthorn and Miikkulainen (2010)	SAT	past performance	runtime performance	latent class models	offline	static
Stern et al. (2010)	QBF and combinatorial auctions	problem and algorithm features	best algorithm	Bayesian model	offline and online	static
Garrido and Riff (2010)	dynamic vehicle routing problem	runtime performance	combination of low-level heuristics	genetic algorithms	online	dynamic
Domshlak et al. (2010)	planning	search statistics	heuristic	naïve Bayes classifier	online	static

continued on next page

citation	domain	features	predict what	predict how	type	portfolio
Kadioglu et al. (2010)	SAT, MIP and set covering	problem features	best algorithm	clustering	offline	dynamic
Tolpin and Shimony (2011)	constraints	search statistics	heuristic	hand-crafted rule	online	static
Malitsky et al. (2011)	SAT	problem features	best algorithm	nearest neighbour	offline	static
Kadioglu et al. (2011)	SAT	problem features	schedule	nearest neighbour	offline	static
Kroer and Malitsky (2011)	SAT and constraints	problem features	best algorithm	clustering	offline	dynamic

Table A.1. Summary of the Algorithm Selection literature.

B

Dominion – A constraint solver generator

This appendix presents Dominion, a framework for the automatic generation of constraint solvers tailored to the problem they are to solve. The implementation efforts completed so far are described. The system is still in an early stage and not all of the components are present or complete.

B.1. Overview of the Dominion system

There are two main parts that make up the Dominion system – the analyser and the generator. Roughly speaking, the analyser takes the specification of a constraint problem to be solved and generates a solver architecture specification that is then synthesised into a constraint solver by the generator. A high-level overview of the system, its components and how they work together is given in Figure B.1 on the next page.

The constraint problem to be solved is specified in the Dominion Input Language and transformed into a problem component. The main task of the problem component is to specify the other constraint solver components required to solve the problem. To achieve this, component specifications from a component library are consulted. The problem component, along with the original problem specification, is passed to the analyser, which decides the most suitable implementations for the individual components of the target solver. The resulting solver architecture is passed to the solver generator, which, taking the specified implementations from the component library, synthesises a constraint solver tailored to the analysed problem. The

Parts of the material here have been published previously in: Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, and Ian Miguel. Modelling Constraint Solver Architecture Design as a Constraint Problem. In *Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, 2011.
and: Dharini Balasubramaniam, Lakshitha de Silva, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Dominion: An Architecture-driven Approach to Generating Efficient Constraint Solvers. In *9th Working IEEE/IFIP Conference on Software Architecture*, June 2011.
Parts also appear in: Dharini Balasubramaniam, Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. An Automated Approach to Generating Efficient Constraint Solvers. In *34th International Conference on Software Engineering*, June 2012.
The contributions of the author of this dissertation are listed on page xix et seqq.

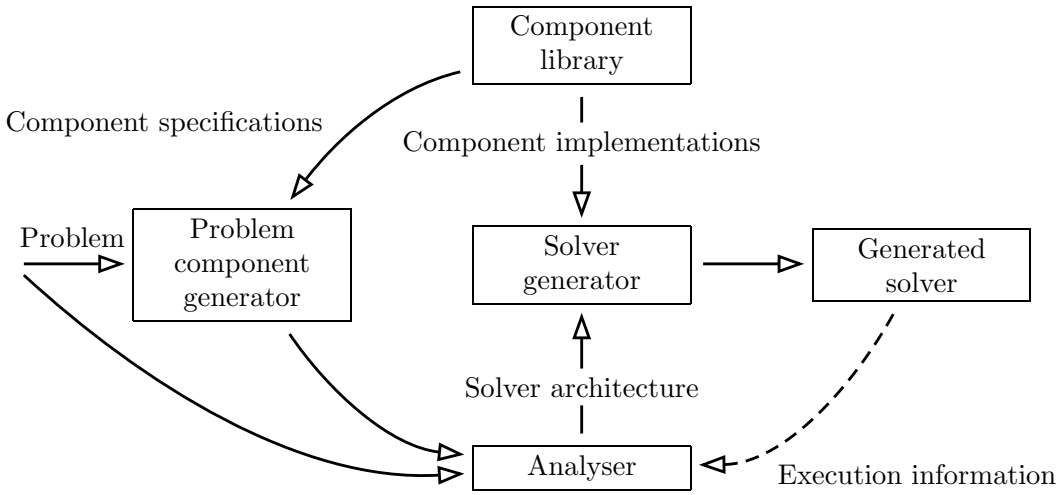


Figure B.1. Overview of the Dominion system.

execution of this solver can be monitored and information about it passed back to the analyser. Based on this information, the analyser can decide to change the implementation of components to create a more efficient solver.

The analyser is the part that this dissertation is most relevant to. Given a constraint problem, the task of the analyser is to determine the solver configuration that is most efficient for solving the problem. In addition to assessing the suitability of individual components, the analyser also has to take into account the complex restrictions on how components can be connected. The choice of a particular implementation for a component may impact the entire solver configuration – it may require or preclude the use of other components elsewhere in the solver, which in turn may impose similar restrictions. The task of finding a valid configuration for a constraint solver from a set of components is a non-trivial task.

B.2. Related work

A lot of the literature that is relevant to Dominion has already been examined in Chapter 3. Only some of the more important approaches are mentioned here again and put in the specific context of Dominion. Furthermore, there is some literature relevant to the automatic generation of software in general and constraint solver or constraint solver components in particular that deserves mentioning.

One of the earliest examples of a system that attempts to generate constraint solvers tailored to a specific problem is MULTI-TAC (Minton, 1996), which configures and compiles a constraint solver for a specific set of problems. It is written in LISP and performs ad-hoc customisation of a base constraint solver limited to a few characteristics.

KIDS (Smith, 1990) is a more general system that also uses LISP to synthesise efficient algorithms from an initial specification. The approach is knowledge-based, i.e. the user supplies the knowledge required to generate an efficient algorithm for the specific problem. Refinements of the initial specification are limited to a number of generic transformation operations. The Dominion approach is more general and, crucially, relies on almost no background knowledge. Westfold and Smith (2001) use KIDS to synthesise efficient constraint solvers. They rely on reformulation and specialisation of the constraints however and do not consider the other components of a solver. Srivastava and Kambhampati (1998) use KIDS to synthesise planners, but rely on explicitly-specified domain knowledge to do so.

RT-Syn (Smith and Setliff, 1992) uses simulated annealing to select the best from a set of abstract data structures and algorithms and synthesises a programme from the selected abstract descriptions. First, all algorithms and data structures that meet the requirements specified by the problem to solve are chosen. Then RT-Syn analyses all candidates and greedily selects the best one. The analysis is based purely on the abstract representation.

Cahill (1994) builds a knowledge base to aid with the construction of numerical algorithms from subcomponents. He models dependencies between components, but relies (at least partially) on knowledge input manually by human experts and does not report any results demonstrating the effectiveness of the system.

Brewer (1995) builds statistical models to select data layout and sorting algorithm for iterative partial differential equation solvers. He also automatically tunes the parameters of the selected algorithms. Dominion follows the same general idea, but is not restricted to a small number of decisions and takes dependencies of components into account. It selects implementations for *every* component for which more than one implementation is available and models the ramifications of that choice on the rest of the software.

There are a number of approaches that do not do Algorithm Selection, but investigate the automatic generation of algorithms from high-level descriptions. The EasySyn++ system (Di Gaspero and Schaerf, 2007) automatically generates stochastic local search algorithms from a number of templated components. Again the synthesis is limited to a number of key components and does not encompass all aspects of the solver. Aeon (Monette et al., 2009) is a similar system for the automated generation of scheduling algorithms.

Van Hentenryck and Michel (2007) describe how to generate efficient implementations from high-level descriptions of local search procedures. They focus on the high-level choices and abstract from low-level details. Elsayed and Michel (2010) propose a framework that uses rules written by human experts to determine the implementation for particular components. Schulte and Tack (2008) describe how to automatically generate variations of specific solver components.

B.3. Challenges for the automatic generation of constraint solvers

A simple constraint solver is not a fundamentally complex piece of software. The necessary components are a representation of variables, a representation of constraints, a search engine that decides heuristically what decisions to make, a propagation engine that allows constraints to act on the consequences of those decisions and a state maintenance facility that allows changes as a result of search and propagation, and reverses those changes on backtracking.

Each of these components can be implemented in a simple way – state maintenance for example can be as simple as copying all data structures before changes are made and then copying them back on backtracking. Certain propagation engines may be algorithmically complex to obtain optimal performance, but this need not be a problem for the construction of solvers from components.

The main source of complexity in constraint solver design comes from the need to optimise the target solver for efficiency. First, the ramifications of an individual choice can be large because of restrictions it places on other choices. Second, a constraint problem can require thousands of variables and many constraints per variable. There may be different optimal choices for different variable or constraint components; we can either make a compromise choice for all of those implementations and accept the possibility of suboptimal performance, or allow for many different component implementations and accept a greatly increased overall complexity. While it is possible to provide a fine-grained level of choice for algorithmic parts in static software that is not assembled from components for each problem, this flexibility comes at the expense of inefficiencies.

These inefficiencies can be as simple as unnecessary code being in an executable, but more important ones arise where time and memory are used to maintain superfluous data structures which are needed only to support component choices not currently being used. While for an individual component the performance penalty incurred because of this may be small, the overall impact in a constraint solver that calls this component thousands of times per second would be significant. The nature of constraint solving, and indeed solving any $\mathcal{NP}$-complete problem, is such that the difference between an optimal and suboptimal solver can be as pronounced as finding a solution in a matter of seconds or in more than a lifetime.

For an investigation into the impact some implementation choices have on performance, see Chapter 2.

B.4. Specification languages

The development of the Dominion system involved the specification and implementation of two languages – one to specify constraint problems to solve and one to specify the architecture of constraint solvers. Both are described below; the reader

```
language Dominion 0.1
given n: int {3..}
dim queens[n]: int
find queens[..]: int {1..n}

such that

alldifferent alldiff(queens[..])
diagonals1 [ sumneq([queens[j], j-i], queens[i]) |
    i in {0..n-2}, j in {i+1..n-1} ]
diagonals2 [ sumneq([queens[j], i-j], queens[i]) |
    i in {0..n-2}, j in {i+1..n-1} ]
```

Figure B.2. The n-Queens problem specified in the Dominion Input Language.

is referred to the cited publications for more detail. Backus-Naur-Form specifications of the languages can be found in Appendices C and D.

The n-Queens problem, introduced in Chapter 2, will be used as an example for illustrating how the languages are used in practice.

B.4.1. Problem specification – Dominion Input Language

The Dominion Input Language is modelled on Essence (Frisch et al., 2008). There are a number of important differences however, particularly in the way arrays are handled. Furthermore, the Dominion Input Language supports set and list comprehensions, a feature that Essence does not have.

In brief, the main constructs of the language are parameters, constants, decision variables and constraints. Parameters are supplied externally and instantiate problems to solve where the specification depends on parameters. Constants are names given to specific values. Decision variables are given a domain that the generated constraint solver draws assignments from. The constraints restrict the possible assignments of domain values to decision variables. Parameters, constants and decision variables can be defined as matrices. All of the language constructs but parameters can be part of a comprehension.

An example of a constraint problem specification in the Dominion Input Language is given in Figure B.2.

The complete specification of the Dominion Input Language can be found in Appendix C. The language is described in detail by Gent et al. (2009).

B.4.2. Architecture specification – Grasp

Grasp is a generic software architecture description language. The advantages of using a generic architecture description language include available tools for checking

architecture descriptions for consistency and that people without a background in constraint programming are able to work with it.

The elements of the language that are relevant to this appendix are described below.

templates Templates are the high-level elements of the language that describe components. A single template can describe a memory manager for example. Templates may take parameters when they are instantiated to customise their behaviour further.

requires/provides Describe things a template needs and offers for other templates to use. A memory manager for example provides a facility for storing and retrieving data. This facility could be required by a variable to keep track of its domain.

properties Properties characterise components beyond the generic facilities they provide. A Boolean variable for example would have the property that the size of the domain is at most two.

checks Check statements model the interdependencies between components and restrictions of customisations of a component. A component that implements a specific constraint would place restrictions on the parameters it can be customised with (i.e. the variables that it constrains) by limiting the domain size for example.

The check statements of Grasp provide much power and flexibility. Only a small subset is required to model the architecture of a constraint solver though. The relevant parts are explained below.

A subsetof B Asserts that set B contains all the elements of set A. It is used to ensure that a certain implementation has a specific set of properties and provides a specific set of facilities. It can also be used to ensure that an implementation does *not* have a property or facility.

A accepts B Asserts that B is accepted as A, e.g. if A is the parameter given to the implementation of a constraint and B is a variable implementation, it makes sure that the constraint can be put on variables of that type.

Apart from the components that describe the building blocks of a solver, there is a top-level component that describes the problem to be solved. It specifies the types of variables and constraints needed and which constraint implementation needs to work with which variable implementation.

The description of the constraint solver consists of a library of solver component implementations and the problem component. The library of solver component implementations is not specific to any constraint problem to be solved by the generated solver and describes the space of possible implementations for the components of any

```
@Dominion(Filename="../../models/queens.dominion.hpp")
@Dominion(Classname = "DominionProblemClassFactory")
template DominionProblem() {
    provides IProblemClassFactory;
    requires IConstraintStoreFactory csf;
    requires IPropagatorFactory_alldiff alldifferent;
    requires IPropagatorFactory_sumneq diagonals1;
    requires IPropagatorFactory_sumneq diagonals2;
    requires IDiscreteVarFactory queens;

    check queens.properties() subsetof [(DomainType, 'bound')];
    check alldifferent.param(1) accepts (queens +
                                         [(DomainType, 'bound')]);
    check diagonals1.param(1) accepts (queens
                                       [(DomainType, 'bound')]);
    check diagonals1.param(2) accepts (queens
                                       [(DomainType, 'bound')]);
    check diagonals2.param(1) accepts (queens
                                       [(DomainType, 'bound')]);
    check diagonals2.param(2) accepts (queens
                                       [(DomainType, 'bound')]);
}
```

Figure B.3. The n-Queens problem component specified in Grasp.

solver. The problem component encodes the requirements for solving a particular constraint problem.

Figure B.3 gives an example of the specification of a problem component in Grasp.

The complete specification of Grasp can be found in Appendix D. The language is described in detail by Balasubramaniam and de Silva (2011).

B.5. Configuration of a valid solver

The synthesis of a constraint solver from components is a configuration problem. One of the earliest approaches to solving configuration problems as constraint problems is by Mittal and Falkenhainer (1990) and proposes dynamic constraint problems that introduce new variables as the requirements for configured components become known. They require special constraints that express whether a variable is still relevant to the partially solved problem based on the assignments made so far.

Sabin and Freuder (1996) propose solving configuration problems as composite constraint satisfaction problems where values for variables can be constraint problems themselves. Stumptner et al. (1998) introduce the constraint-based configuration system COCOS. It requires several extensions of the standard constraint paradigm as well. Mailharro (1998) proposes a constraint formulation that inte-

grates concepts from object-oriented programming. His approach relies on many of the concepts introduced in earlier work and infinite-sized domains for variables. Hinrich et al. (2004) use object-oriented constraint satisfaction for modelling configuration problems. They then transform the constraint model into first order logic sentences and find a solution using a theorem solver.

The requirements of a component naturally map to variables in a constraint problem that we want to find assignments for. The domain of each of those variables consists of the components that provide the facility required, i.e. the possible implementations. Each implementation variable has a set of provides and properties attached to it. The set of provides is necessary because an implementation may provide more than the one main facility that would be required by another component. If a variable is assigned a value that determines its implementation, it *must* provide all the facilities and have all the properties that this implementation provides and has and it *must not* provide any other facilities or have any other properties. We therefore add constraints to ensure that a component variable has a certain property or provide if and only if it is assigned an implementation that has this property or provide.

There are several cases we need to consider for converting the check statements of Grasp into constraints. The first case is of the form `list subsetof properties/provides`. This requires a component implementation to provide a list of facilities or have a set of properties. The translation into constraints is straightforward; we simply require the things in `list` to be in the set of `properties/provides`. The second case of the form `properties/provides subsetof list`. This is the opposite of the previous case and *forbids* the properties/provides that are not listed explicitly. The translation into constraints is analogous to the previous case.

The final case deals with the `accepts`. The general requirement encoded is that if a parameter to an implementation requires a certain property or facility, the implementation of the parameter must provide it. The corresponding constraints are implications that require properties and provides of an implementation that might be used as a parameter to be set if they are set for the parameter.

B.5.1. Conditional variables and constraints

The variables and constraints mentioned so far are only valid at the top level, i.e. for the problem component. We need additional constructs that encode the requirements that arise if a component is implemented in a certain way. The variables and constraints to encode the requirements take the same form as above, but they have prerequisites that need to be true in order for them to become relevant.

We chose an explicit representation of the prerequisites where the conditional variables encode them in their names. The names of the variables that model the requirements for an implementation of a component not at the top level are prefixed by the implementation choices for the top-level components. The constraints on these variables can be encoded as an implication, e.g. if component x is implemented as

an A, its first parameter needs to have property Y. The name of the variable that models this first parameter would have a prefix that indicates that the superior component x is implemented as an A. The left-hand side of the implication is a conjunction of the implementation decisions made in the prerequisites.

B.5.2. Solving the configuration problem

The constraint formulation of the configuration problem is solved using the Minion (Gent et al., 2006a) solver. Despite the large number of variables and constraint generated even for relatively simple architectures, Minion is able to find a solution to the problem within a fraction of a second. Most of the constraints and variables are not relevant in large parts of the search space because they depend on specific assignments to other variables.

The solution of the constraint problem is mapped to the chosen implementations for components based on the variable names and domain values. The resulting complete solver architecture describes a valid solver, but not necessarily an efficient one.

B.6. Analyser

The analyser in its current implementation performs a basic analysis of the problem to solve. It constructs symbolic expressions that describe the features of the problem. These symbolic expressions are simplified with the DoCon computer algebra system (Mechveliani, 2001). Performing the analysis in this manner enables us to analyse both constraint problem instances and problem classes that depend on parameters whose value is not known at the time of analysis.

Determining the features of a problem constitutes a first step towards using Machine Learning to make decisions as to which component implementations to use. Almost all Machine Learning approaches rely on some kind of feature of the input problem to make their predictions.

The current implementation of the analyser however relies only on features based on execution information to make the decision which implementation to choose for which component and uses a hill climbing approach in the space of candidate solver specifications. Given a current state A, we search the neighbourhood of A in a random order. As soon as a state B that is better than A is found, the hill climber moves to state B and starts exploring its neighbourhood.

Section B.5 describes the process of creating a valid solver specification by solving a constraint problem using the Minion solver. The hill climbing algorithm sits above this and adjusts the variable and value ordering of the constraint problem to guide Minion towards different valid solver specifications.

For each candidate specification, a solver is generated, compiled and run on a set of problems with a time limit. A solver is considered better than a previously found

solver if it either solved more problems within the time limit or the same number of problems in less time.

B.7. Generator

The solver architecture chosen by the analyser is passed to the generator. It finds the specified component implementations in the component library using the location and file name attached to each component specification. It then

- includes the component files required by the chosen architecture,
- instantiates the included components and parameters as appropriate and
- generates code to read runtime parameters, perform initialisations and begin the execution of the solver.

The translation from the Grasp solver architecture to a solver implementation is straightforward given the component-based design of the system and the decisions made and information recorded by other parts of the Dominion system earlier in the process.

B.8. Experimental evaluation

In this section, we compare the performance of solvers generated by the Dominion framework to Minion version 0.12. The experiments were run on an 8-core Intel Xeon E5430 server with a clock speed of 2.66GHz. We used the six problem classes below. Some of them are described in CSPLib (Gent and Walsh, 1999) and some of them have been used for the evaluation of the impact of design decisions on performance in Chapter 2. For problems that are not described in either of those places, a reference is given.

- **n-Queens** The n-Queens problem (cf. Chapter 2).
- **BIBD** The Balanced Incomplete Block Design problem (CSPLib problem 28).
- **Golomb Ruler** The problem of proving optimality of known optimal Golomb Rulers (a variation of CSPLib problem 6).
- **Graceful Graphs** The problem of finding graceful labellings of graphs (Petrie and Smith, 2003).
- **NMR** The problem of finding non-monochromatic rectangles (Fenner et al., 2010).
- **Magic Square** The problem of finding magic squares (CSPLib problem 19).

For all problems with the exception of Graceful Graphs, we exit after finding the first solution. For Graceful Graphs, we searched for all solutions. We compared a Dominion-generated solver with Minion on several instances of the above problem classes. We chose a range of parameter settings where both solvers would finish within one hour of CPU time. In total, we compared Dominion to Minion on 12 n-Queens problem instances, 8 BIBD instances, 5 Golomb Ruler instances, 3 Graceful Graph instances, 5 NMR instances and 3 Magic Square instances.

For each problem class, we ran the hill climbing analyser 10 times. For n-Queens, NMR and Graceful Graphs, each run produced a solver that was able to solve at least one instance of the training set within the imposed time limit of 10 CPU seconds. For Golomb Ruler and Magic Square, 9 of the 10 runs produced such a solver. However for BIBD only 4 of the 10 runs produced a solver that was able to solve at least one instance. Improving this process to find good solvers faster and with a higher probability is a direction of current research.

Figure B.4 summarises the behaviour of hill climbing analyser runs on the n-Queens problem. The analyser is able to incrementally improve upon the solver found at the random starting point and finds a solver that solves all instances quickly. Flat sections of a line show the analyser iterating though parts of the neighbourhood where the generated solvers perform no better than the current best.

B.8.1. Results

Figure B.5 compares the times taken by Dominion and Minion. Overall, Dominion shows promising speed improvements of up to several orders of magnitude. Figure B.6 shows the memory usage for Dominion and Minion as determined by the maximum resident set size. Here Dominion improves over Minion across all problem instances.

For both n-Queens and Magic Square, the times measured for the Dominion solver were zero except on the two largest instances of n-Queens. The largest instance of n-Queens was solved by Dominion in 0.01 seconds and by Minion in 1459 seconds – an improvement of five orders of magnitude. For both problem classes, Minion used between 35.6 MiB and 37 MiB memory. Dominion on the other hand used significantly less; between 3.5 MiB and 5.7 MiB memory. Dominion is also consistently faster than Minion on BIBDs, with increasing performance gains as problem size increases. Dominion again improves over Minion in terms of memory use. For the largest instance of Golomb Ruler, Dominion was slightly more than 2.5 times faster than Minion and Minion used over 6 times the memory. The picture for the other Golomb Ruler instances was similar. The Graceful Graph instances showed almost identical performance of both solvers in terms of CPU time, but Dominion used significantly less memory.

NMR is the only class where the Dominion-generated solver is slower than Minion. On the largest instance, Minion is 1.51 times faster. Dominion is substantially more efficient in terms of memory however, requiring 760 MiB whereas Minion takes 2909

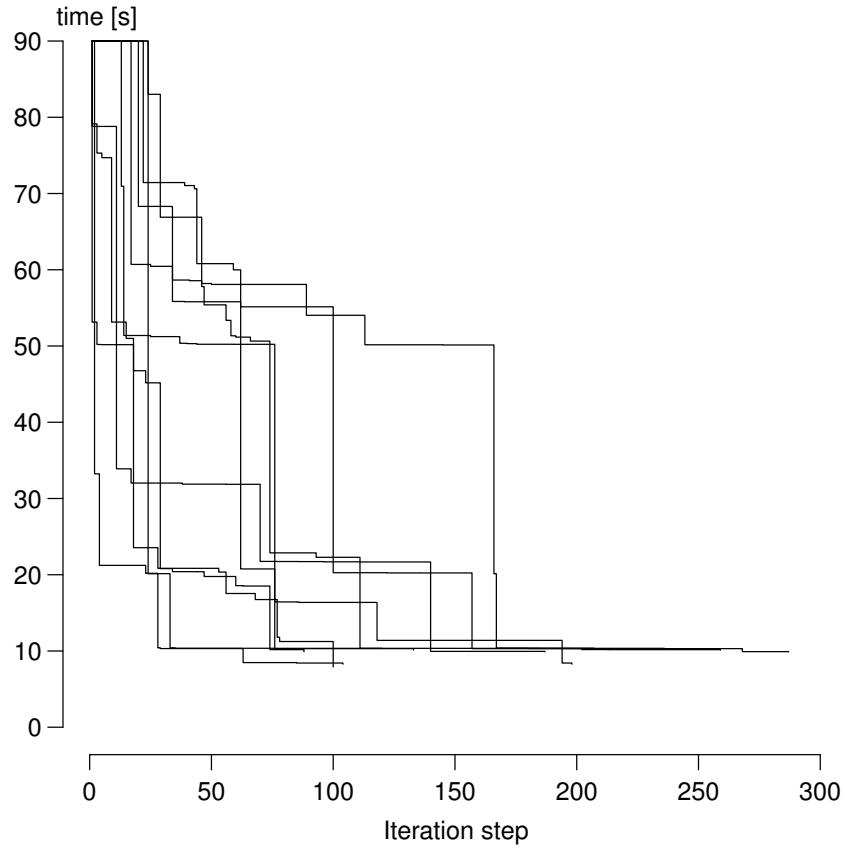


Figure B.4. Performance improvements achieved during analyser runs on n-Queens. The y-axis shows total time for solving the given set of problems, including the timeout if a problem was not solved within the allocated time. The x-axis shows iterations of the hill climbing analyser. Each line shows the progress of a different analyser run.

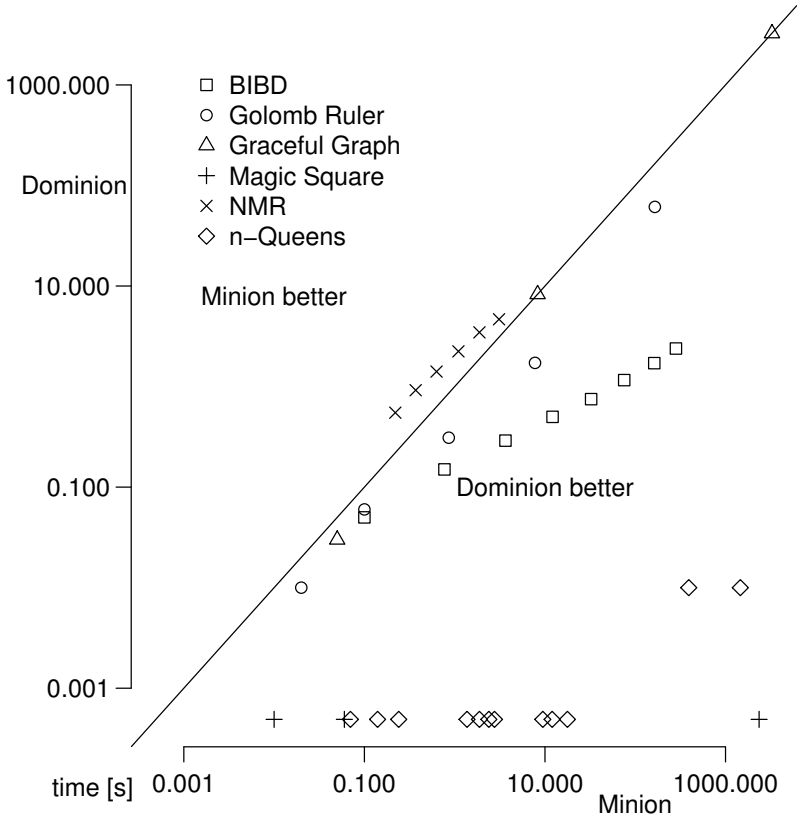


Figure B.5. CPU times for Dominion and Minion on the benchmark problems. The reported times are the median of three runs. The diagonal line denotes the boundary of equal performance for both solvers.

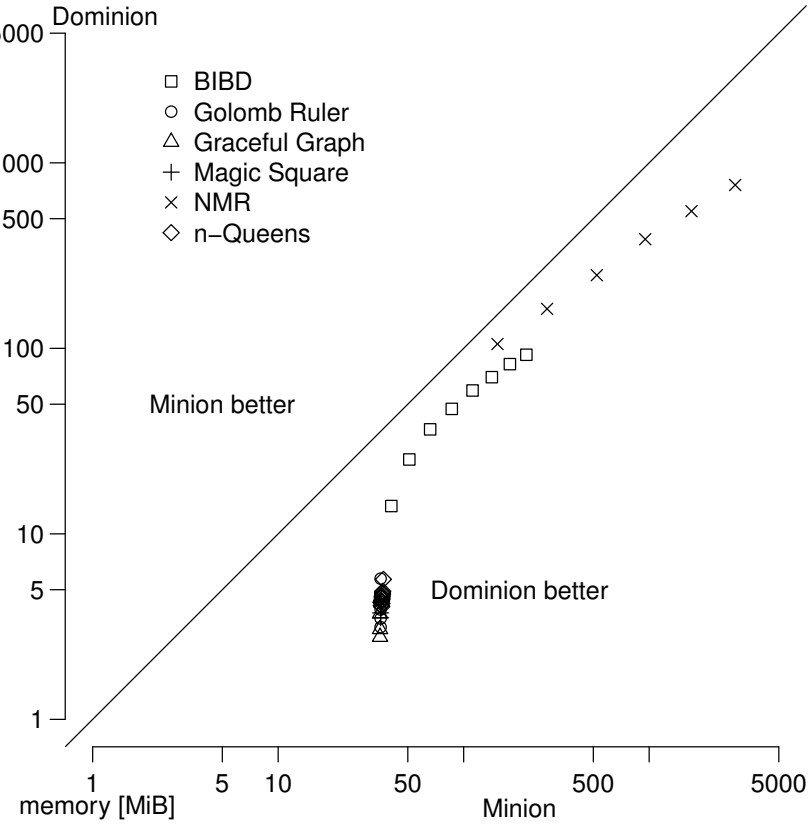


Figure B.6. Memory usage for Dominion and Minion on the benchmark problems. The reported memory numbers are the median of three runs. The diagonal line denotes the boundary of equal performance for both solvers.

MiB and using consistently less memory on all instances.

These results demonstrate that the Dominion framework, even in its current preliminary implementation, is capable of achieving substantial performance improvements over a state of the art traditional solver, using no domain-specific knowledge.

B.9. Summary

This appendix introduced a framework for the automatic generation of constraint solvers tailored to solving a particular problem. It described the implementation efforts to date and presented the results of a preliminary experimental evaluation.

The Dominion approach represents a significant advancement of the current state of the art. Algorithm portfolios were introduced as a means of combining the strengths of different algorithms and alleviate their weaknesses. This idea was combined with the automated tuning of the algorithms in the portfolio to improve the performance even further. Dominion is the next step – algorithms are synthesised from a library of building blocks and tailored to the problem to solve. This, similar to the introduction of automatic tuning, significantly increases the potential performance improvements.

Instead of limiting the improvements to high-level decisions that are controlled through parameters, Dominion enables improvements to be gained by low-level implementation decisions and eliminates the overhead introduced by a highly configurable static architecture. While the automated generation of software is considerably more complex than tuning parameters, the potential benefits are much higher as well.

Despite the preliminary and incomplete nature of the implementation of the system, it is already capable of achieving performance improvements over a state of the art constraint solver. One of the areas where the current implementation is most lacking is the analyser, which uses only execution information at the moment. The research results presented in the rest of this thesis will provide the basis for more sophisticated implementations that are able to efficiently determine the most suitable constraint solver implementation for a given problem.

C

BNF of Dominion Input Language

```
<DominionModel> ::= <Preamble> <Constraints>

<Preamble> ::= "language" "Dominion" <Version>
               (<Given> | <Letting> | <Find> |
                <LetFindAliasComprehension> | <Dim> |
                <Alias>)*
               [<Minimizing> | <Maximizing>]

<Version> ::= <Integer> "." <Integer>

<Dim> ::= "dim" <IdentifierString> <SizeMatrixExpr> ":" ["alias"]
          ["const"] ["set" "of"] "int"

<Given> ::= "given" <IdentifierString> [<SizeMatrixExpr>]
            ":" ["set" "of"] "int" ["{" <UnboundedSetOfRanges> "}" |
                                    <AtomicId>]

<Letting> ::= "letting" <RangeVariableId> "=" (<ArithExpr> | <SetExpr>)

<Find> ::= "find" <RangeVariableId>
           ":" ["set" "of"] "int" <BoundedConstantSet>

<Alias> ::= "alias" <IdentifierString> [<SizeMatrixExpr>] "="
            <AliasExpr>
<AliasExpr> ::= "flatten" "(" <AliasExpr> ")" |
                <Function> "(" <AliasExpr> "," <ArithExpr> ")" |
                <IdentifierString> <SizeMatrixExpr>

<Minimizing> ::= ("minimizing" | "minimising") <AtomicId>
<Maximizing> ::= ("maximizing" | "maximising") <AtomicId>

<BoundedConstantSet> ::= ("{" <BoundedSetOfRanges> "}" |
```

```
                              <AtomicId>)

<LetFindAliasComprehension> ::= "[" (<Letting> | <Find> | <Alias>)
                                  "|" <CompParamsAndConditions> "]"

<CtComprehension> ::= "[" <ConstraintExpression> "|"
                      <CompParamsAndConditions> "]"
<SetComprehension> ::= "{" <ArithExpr> "|" <CompParamsAndConditions> "}"

<CompParamsAndConditions> ::= <RangeList> ["," <ConditionList>] |
                               <ConditionList>

<RangeList> ::= <ParameterRange> ["," <RangeList>]
<ParameterRange> ::= <IdentifierString> "in" <BoundedConstantSet>
<ConditionList> ::= <ComprehensionCondition> ["," <ConditionList>]
<ComprehensionCondition> ::= <ArithExpr>
                             ("!=" | "<" | ">" | "<=" | ">=" | "==")
                             <ArithExpr>

<Constraints> ::= "such that" (<Constraint>)*

<Constraint> ::= <IdentifierString> (<CtComprehension> |
                                     <ConstraintExpression>)
<ConstraintExpression> ::= <IdentifierString> "("
                           [<ConstraintArg>["," <ConstraintArg>]] ")"

<ConstraintArg> ::= <IdList> | <ConstraintList>

<IdList> ::= (<Id> | "[" [<Id> ("," <Id>)*] "]")
<ConstraintList> ::= (<Constraint> | "[" [<Constraint>
                      ("," <Constraint>)*] "]")

<ArithExpr> ::= "(" <ArithExpr> ")" |
                <Integer> | <AtomicId> |
                <ArithExpr> ("*" | "+" | "-" | "/") <ArithExpr> |
                "-" <ArithExpr>

<Integer> ::= "0" | <DigitNonZero> (<Integer>)*
<DigitNonZero> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<SetExpr> ::= "{" <BoundedSetOfRanges> "}" | <SetComprehension>

<UnboundedSetOfRanges> ::= <OpenRange> ("," <OpenRange>)*
```

```
<MatrixIndexRanges> ::= <OpenRange> ("," <OpenRange>)*
<BoundedSetOfRanges> ::= <ClosedRange> ("," <ClosedRange>)*

<OpenRange>    ::= <ClosedRange> |  ".." | <ArithExpr> ".." |
                   ".." <ArithExpr>
<ClosedRange>  ::= <ArithExpr> [".." <ArithExpr>]

<Id> ::= <AtomicId> | <MatrixExpr> | <ArithExpr>
<AtomicId> ::= <IdentifierString> [<DerefMatrixExpr>]
<RangeVariableId> ::= <IdentifierString> [ "[" <MatrixIndexRanges> "]" ]

<IdentifierString> ::= ( "a".."z" | "A".."Z" ) <AlNum>
<AlNum> ::= [("a".."z" | "A".."Z" | "0".."9" | "_") <AlNum>]

<MatrixExpr> ::= "flatten" "(" <MatrixExpr> ")" |
                 <Function> "(" <MatrixExpr> "," <ArithExpr> ")" |
                 <IdentifierString> "[" <MatrixIndexRanges> "]"

<Function> ::= "mult" | "add"

<SizeMatrixExpr> ::= "[" <ArithExpr> ("," <ArithExpr>)* "]"

<DerefMatrixExpr> ::= "[" <ArithExpr> ("," <ArithExpr>)* "]"

<Comment> ::= "$" (<Anything>)* <Newline>
<Newline> ::= "\n" | "\r"
```

D

BNF of Grasp

```
<architecture_statement> ::= <annotation>* "architecture" <IDENTIFIER>
    "{" (<template_statement> | <system_statement>)* "}"

<template_statement> ::= <annotation>* "template" <IDENTIFIER>
    "(" <parameter_list>? ")" "{" <statement>* "}"

<system_statement> ::= <annotation>* "system" <IDENTIFIER>
    "{" <statement>* "}"

<component_statement> ::= <annotation>* "component" <IDENTIFIER>
    "=" <IDENTIFIER> "(" <argument_list>? ")" ";"

<provides_statement> ::= <annotation>* "provides" <IDENTIFIER>
    ("{" <property_statement>* "}" | ";")

<requires_statement> ::= <annotation>* "requires" <IDENTIFIER>
    <variable_list>? ("{" <property_statement>* "}" | ";")

<check_statement> ::= <annotation>* "check" <expression> ";"

<property_statement> ::= <annotation>* "property" <IDENTIFIER>
    ("=" <expression>)? ";"

<link_statement> ::= <annotation>* "link" <member_expression> "to"
    <member_expression> ("{" <check_statement>* "}" | ";")

<annotation> ::= "@" <IDENTIFIER>? "(" <annotation_node>
    ("," <annotation_node>)* ")"
<annotation_node> ::= <IDENTIFIER> "=" <expression>

<expression> ::= <subsetof_expression>
<subsetof_expression> ::= <logicalOr_expression> SUBSETOF
```

```
    <logicalOr_expression>
<logicalOr_expression> ::= <logicalAnd_expression> DIS
    <logicalAnd_expression>
<logicalAnd_expression> ::= <bitwiseOr_expression> CON
    <bitwiseOr_expression>
<equality_expression> ::= <relational_expression> (EQL | NEQ)
    <relational_expression>
<relational_expression> ::= <acceptance_expression>
    (GTN | GTE | LTN | LTE) <acceptance_expression>
<acceptance_expression> ::= <augmentation_expression> ACCEPTS
    <augmentation_expression>
<augmentation_expression> ::= <additive_expression> (AUG | NAG)
    <additive_expression>
<additive_expression> ::= <multiplicative_expression> (ADD | SUB)
    <multiplicative_expression>
<multiplicative_expression > ::= <unary_expression> (MUL | DIV | MOD)
    <unary_expression>
<unary_expression> ::= NOT <unary_expression> | <primary_expression>
<primary_expression> ::= "(" <expression> ")" | member_expression
| <literal>
<member_expression> ::= <member_part> DOT <member_part>)*
<member_part> ::= <IDENTIFIER> "(" <argument_list>? ")" | <IDENTIFIER>
<literal> ::= <INTEGER_LITERAL> | <REAL_LITERAL> | <BOOLEAN_LITERAL>
| <STRING_LITERAL> | <set_literal>
<set_literal> ::= "[" <set_element_list>? "]"
<set_element_list> ::= <set_element> ("," <set_element>)*
<set_element> ::= <literal> | "(" <set_pair> ")" | <IDENTIFIER>
<set_pair> ::= <IDENTIFIER> "," <literal>

<parameter_list> ::= <IDENTIFIER> ("," <IDENTIFIER>)*

<variable_list> ::= <IDENTIFIER> ("," <IDENTIFIER>)*

<argument_list> ::= <expression> ("," <expression>)*

<statement> ::= <component_statement> | <requires_statement>
| <provides_statement> | <link_statement> | <check_statement>
| <property_statement>

<INTEGER_LITERAL> ::= "-"? <DECIMAL_DIGIT>+

<REAL_LITERAL> ::= "-"? <DECIMAL_DIGIT>* "." <DECIMAL_DIGIT>+
```

```
<BOOLEAN_LITERAL> ::= "true" | "false"

<STRING_LITERAL> ::= "\"" <ALPHANUMERIC_CHAR>* "\""

<IDENTIFIER> ::= (<ALPHA_CHAR> | "_") (<ALPHANUMERIC_CHAR> | "_")*

<ALPHANUMERIC_CHAR> ::= : <ALPHA_CHAR> | <DECIMAL_DIGIT>
<ALPHA_CHAR> ::= "a".."z" | "A".."Z"
<DECIMAL_DIGIT> ::= "0".."9"
```

E

Problem classes used in experiments

E.1. Learning when to use lazy learning in constraint solving

- fullins/insertions
- abb313GPIA
- aim
- ash331GPIA
- bf
- Black Hole
- bqwh
- composed
- Costas Array
- cril
- crossword
- driverlog
- dsjc
- dubois
- ehi
- frb
- geo
- geom

- hanoi
- hole
- jnh
- langford
- lard
- large
- Latin Square
- le
- lemma
- Magic Square
- mug
- myciel
- Non-monochromatic Rectangles
- ortholatin
- Peg Solitaire
- Pigeon Hole Problem
- pret
- qcp
- n-Queens
- qwh
- random
- renault
- Golomb Ruler
- series
- Social Golfers
- ssa
- tsp
- will199GPIA

E.2. Case study for the `alldifferent` constraint

- adt
- Black Hole
- bqwh
- contrived
- Costas Array
- Golomb Ruler
- langford
- Latin Square
- Magic Square
- ortholatin
- Pigeon Hole Problem
- Quasigroup existence
- n-Queens
- qwh
- Social Golfers
- Sports Scheduling

Bibliography

Belarmino Adenso-Diaz and Manuel Laguna. Fine-Tuning of Algorithms Using Fractional Experimental Designs and Local Search. *Oper. Res.*, 54(1):99–114, 2006. Cited on page 34.

Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis A. Tsahageas and Dominique H. de Villeneuve. *ECLiPSe User Manual*. July 2011. URL `http://www.eclipse-clp.org/`. Cited on page 13.

David W. Aha. Generalizing from Case Studies: A Case Study. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 1–10, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. Cited on pages 25, 30, and 112.

John A. Allen and Steven Minton. Selecting the Right Heuristic Algorithm: Runtime Performance Predictors. In *The Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 41–53. Springer-Verlag, 1996. Cited on pages 30, 32, 38, 40, 41, 42, and 113.

Carlos Ansótegui, Meinolf Sellmann and Kevin Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *CP*, pages 142–157, 2009. Cited on page 34.

Alejandro Arbelaez, Youssef Hamadi and Michele Sebag. Online Heuristic Selection in Constraint Programming. In *Symposium on Combinatorial Search*, 2009. Cited on pages 35, 37, 39, and 122.

Warren Armstrong, Peter Christen, Eric McCreath and Alistair P. Rendell. Dynamic Algorithm Selection Using Reinforcement Learning. In *International Workshop on Integrating AI and Data Mining*, pages 18–25, December 2006. Cited on pages 37, 39, 42, and 120.

Dharini Balasubramaniam and Lakshitha de Silva. Grasp Language Reference Manual Version 1.0. Technical Report, University of St Andrews, March 2011. URL `http://www.cs.st-andrews.ac.uk/ dharini/reports/ GraspManual.pdf`. Cited on page 131.

Eric Bauer and Ron Kohavi. An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants. *Machine Learning*, 36(1-2):105–139, 1999. Cited on page 75.

J. Christopher Beck and Mark S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1): 31–81, 2000. Cited on pages 42, 44, and 115.

J. Christopher Beck and Eugene C. Freuder. Simple Rules for Low-Knowledge Algorithm Selection. In *CPAIOR*, pages 50–64. Springer, 2004. Cited on pages 2, 42, 43, 44, and 118.

Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes and David Keyes. Application of Machine Learning in Selecting Sparse Linear Solvers. Technical Report, Columbia University, 2006. Cited on pages 39, 41, 44, and 120.

Sanjukta Bhowmick, Brice Toth and Padma Raghavan. Towards Low-Cost, High-Accuracy Classifiers for Linear Solver Selection. In *Proceedings of the 9th International Conference on Computational Science*, ICCS '09, pages 463–472, Berlin, Heidelberg, 2009. Springer-Verlag. Cited on pages 43 and 123.

Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh. *Handbook of Satisfiability*, Volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. Cited on page 31.

Mauro Birattari, Thomas Stützle, Luis Paquete and Klaus Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann, 2002. Cited on page 34.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2nd Edition, 2007. Cited on page 31.

James E. Borrett and Edward P. K. Tsang. A Context for Constraint Satisfaction Problem Formulation Selection. *Constraints*, 6(4):299–327, October 2001. Cited on pages 42 and 116.

James E. Borrett, Edward P. K. Tsang and Natasha R. Walsh. Adaptive Constraint Satisfaction: The Quickest First Principle. In *ECAI*, pages 160–164, 1996. Cited on pages 28, 32, 35, 37, 38, 41, 43, and 113.

Pavel Brazdil and Carlos Soares. A Comparison of Ranking Methods for Classification Algorithm Selection. In *Proceedings of the 11th European Conference on Machine Learning*, ECML '00, pages 63–74, London, UK, 2000. Springer-Verlag. Cited on pages 40, 41, and 115.

Eric A. Brewer. High-Level Optimization via Automated Statistical Modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 80–91, New York, NY, USA, 1995. ACM. Cited on pages 42, 44, 113, and 127.

Carla E. Brodley. Addressing the Selective Superiority Problem: Automatic Algorithm/Model Class Selection. In *ICML*, pages 17–24, 1993. Cited on pages 30, 35, 39, and 112.

Eamonn Cahill. Knowledge-based algorithm construction for real-world engineering PDEs. *Mathematics and Computers in Simulation*, 36(4-6):389–400, 1994. Cited on pages 113 and 127.

Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton and Manuela Veloso. PRODIGY: An Integrated Architecture for Planning and Learning. *SIGART Bull.*, 2:51–55, July 1991. Cited on pages 35, 37, 39, 42, 44, and 112.

Tom Carchrae and J. Christopher Beck. Low-Knowledge Algorithm Control. In *AAAI*, pages 49–54, 2004. Cited on pages 37, 39, 41, 43, and 118.

Tom Carchrae and J. Christopher Beck. Applying Machine Learning to Low-Knowledge Control of Optimization Algorithms. *Computational Intelligence*, 21 (4):372–387, 2005. Cited on pages 43 and 118.

Yves Caseau, François Laburthe and Glenn Silverstein. A Meta-Heuristic Factory for Vehicle Routing Problems. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 144–158, London, UK, 1999. Springer-Verlag. Cited on pages 38 and 114.

Feilong Chen and Rong Jin. Active Algorithm Selection. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 534–539. AAAI Press, 2007. Cited on page 74.

Vincent A. Cicirello and Stephen F. Smith. The Max K-Armed Bandit: A New Model of Exploration Applied to Search Heuristic Selection. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 1355–1361. AAAI Press, 2005. Cited on pages 36, 40, 42, 44, and 119.

Diane J. Cook and R. Craig Varnell. Maximizing the Benefits of Parallel Search Using Machine Learning. In *Proceedings of the 14th National Conference on Artificial Intelligence AAAI-97*, pages 559–564. AAAI Press, 1997. Cited on pages 27, 35, 39, 40, 41, 42, 44, and 114.

Peter Cowling, Graham Kendall and Eric Soubeiga. A Parameter-Free Hyperheuristic for Scheduling a Sales Summit. In *Proceedings of the 4th Metaheuristic International Conference, MIC 2001*, pages 127–131, 2001. Cited on pages 30 and 115.

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. Cited on page 34.

Romuald Debruyne and Christian Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of IJCAI*, pages 412–417, 1997. Cited on page 50.

Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 1st Edition, 2003. Cited on pages 11, 31, and 50.

James Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Richard Vuduc, R. Clint Whaley and Katherine Yelick. Self-Adapting Linear Algebra Algorithms and Software. *Proceedings of the IEEE*, 93(2):293–312, February 2005. Cited on pages 35, 39, 44, and 119.

Luca Di Gaspero and Andrea Schaerf. EasySyn++: A Tool for Automatic Synthesis of Stochastic Local Search Algorithms. In *Proceedings of the 2007 International Conference on Engineering Stochastic Local Search Algorithms: Designing, Implementing and Analyzing Effective Heuristics*, SLS'07, pages 177–181, Berlin, Heidelberg, 2007. Springer-Verlag. Cited on page 127.

Thomas G. Dietterich. Ensemble Methods in Machine Learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, Volume 1857 of *Lecture Notes In Computer Science*, pages 1–15. Springer-Verlag, 2000. Cited on pages 68 and 74.

Mehmet Dincbas, Pascal van Hentenryck, Helmut Simonis, Abderrahamane Aggoun and Alexander Herold. The CHIP system: Constraint handling in Prolog. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, Volume 310 of *Lecture Notes in Computer Science*, pages 774–775. Springer Berlin / Heidelberg, 1988. Cited on page 13.

Carmel Domshlak, Erez Karpas and Shaul Markovitch. To Max or Not to Max: Online Learning for Speeding Up Optimal Planning. In *AAAI*, 2010. Cited on pages 39, 43, and 123.

Wayne R. Dyksen and Carl R. Gritter. Ellipic Expert: An Expert System for Elliptic Partial Differential Equations. *Mathematics and Computers in Simulation*, 31(4-5):333–342, 1989. Special Double Issue. Cited on page 31.

Samir A. Mohamed Elsayed and Laurent Michel. Synthesis of Search Algorithms from High-level CP Models. In *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation*, September 2010. Cited on page 127.

Susan L. Epstein and Eugene C. Freuder. Collaborative Learning for Constraint Solving. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 46–60, London, UK, 2001. Springer-Verlag. Cited on pages 33, 42, and 116.

Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov and Bruce Samuels. The Adaptive Constraint Engine. In *Principles and Practice of Constraint Programming*, pages 525–540. Springer, 2002. Cited on pages 33, 39, 44, and 116.

Stephen Fenner, William Gasarch, Charles Glover and Semmy Purewal. Rectangle Free Coloring of Grids. Technical Report 1005.3750v1, arXiv, May 2010. URL http://arxiv.org/abs/1005.3750v1. Cited on page 134.

Eugene Fink. Statistical Selection Among Problem-Solving Methods. Technical Report CMU-CS-97-101, Carnegie Mellon University, 1997. Cited on page 114.

Eugene Fink. How to Solve It Automatically: Selection Among Problem-Solving Methods. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 128–136. AAAI Press, 1998. Cited on pages 35, 40, 42, and 114.

Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. Cited on page 62.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT '95: Proceedings of the Second European Conference on Computational Learning Theory*, pages 23–37, London, UK, 1995. Springer-Verlag. Cited on pages 68 and 74.

Alan M. Frisch, Warwick Harvey, Christopher A. Jefferson, Bernadette Martínez-Hernández and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. Cited on page 129.

Alex S. Fukunaga. Automated Discovery of Composite SAT Variable-Selection Heuristics. In *Eighteenth National Conference on Artificial Intelligence*, pages 641–648, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. Cited on pages 33, 34, and 117.

Alex S. Fukunaga. Automated Discovery of Local Search Heuristics for Satisfiability Testing. *Evol. Comput.*, 16:31–61, 2008. Cited on pages 33, 38, and 117.

Grigori Fursin, Yuriy Kashnikov, Abdul Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams and Michael O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39 (3):296–327, 2011. Cited on page 2.

Matteo Gagliolo and Jürgen Schmidhuber. A Neural Network Model for Inter-Problem Adaptive Online Time Allocation. In *Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005, 15th International Conference*, pages 7–12. Springer, 2005. Cited on pages 37, 39, 44, and 119.

Matteo Gagliolo and Jürgen Schmidhuber. Impact of Censored Sampling on the Performance of Restart Strategies. In *CP*, pages 167–181, 2006a. Cited on page 38.

Matteo Gagliolo and Jürgen Schmidhuber. Learning Dynamic Algorithm Portfolios. *Ann. Math. Artif. Intell.*, 47(3-4):295–328, 2006b. Cited on pages 37, 40, 41, 44, and 120.

Matteo Gagliolo, Viktor Zhumatiy and Jürgen Schmidhuber. Adaptive Online Time Allocation to Search Algorithms. In *ECML*, pages 134–143. Springer, 2004. Cited on pages 34, 37, 41, 44, and 119.

Pablo Garrido and María Riff. DVRP: a hard dynamic combinatorial optimisation problem tackled by an evolutionary hyper-heuristic. *Journal of Heuristics*, 16: 795–834, 2010. Cited on pages 38 and 123.

Cormac Gebruers, Alessio Guerri, Brahim Hnich and Michela Milano. Making Choices Using Structure at the Instance Level within a Case Based Reasoning Framework. In *CPAIOR*, pages 380–386, 2004. Cited on pages 2, 39, 42, 44, and 118.

Cormac Gebruers, Brahim Hnich, Derek Bridge and Eugene Freuder. Using CBR to Select Solution Strategies in Constraint Programming. In *Proc. of ICCBR-05*, pages 222–236, 2005. Cited on pages 41, 42, and 119.

Ian P. Gent and Toby Walsh. CSPLib: a benchmark library for constraints. Technical Report 9, APES, 1999. Cited on pages 14 and 134.

Ian P. Gent, Christopher A. Jefferson and Ian Miguel. MINION: A Fast, Scalable, Constraint Solver. In *ECAI*, pages 98–102, Amsterdam, The Netherlands, The Netherlands, 2006a. IOS Press. Cited on pages 48 and 133.

Ian P. Gent, Christopher A. Jefferson and Ian Miguel. Watched Literals for Constraint Propagation in Minion. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, pages 182–197, 2006b. Cited on page 2.

Ian P. Gent, Ian Miguel and Peter Nightingale. Generalised Arc Consistency for the AllDifferent Constraint: An Empirical Survey. *Artif. Intell.*, 172(18):1973–2000, December 2008. Cited on pages 62 and 63.

Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel and Peter Nightingale. Specification of the Dominion Input Language Version 0.1. Technical Report, University of St Andrews, 2009. URL `http://www-circa.mcs.st-and.ac.uk/Preprints/InLangSpec.pdf`. Cited on page 129.

Ian P. Gent, Ian Miguel and Neil C. A. Moore. Lazy Explanations for Constraint Propagators. In *PADL'10*, pages 217–233, 2010. Cited on page 47.

Alfonso E. Gerevini, Alessandro Saetti and Mauro Vallati. An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*, pages 350–353, 2009. Cited on pages 36, 40, 42, 85, and 123.

Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. The MIT Press, 2007. Cited on page 85.

Mark Ghallab, Dana Nau and Paolo Traverso. *Automated Planning: Theory & Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2004. Cited on page 31.

Fred Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Comput. Oper. Res.*, 13(5):533–549, 1986. Cited on page 30.

Carla P. Gomes and Bart Selman. Algorithm Portfolio Design: Theory vs. Practice. In *UAI*, pages 190–197, 1997a. Cited on pages 32 and 114.

Carla P. Gomes and Bart Selman. Practical Aspects of Algorithm Portfolio Design. In *Proc. of Third ILOG International Users Meeting*, 1997b. Cited on pages 32 and 114.

Carla P. Gomes and Bart Selman. Algorithm Portfolios. *Artificial Intelligence*, 126 (1-2):43–62, 2001. Cited on pages 1, 2, 32, 36, and 115.

Jonathan Gratch and Gerald DeJong. COMPOSER: A Probabilistic Solution to the Utility Problem in Speed-Up Learning. In *AAAI*, pages 235–240, 1992. Cited on pages 39 and 112.

Alessio Guerri and Michela Milano. Learning Techniques for Automatic Algorithm Portfolio Selection. In *ECAI*, pages 475–479, 2004. Cited on pages 38, 39, 41, 42, 48, and 118.

Haipeng Guo. *Algorithm Selection for Sorting and Probabilistic Inference: A Machine Learning-Based Approach*. PhD thesis, Kansas State University, 2003. Cited on page 117.

Haipeng Guo and William H. Hsu. A Learning-Based Algorithm Selection Meta-reasoner for the Real-Time MPE Problem. In *Australian Conference on Artificial Intelligence*, pages 307–318, 2004. Cited on pages 27, 39, 40, 41, 43, 44, 48, and 119.

Shai Haim and Toby Walsh. Restart Strategy Selection Using Machine Learning Techniques. In *SAT '09: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 312–325, Berlin, Heidelberg, 2009. Springer-Verlag. Cited on pages 40, 63, and 122.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009. Cited on pages 53, 66, 75, and 82.

Tim Hinrich, Nathaniel Love, Charles Petrie, Lyle Ramshaw, Akhil Sahai and Sharad Singhal. Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation. In *DSOM*, 2004. Cited on page 132.

Robert C. Holte. Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Mach. Learn.*, 11:63–90, April 1993. Cited on page 89.

John E. Hopcroft and Richard M. Karp. An $n^{\frac{5}{2}}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. Cited on page 62.

Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry A. Kautz, Bart Selman and David M. Chickering. A Bayesian Approach to Tackling Hard Computational Problems. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 235–244, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. Cited on pages 33, 37, 41, 42, and 116.

Patricia D. Hough and Pamela J. Williams. Modern Machine Learning for Automatic Optimization Algorithm Selection. In *Proceedings of the INFORMS Artificial Intelligence and Data Mining Workshop*, November 2006. Cited on pages 32, 39, 41, 42, and 120.

Elias N. Houstis, John R. Rice, Nikos P. Chrisochoides, Haralambos C. Karathanasis, Panayiotis N. Papachiou, Meletis K. Samartzis, Emmanuel A. Vavalis, Ko Y. Wang and Sanjiva Weerawarana. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. *SIGARCH Comput. Archit. News*, 18 (3b):96–107, June 1990. Cited on page 31.

Adele E. Howe, Eric Dahlman, Christopher Hansen, Michael Scheetz and Anneliese von Mayrhauser. Exploiting Competitive Planner Performance. In *Proceedings of the Fifth European Conference on Planning*, pages 62–72. Springer, 1999. Cited on pages 32, 36, 40, 41, 42, 43, 44, and 114.

Bernardo A. Huberman, Rajan M. Lukose and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275(5296):51–54, 1997. Cited on page 32.

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. Cited on pages 30, 42, and 120.

Frank Hutter, Holger H. Hoos and Thomas Stützle. Automatic Algorithm Configuration based on Local Search. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 1152–1157. AAAI Press, 2007. Cited on pages 33 and 34.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown and Kevin P. Murphy. An Experimental Investigation of Model-Based Parameter Optimisation: SPO and

Beyond. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 271–278, New York, NY, USA, 2009a. ACM. Cited on page 34.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Int. Res.*, 36(1): 267–306, 2009b. Cited on pages 33, 34, and 63.

IBM. *ILOG Solver User's Manual*. 2011. Cited on page 13.

Christopher A. Jefferson, Lars Kotthoff, Neil C.A. Moore, Peter Nightingale, Karen E. Petrie and Andrea Rendl. *The Minion Manual*. 2011. URL `http://minion.sourceforge.net/`. Cited on page 13.

Thorsten Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 217–226, New York, NY, USA, 2006. ACM. Cited on page 85.

Anupam Joshi, Sanjiva Weerawarana, Narendran Ramakrishnan, Elias N. Houstis and John R. Rice. Neuro-Fuzzy Support for Problem-Solving Environments: A Step Toward Automated Solution of PDEs. *IEEE Comput. Sci. Eng.*, 3(1):44–56, March 1996. Cited on pages 31 and 113.

Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann and Kevin Tierney. ISAC – Instance-Specific Algorithm Configuration. In *ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 751–756. IOS Press, 2010. Cited on pages 33, 38, 39, 107, and 124.

Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz and Meinolf Sellmann. Algorithm Selection and Scheduling. In *17th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 454–469, 2011. Cited on pages 36 and 124.

Mohamed S. Kamel, Wayne H. Enright and K. S. Ma. ODEXPERT: An Expert System to Select Numerical Solvers for Initial Value ODE Systems. *ACM Trans. Math. Softw.*, 19(1):44–62, March 1993. Cited on pages 31, 44, and 112.

George Katsirelos. *Nogood processing in CSPs*. PhD thesis, University of Toronto, January 2009. Cited on page 47.

George Katsirelos and Fahiem Bacchus. Unrestricted Nogood Recording in CSP search. In *CP*, pages 873–877. Springer, 2003. Cited on page 47.

George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In *AAAI*, pages 390–396. AAAI Press, 2005. Cited on page 47.

Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, pages 1137–1143. Morgan Kaufmann, 1995. Cited on pages 56, 68, 75, and 87.

Christian Kroer and Yuri Malitsky. Feature Filtering for Instance-Specific Algorithm Configuration. In *Proceedings of the 23rd International Conference on Tools with Artificial Intelligence*, 2011. Cited on pages 43 and 124.

Erik Kuefler and Tzu-Yi Chen. On Using Reinforcement Learning to Solve Sparse Linear Systems. In *Proceedings of the 8th International Conference on Computational Science*, ICCS '08, pages 955–964, Berlin, Heidelberg, 2008. Springer-Verlag. Cited on pages 44 and 122.

Michail G. Lagoudakis and Michael L. Littman. Algorithm Selection using Reinforcement Learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. Cited on pages 35, 37, 39, 41, and 115.

Michail G. Lagoudakis and Michael L. Littman. Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. In *LICS/SAT*, pages 344–359, 2001. Cited on pages 35, 44, and 116.

Pat Langley. Learning Effective Search Heuristics. In *IJCAI*, pages 419–421, 1983a. Cited on pages 33, 35, 37, 39, and 112.

Pat Langley. Learning Search Strategies through Discrimination. *International Journal of Man-Machine Studies*, pages 513–541, 1983b. Cited on pages 33, 42, 44, and 112.

Niklas Lavesson and Paul Davidsson. Quantifying the Impact of Learning Algorithm Parameter Tuning. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 395–400. AAAI Press, 2006. Cited on page 74.

Christophe Lecoutre, Olivier Roussel and Marc R. C. van Dongen. Third International CSP Solver Competition. Technical Report, 2008. Cited on page 17.

Rui Leite, Pavel Brazdil, Joaquin Vanschoren and Francisco Queiros. Using Active Testing and Meta-Level Information for Selection of Classification Algorithms. In *3rd PlanLearn Workshop*, August 2010. Cited on pages 40, 41, 42, 44, and 123.

Kevin Leyton-Brown, Eugene Nudelman and Yoav Shoham. Learning the Empirical Hardness of Optimization Problems: The Case of Combinatorial Auctions. In *CP '02: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, pages 556–572, London, UK, 2002. Springer-Verlag. Cited on pages 30, 40, 43, 44, and 117.

Kevin Leyton-Brown, Eugene Nudelman and Yoav Shoham. Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions. *J. ACM*, 56: 22:1–22:52, July 2009. Cited on page 30.

James Little, Cormac Gebruers, Derek Bridge and Eugene Freuder. Capturing Constraint Programming Experience: A Case-Based Approach. In *Modref*, 2002. Cited on pages 42 and 117.

Lionel Lobjois and Michel Lemaître. Branch and Bound Algorithm Selection by Performance Prediction. In *AAAI '98/IAAI '98: Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 353–358, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. Cited on pages 30, 32, 40, 41, 42, 44, and 114.

Daniel Mailharro. A classification and constraint-based framework for configuration. *Artif. Intell. Eng. Des. Anal. Manuf.*, 12:383–397, September 1998. Cited on page 131.

Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz and Meinolf Sellmann. Non-Model-Based Algorithm Portfolios for SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 369–370, 2011. Cited on page 124.

Oded Maron and Andrew Moore. The Racing Algorithm: Model Selection for Lazy Learners. In *Artificial Intelligence Review*, Volume 11, pages 193–225, 1997. Cited on page 34.

Brendan D. McKay. Practical Graph Isomorphism. In *10th Manitoba Conf., Congressus Numerantium 30*, pages 45–87, 1981. Cited on page 51.

Sergey D. Mechveliani. Computer algebra with Haskell: applying functional-categorial-lazy programming. In *Proceedings of International Workshop CAAP*, Dubna, Russia, 2001. Cited on page 133.

Steven Minton. An Analytic Learning System for Specializing Heuristics. In *IJCAI'93: Proceedings of the 13th International Joint Conference on Artifical Intelligence*, pages 922–928, San Francisco, CA, USA, 1993a. Morgan Kaufmann Publishers Inc. Cited on pages 33 and 113.

Steven Minton. Integrating Heuristics for Constraint Satisfaction Problems: A Case Study. In *AAAI: Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 120–126, 1993b. Cited on pages 33 and 113.

Steven Minton. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, 1:7–43, 1996. Cited on pages 33, 34, 35, 37, 38, 42, 44, 63, 113, and 126.

Sanjay Mittal and Brian Falkenhainer. Dynamic Constraint Satisfaction Problems. In *AAAI*, pages 25–32, 1990. Cited on page 131.

Jean-Noël Monette, Yves Deville and Pascal van Hentenryck. Aeon: Synthesizing Scheduling Algorithms from High-Level Models. In *Operations Research and Cyber-Infrastructure*, pages 43–59. 2009. Cited on page 127.

Neil C.A. Moore. *Improving the efficiency of learning CSP solvers.* PhD thesis, University of St Andrews, 2011. Cited on page 48.

Alexander Nareyek. Choosing Search Heuristics by Non-Stationary Reinforcement Learning. In *Metaheuristics: Computer Decision-Making*, pages 523–544. Kluwer Academic Publishers, 2001. Cited on pages 39, 41, 43, and 116.

David M. Neves. Learning procedures from examples and by doing. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 624–630, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc. Cited on page 32.

Mladen Nikolić, Filip Marić and Predrag Janičić. Instance-Based Selection of Policies for SAT Solvers. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 326–340, Berlin, Heidelberg, 2009. Springer-Verlag. Cited on pages 32, 35, 39, 41, and 122.

Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar and Yoav Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, Volume 3258 of *Lecture Notes in Computer Science*, pages 438–452. Springer Berlin / Heidelberg, 2004. Cited on pages 32, 35, and 118.

Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent and Barry O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, 2008. Cited on pages 2, 32, 36, 37, 38, 39, 41, 42, 44, 85, and 122.

Karen E. Petrie and Barbara M. Smith. Symmetry Breaking in Graceful Graphs. In *CP*, Volume 2833 of *LNCS*, pages 930–934. Springer, 2003. Cited on page 134.

Marek Petrik. Statistically Optimal Combination of Algorithms. In *Local Proceedings of SOFSEM 2005*, 2005. Cited on pages 36, 39, 41, and 119.

Marek Petrik and Shlomo Zilberstein. Learning Parallel Portfolios of Algorithms. *Annals of Mathematics and Artificial Intelligence*, 48(1-2):85–106, 2006. Cited on page 36.

Sanja Petrovic and Rong Qu. Case-Based Reasoning as a Heuristic Selector in Hyper-Heuristic for Course Timetabling Problems. In *KES*, pages 336–340, 2002. Cited on pages 43 and 117.

Mike Preuss and Thomas Bartz-Beielstein. Sequential Parameter Optimization Applied to Self-Adaptation for Binary-Coded Evolutionary Algorithms. In Fernando Lobo, Claudio Lima and Zbigniew Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, Studies in Computational Intelligence, pages 91–120. Springer, 2007. Cited on page 34.

Jean-François Puget. Constraint Programming Next Challenge: Simplicity of Use. In *CP*, pages 5–8, 2004. Cited on page 14.

Luca Pulina and Armando Tacchella. A multi-engine solver for quantified Boolean formulas. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, pages 574–589, Berlin, Heidelberg, 2007. Springer-Verlag. Cited on pages 32, 38, 39, 41, 42, 43, and 121.

Luca Pulina and Armando Tacchella. A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints*, 14(1):80–116, 2009. Cited on pages 32, 36, 37, 44, 83, 85, 90, and 121.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1st Edition, January 1993. Cited on pages 53 and 76.

R. Bharat Rao, Diana Gordon and William Spears. For Every Generalization Action, Is There Really An Equal And Opposite Reaction? Analysis of the Conservation Law for Generalization Performance. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 471–479. Morgan Kaufmann, 1995. Cited on page 98.

Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Volume 1 of *AAAI '94*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. Cited on page 62.

John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976. Cited on pages xv, 1, 2, 25, 26, 27, 28, and 44.

Christopher K. Riesbeck and Roger C. Schank. *Inside Case-Based Reasoning*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1989. Cited on page 82.

Mark Roberts and Adele E. Howe. Directing a Portfolio with Learning. In *AAAI 2006 Workshop on Learning for Search*, 2006. Cited on pages 36, 39, 41, and 120.

Mark Roberts and Adele E. Howe. Learned Models of Performance for Many Planners. In *ICAPS 2007 Workshop AI Planning and Learning*, 2007. Cited on pages 36, 40, 41, 42, and 121.

Mark Roberts, Adele E. Howe, Brandon Wilson and Marie desJardins. What Makes Planners Predictable? In *ICAPS*, pages 288–295, 2008. Cited on pages 41, 43, and 121.

Francesca Rossi, Peter van Beek and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006. Cited on page 31.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd Edition, 2009. Cited on page 31.

Daniel Sabin and Eugene C. Freuder. Configuration as Composite Constraint Satisfaction. *Proceedings of the 1st Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, January 1996. Cited on page 131.

Hani El Sakkout, Mark G. Wallace and E. Barry Richards. An Instance of Adaptive Constraint Propagation. In *Proc. of CP96*, pages 164–178. Springer Verlag, 1996. Cited on pages 36, 37, 38, 41, 43, and 113.

Horst Samulowitz and Roland Memisevic. Learning to Solve QBF. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 255–260. AAAI Press, 2007. Cited on pages 32, 35, 37, 39, and 121.

Christian Schulte. *Programming Constraint Services*, Volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002. Cited on page 22.

Christian Schulte and Guido Tack. Perfect Derived Propagators. In *CP*, Volume 5202 of *LNCS*, pages 571–575. Springer, 2008. Cited on page 127.

Christian Schulte, Guido Tack and Mikael Lagerkvist. *Modeling and Programming with Gecode*. 2011. URL `http://www.gecode.org/`. Cited on page 13.

Jonathan Sillito. Improvements to and Estimating the Cost of Solving Constraint Satisfaction Problems. Master's thesis, University of Alberta, 2000. Cited on pages 40 and 115.

Bryan Silverthorn and Risto Miikkulainen. Latent Class Models for Algorithm Portfolio Methods. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. Cited on pages 2, 40, 41, 42, 44, and 123.

Douglas R. Smith. KIDS - A Knowledge-Based Software Development System. In *Automating Software Design*, pages 483–514. MIT Press, 1990. Cited on page 127.

Tobiah E. Smith and Dorothy E. Setliff. Knowledge-Based Constraint-Driven Software Synthesis. In *Knowledge-Based Software Engineering Conference*, pages 18–27, September 1992. Cited on pages 1, 33, 37, 42, 44, 112, and 127.

Kate A. Smith-Miles. Cross-Disciplinary Perspectives on Meta-Learning for Algorithm Selection. *ACM Comput. Surv.*, 41:6:1–6:25, January 2009. Cited on page 46.

Carlos Soares, Pavel B. Brazdil and Petr Kuba. A Meta-Learning Method to Select the Kernel Width in Support Vector Regression. *Mach. Learn.*, 54(3):195–209, March 2004. Cited on pages 1, 41, 42, 44, and 118.

Biplav Srivastava and Subbarao Kambhampati. Synthesizing Customized Planners from Specifications. *J. Artif. Int. Res.*, 8(1):93–128, March 1998. Cited on page 127.

Efstathios Stamatatos and Kostas Stergiou. Learning How to Propagate Using Random Probing. In *CPAIOR '09: Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 263–278, Berlin, Heidelberg, 2009. Springer-Verlag. Cited on pages 39, 42, and 122.

Kostas Stergiou. Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems. *AI Commun.*, 22(3):125–141, 2009. Cited on pages 36, 37, 39, 43, and 122.

David H. Stern, Horst Samulowitz, Ralf Herbrich, Thore Graepel, Luca Pulina and Armando Tacchella. Collaborative Expert Portfolio Management. In *AAAI*, pages 179–184, 2010. Cited on pages 39, 44, and 123.

Matthew J. Streeter and Stephen F. Smith. New Techniques for Algorithm Portfolio Design. In *UAI*, pages 519–527, 2008. Cited on pages 36 and 121.

Matthew J. Streeter, Daniel Golovin and Stephen F. Smith. Combining Multiple Heuristics Online. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 1197–1203. AAAI Press, 2007. Cited on pages 36, 39, 42, and 121.

Markus Stumptner, Gerhard E. Friedrich and Alois Haselböck. Generative constraint-based configuration of large technical systems. *Artif. Intell. Eng. Des. Anal. Manuf.*, 12:307–320, September 1998. Cited on page 131.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2011. URL `http://www.R-project.org/`. ISBN 3-900051-07-0. Cited on pages 66 and 75.

Hugo Terashima-Marín, Peter Ross and Manuel Valenzuela-Rendón. Evolution of Constraint Satisfaction Strategies in Examination Timetabling. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO99)*, pages 635–642. Morgan Kaufmann, 1999. Cited on pages 34 and 114.

The Choco Team. *Choco: an Open Source Java Constraint Programming Library*. July 2011. URL `http://www.emn.fr/z-info/choco-solver/`. Cited on page 13.

David Tolpin and Solomon E. Shimony. Rational Deployment of CSP Heuristics. In *IJCAI*, pages 680–686, 2011. Cited on pages 38, 41, and 124.

Edward P. K. Tsang, James E. Borrett and Alvin C. M. Kwan. An Attempt to Map the Performance of a Range of Algorithm and Heuristic Combinations. In *Proc. of AISB'95*, pages 203–216. IOS Press, 1995. Cited on pages 28, 32, and 113.

Paul E. Utgoff. Perceptron Trees: A Case Study In Hybrid Concept Representations. In *National Conference on Artificial Intelligence*, pages 601–606, 1988. Cited on page 30.

Pascal van Hentenryck and Laurent Michel. Synthesis of Constraint-Based Local Search Algorithms from High-Level Models. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 273–278. AAAI Press, 2007. Cited on page 127.

Willem-Jan van Hoeve. *The Alldifferent Constraint: A Survey.* 2001. Cited on page 62.

Virginia Vassilevska, Ryan Williams and Shan L. M. Woo. Confronting Hardness Using a Hybrid Approach. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '06, pages 1–10, New York, NY, USA, 2006. ACM. Cited on pages 30 and 37.

Dimitris Vrakas, Grigorios Tsoumakas, Nick Bassiliades and Ioannis Vlahavas. Learning Rules for Adaptive Planning. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 82–91, 2003. Cited on pages 27, 33, 39, 44, and 117.

Jean-Paul Watson. *Empirical modeling and analysis of local search algorithms for the job-shop scheduling problem.* PhD thesis, Colorado State University, Fort Collins, CO, USA, 2003. Cited on pages 40 and 117.

Duncan J. Watts and Steven H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, 1998. Cited on page 50.

Sanjiva Weerawarana, Elias N. Houstis, John R. Rice, Anupam Joshi and Catherine E. Houstis. PYTHIA: A Knowledge-Based System to Select Scientific Algorithms. *ACM Trans. Math. Softw.*, 22(4):447–468, 1996. Cited on pages 31, 40, 42, 44, and 113.

Wanxia Wei, Chu Min Li and Harry Zhang. Switching among Non-Weighting, Clause Weighting, and Variable Weighting in Local Search for SAT. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, pages 313–326, Berlin, Heidelberg, 2008. Springer-Verlag. Cited on page 122.

Stephen J. Westfold and Douglas R. Smith. Synthesis of Efficient Constraint Satisfaction Programs. *Knowl. Eng. Rev.*, 16(1):69–84, 2001. Cited on page 127.

David Wilson, David Leake and Randall Bramley. Case-Based Recommender Components for Scientific Problem-Solving Environments. In *Proc. of the 16th International Association for Mathematics and Computers in Simulation World Congress*, 2000. Cited on pages 39, 42, 43, 44, and 115.

Ian H. Witten, Eibe Frank and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 3rd Edition, 2011. Cited on pages 31, 52, 66, 67, and 74.

David H. Wolpert. The Supervised Learning No-Free-Lunch Theorems. In *Proc. 6th Online World Conference on Soft Computing in Industrial Applications*, pages 25–42, 2001. Cited on page 89.

Huayue Wu and Peter van Beek. On Portfolios for Backtracking Search in the Presence of Deadlines. In *ICTAI '07: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, pages 231–238, Washington, DC, USA, 2007. IEEE Computer Society. Cited on pages 33, 36, and 121.

Lin Xu, Holger H. Hoos and Kevin Leyton-Brown. Hierarchical Hardness Models for SAT. In *CP*, pages 696–711, 2007a. Cited on pages 30, 40, 63, and 120.

Lin Xu, Frank Hutter, Holger H. Hoos and Kevin Leyton-Brown. SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. In *CP*, pages 712–727, 2007b. Cited on pages 32, 35, and 118.

Lin Xu, Frank Hutter, Holger H. Hoos and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008. Cited on pages 2, 3, 27, 32, 35, 36, 38, 40, 41, 42, 43, 44, 84, 98, and 118.

Lin Xu, Frank Hutter, Holger H. Hoos and Kevin Leyton-Brown. SATzilla2009: An Automatic Algorithm Portfolio for SAT. In *2009 SAT Competition*, 2009. Cited on pages 41, 43, 44, and 85.

Lin Xu, Holger H. Hoos and Kevin Leyton-Brown. Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. In *Twenty-Fourth Conference of the Association for the Advancement of Artificial Intelligence (AAAI-10)*, pages 210–216, 2010. Cited on pages 33 and 107.