Exploiting Machine Learning for Combinatorial Problem Solving and Optimisation

Barry Hurley

NATIONAL UNIVERSITY OF IRELAND, CORK

College of Science, Engineering, and Food Science

Department of Computer Science

Insight Centre for Data Analytics

Thesis submitted for the degree of Doctor of Philosophy

December 2016

Supervisor: Professor Barry O'Sullivan

Head of Department/School: Professor Cormac Sreenan

Contents

List of Figures . . . iv
List of Tables . . . v
Abstract . . . vii
Declaration . . . viii
Acknowledgements . . . x

1 Introduction 1
1.1 Motivation . . . 1
1.2 Thesis Statement and Contributions . . . 2
1.3 Overview of the Dissertation . . . 5

2 Background and Related Work 7
2.1 Numberjack . . . 7
2.2 Modelling using Numberjack . . . 8
2.2.1 Variables . . . 9
2.2.2 Constraints . . . 10
2.2.3 Inference . . . 12
2.2.4 Global Constraints . . . 13
2.2.5 Optimisation . . . 17
2.3 Solving Technologies . . . 18
2.3.1 Constraint Programming . . . 18
2.3.2 Satisfiability . . . 19
2.3.3 Mixed Integer Programming . . . 20
2.3.4 Cost Function Networks . . . 21
2.3.5 Choice is good . . . 22
2.4 Systematic Search . . . 26
2.4.1 Search Heuristics in Constraint Programming . . . 28
2.4.2 Restarting and Randomness . . . 28
2.5 Encoding a CSP as SAT . . . 29
2.5.1 Direct Encoding . . . 30
2.5.2 Support Encoding . . . 32
2.5.3 Full Regular Encoding . . . 33
2.5.4 Regular Encoding . . . 35
2.5.5 Additional Encodings . . . 36
2.5.6 Encoding Reified Constraints . . . 36
2.6 Solver Portfolios . . . 37
2.6.1 Selection Models . . . 39
2.6.2 Non-Model Based Portfolios . . . 41
2.6.3 Disparity of Approaches . . . 41
2.7 Chapter Summary . . . 42

3 Case-Based Reasoning for Solver Selection 43
3.1 Building Case-bases for SAT Solving . . . 44
3.1.1 Feature Representation . . . 44
3.1.2 Case-Bases . . . 45

3.1.3 Similarity Metric 46
3.2 Adaptation Strategies 46
3.2.1 Kendal-Tau Distance 47
3.2.2 Kemeny Consensus Ranking 48
3.2.3 Borda Voting 49
3.2.4 Weighted Borda Voting 49
3.2.5 Copeland Voting 50
3.2.6 Bucklin Voting 50
3.2.7 Coomb's Voting 51
3.2.8 Instant Runoff Voting 51
3.2.9 Best Average Score 52
3.2.10 Very Best Ranking 52
3.2.11 Summary of Voting Methods 52
3.3 An Exact Method for Computing Optimal Kemeny Rankings . . . 53
3.4 Evaluation of a CBR-based Portfolio for SAT 55
3.4.1 Methodology 56
3.4.2 Evaluation of the Adaptation Schemes 56
3.4.3 A CBR-based Solver Portfolio for SAT 57
3.5 Chapter Summary 63

4 A Hierarchical Portfolio of Representations and Solvers 64
4.1 Introduction 64
4.2 Background and Related Work 65
4.2.1 Related Portfolios 66
4.2.2 Solving a CSP using SAT 66
4.2.3 Uniform Random Binary CSPs 67
4.3 Multiple Encodings and Solvers 68
4.3.1 Uniform Random Instances 68
4.3.2 CSP Competition Instances 70
4.4 Proteus: A Hierarchical Portfolio for CSPs 71
4.5 Experimental Evaluation 73
4.5.1 Setup 73
4.5.2 Portfolio and Solver Results 77
4.5.3 Greater than the Sum of its Parts 80
4.6 CSP versus SAT Analysis 81
4.6.1 Empirical Setup 81
4.6.2 Orders of Magnitude Differences 83
4.6.3 Explaining Portfolio Performance 84
4.7 Proteus for Graphical Model Optimisation 87
4.7.1 Graphical Models Background 88
4.7.2 Translations Between Formalisms 89
4.7.3 Graphical Model Benchmarks 91
4.7.4 Experimental Setup 92
4.7.5 Graphical Model Instance Features 92
4.7.6 Offline Evaluation Results 94
4.7.7 UAI 2014 Probabilistic Inference Competition 95

4.8 Chapter Summary 97

5 Runtime Distributions and Solver Selection 100

5.1 Introduction 100
5.2 Background 101
5.3 Runtime Variation in Empirical Comparisons 103
5.4 Expected Performance 105
5.5 SAT Competition 2014 106
5.6 State-of-the-art Runtime Prediction 110
5.6.1 Log Transformation 110
5.6.2 Experimental Setup 111
5.6.3 Prediction Fragility 112
5.7 Runtime Variation Analysis 114
5.8 Related Work 118
5.9 Conclusions and Discussion 119

6 Balancing Solution Time and Energy Consumption 121

6.1 Motivation 122
6.2 Solution Time, Number of Cores, Energy 123
6.2.1 Empirical Setup 123
6.2.2 Solution Time versus Number of Cores 124
6.2.3 Energy versus Number of Cores 125
6.2.4 Expected Solution Time versus Energy 126
6.2.5 Solution Time versus Energy Tradeoff 127
6.3 Predicting the Optimal Number of Cores 128
6.4 Chapter Summary 130

7 Conclusions and Future Work 131

7.1 Thesis Defence 131
7.2 Future Work 133
7.2.1 SAT Encodings 133
7.2.2 Multi-language 134
7.2.3 Runtime Distributions 135
7.2.4 Portfolios 135

List of Figures

2.1 Abstract process of modelling a problem. 9
2.2 Example of CP propagation on a Sudoku instance. 12
2.3 Example of propagation on a Hall set $\{1, 2, 5\}$. 14
2.4 Example task assignment on a cumulative resource. 17
2.5 Model of the Warehouse Location Problem in Numberjack. 23
2.6 Model of the Costas Array Problem in Numberjack. 25
2.7 Model of the Golomb Ruler Problem in Numberjack. 26
2.8 A partial example of the search tree generated by backtracking search. 27
2.9 Model of the algorithm selection problem. 38

3.1 Summary of SAT instances features. 44
3.2 Cumulative time of each solver in a ranking 57

4.1 `MiniSat` and `Clasp` on random binary CSPs. 69
4.2 Performance of the virtual best CSP portfolio and the virtual best SAT-based portfolio. 71
4.3 A contrasting example of a flattened versus hierarchical approach. 72
4.4 Overview of the machine learning models used in the hierarchical approach. 78
4.5 Box-plot of the orders of magnitude PAR10 difference between VB CSP and VB SAT solvers. 82
4.6 Example decision tree choosing SAT or CSP. 86
4.7 Cumulative solver ranking of UAI 2014 MAP solvers. 98

5.1 Log-log plots showing the empirical runtime distribution of the top three solvers on two instances from the application category at the 2014 SAT Competition. 104
5.2 A histogram of the number of instances solved across 100 runs on 300 instances from the application category of the SAT Competition 2014. 108
5.3 Histogram of runtime standard deviations. 114
5.4 Histogram of runtime standard deviations, broken out by satisfiability. 115
5.5 Boxplot comparison of the runtime variations between solvers on 300 SAT instances, grouped by instance category. 117

6.1 Illustration of solution time versus the number of cores for some sample instances. 124
6.2 Illustration of the total energy versus the number of cores in log-scale for some sample instances. 125
6.3 Illustration of solution time versus total energy in log-scale for some sample instances. 126
6.4 Illustration of trade off between solution time and the best energy achievable over all instances. 127

List of Tables

2.1 Example binary constraint definitions in Numberjack. 11
2.2 Performance comparison between a mixed integer programming solver (SCIP) and a constraint programming solver (Mistral) on some instances of the Warehouse Location Problem. 24
2.3 Performance of a constraint programming, satisfiability, and mixed integer programming solver on two arithmetic puzzles of increasing size. 27
2.4 An example CSP encoded to SAT under the direct encoding. . . . 32
2.5 An example CSP encoded to SAT under the support encoding. . . 33
2.6 An example CSP encoded to SAT under the full regular encoding. 35
2.7 An example CSP encoded to SAT under the regular encoding. . . 36

3.1 The number of problem instances and the number of solvers associated with each of our three case-bases. 46
3.2 An example list of label rankings, which is used for a running example. 47
3.3 Summary of aggregate rankings. 52
3.4 Table of paired t-tests comparing the area under the curve for various aggregation methods. 59
3.5 Leader board for the Handcrafted category of problem instances. 60
3.6 Leader board for the Industrial category of problem instances. . . 61
3.7 Leader board for the Random category of problem instances. . . 62

4.1 Performance of the learning algorithms for the hierarchical approach. 78
4.2 Ranking of each classification, regression, and clustering algorithm to choose the solving mechanism in a flattened setting. 79
4.3 Virtual best performances ranked by PAR10 score. 81
4.4 Stratified 10-fold cross validation predicting CSP or SAT by a decision tree classifier using CSP features. 85
4.5 Mean feature importance for stratified 10-fold cross validation predicting CSP or SAT by a decision tree classifier using CSP features. 85
4.6 Summary of translations between graphical model formalisms. . . 90
4.7 Number of instances and their total compressed (gzipped) size per format for each benchmark resource 92
4.8 Gini importances of graphical model features, truncated at 1%. . 93
4.9 Summary of portfolio approaches sorted by decreasing number of problems solved over the 2,564 instances. 94
4.10 Offline evaluation of the UAI 2014 portfolio on 2,564 instances . . 96

5.1 Number of instances solved out of 300 by the top-3 solvers in the application category of the 2014 SAT Competition. 107
5.2 Expected success rates over multiple runs on 300 application instances from the SAT Competition 2014. 108

5.3 Summary of the runtime variations of three solvers on a set of sample instances from the 2014 SAT Competition. 109
5.4 Samples of the runtime variations of MiniSat on a set of sample industrial category instances from the SAT Competitions, Races, and Challenges between 2002 and 2011. 112
5.5 Comparison of training and testing on different samples of each instance's observed runtimes. 113

6.1 Evaluation of various parallel policies. 129

Abstract

This dissertation presents a number of contributions to the field of solver portfolios, in particular for combinatorial search problems. We propose a novel hierarchical portfolio which does not rely on a single problem representation, but may transform the problem to an alternate representation using a portfolio of encodings, additionally a portfolio of solvers is employed for each of the representations. We extend this multi-representation portfolio for discrete optimisation tasks in the graphical models domain, realising a portfolio which won the UAI 2014 Inference Competition.

We identify a fundamental flaw in empirical evaluations of many portfolio and runtime prediction methods. The fact that solvers exhibit a runtime distribution has not been considered in the setting of runtime prediction, solver portfolios, or automated configuration systems, to date these methods have taken a single sample as ground-truth. We demonstrated through a large empirical analysis that the outcome of empirical competitions can vary and provide statistical bounds on such variations.

Finally, we consider an elastic solver which capitalises on the runtime distribution of a solver by launching searches in parallel, potentially on thousands of machines. We analyse the impact of the number of cores on not only solution time but also on energy consumption, the challenge being to find a optimal balance between the two. We highlight that although solution time always drops as the number of machines increases, the relation between the number of machines and energy consumption is more complicated. We also develop a prediction model, demonstrating that such insights can be exploited to achieve faster solutions times in a more energy efficient manner.

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not ben submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced. Parts of this work have appeared in the following publications which have been subject to peer review:

- Barry Hurley and Barry O'Sullivan. Adaptation in a CBR-Based Solver Portfolio for the Satisfiability Problem. In *Proceedings of the 20th International Conference on Case-Based Reasoning Research and Development, ICCBR 2012*, volume 7466 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2012. doi: 10.1007/978-3-642-32986-9_13.

- Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2014*, volume 8451 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2014. ISBN 978-3-319-07045-2.

- Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki, and Simon de Givry. Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints*, 21 (3):413–434, 2016. ISSN 1572-9354. doi: 10.1007/s10601-016-9245-y.

- Barry Hurley and Barry O'Sullivan. Statistical regimes and runtime prediction. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 318–324, 2015.

Parts of this work have also appeared in the following technical reports:

- Barry Hurley, Serdar Kadioglu, Yuri Malitsky, and Barry O'Sullivan. Transformation based Feature Computation for Algorithm Portfolios. Technical Report abs/1401.2474, January 2014. URL `http://arxiv.org/abs/1401.2474`.

- Barry Hurley, Deepak Mehta, and Barry O'Sullivan. Elastic Solver: Balancing Solution Time and Energy Consumption. Technical Report arXiv:1605.06940, May 2016. URL `http://arxiv.org/abs/1605.06940`.

Parts of this dissertation have appeared in the following book chapters:

- Barry Hurley and Barry O'Sullivan. Introduction to Combinatorial Optimisation in Numberjack. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 3–24. 2016. doi: 10.1007/978-3-319-50137-6_1.

- Barry Hurley, Lars Kotthoff, Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Advanced Portfolio Techniques. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 191–225. 2016. doi: 10.1007/978-3-319-50137-6_8.

- Barry Hurley, Lars Kotthoff, Barry O'Sullivan, and Helmut Simonis. ICON Loop Health Show Case. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 325–333. 2016. doi: 10.1007/978-3-319-50137-6_14.

- Barry Hurley, Barry O'Sullivan, and Helmut Simonis. ICON Loop Energy Show Case. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 334–347. 2016. doi: 10.1007/978-3-319-50137-6_15.

The contents of this dissertation extensively elaborate upon previously published work and mistakes (if any) are corrected.

Barry Hurley

December 2016

Acknowledgements

Firstly, I owe an enormous debt of gratitude to my supervisor, Professor Barry O'Sullivan. I was extremely lucky to have had a supervisor of such calibre and stature. Barry was supremely enabling and encouraging at every stage, I am extremely grateful for the opportunities he afforded me throughout my studies.

My family offered enormous support, without which this dissertation would not have been possible. To my parents Sean and Frances, sister Karen, and girlfriend Sarah, thank you.

To all those in 4C and subsequently Insight with whom I was lucky enough to collaborate with and have many interesting conversations with, there are too many to thank explicitly, however I owe a special thanks to Milan De Cauwer, Tadhg Fitzgerald, Diarmuid Grimes, Lars Kotthoff, Yuri Malitsky, Deepak Mehta, Pádraig O'Duinn, and Helmut Simonis.

Sabin Tabirca was a key figure in the early years of my interest in computer science, his teaching and guidance during the Munster Programming Training outreach programme was instrumental in fostering my interest in computer science.

Simon de Givry and Thomas Schiex gave me a fantastic opportunity to collaborate at INRIA, Toulouse, and to enjoy *la Ville Rose*, thank you.

This dissertation would not have been possible without the support of Science Foundation Ireland Grants 10/IN.1/I3032 and SFI/12/RC/2289, and EU FP7 FET-Open Grant 284715.

Barry

Chapter 1

Introduction

1.1 Motivation

Combinatorial decision and optimisation problems arise in many important real-world applications such as scheduling, planning, configuration, rostering, timetabling, vehicle routing, network design, bioinformatics, and many more. Intelligent, automated approaches to these problems can provide high quality solutions from a number of perspectives such as sustainability, energy efficiency, cost, time, etc., and can scale to tackle large problems far beyond the reach of manual methods. Optimisation technologies have been used to design a fibre optical network for entire countries, minimising the amount of cable to be laid, while also maintaining certain levels of redundancy [118]; to design electricity, water, and data networks [146]; to schedule scientific experiments on the Rosetta-Philae mission [145]; assign gates to airplanes [147]; as well as numerous timetabling, scheduling, and configuration applications [139, 159].

There exist a number of alternative approaches to solve combinatorial problems, three of the most prominent methods being *Constraint Programming* (CP) [139], *Boolean Satisfiability* (SAT) [23], and *Mixed Integer Programming* (MIP) [162]. These techniques provide a generic platform to tackle a broad range of problems, from simple puzzles to large scale industrial applications. They provide a framework upon which real-world problems can be specified declaratively, largely relieving the user of the task of specifying how a solution should be found.

It is generally possible to solve the same problem with any of these methods, however they differ in terms of problem representation and solution methodology.

In a nutshell, in the constraint programming paradigm variables take their values from finite sets of possibilities, with solutions typically found using a combination of systematic backtracking search and polynomial-time inference algorithms that reduce the size of the search space. A satisfiability problem is defined in terms of Boolean variables and a single form of constraint, namely a disjunction of Boolean variables or their negations. Instances are also solved using backtracking search, using unit-propagation for inference, as well as learning new clauses when failures are encountered. The mixed integer programming problem is defined by a set of linear expressions over integer and real-valued variables. Solutions are typically found by branch and bound search, using linear relaxations to make decisions, and the generation of cutting planes to prune the search space.

It is often not clear which approach is best for a particular problem, thus we may employ a higher-level modelling language to aide in the process. Modelling platforms such as Essence [51], MiniZinc [123], and Numberjack [78] offer the user the ability to declare their problem in a high-level language such as constraint programming, and enable it to be solved using one of the underlying technologies through the application of polynomial-time transformations. Additionally, each of these representations may be solved by one of many advanced solvers in each field. However, this adds an additional level of complexity in that there may be significant performance differences between different solvers on different instances. No single solver may dominate all others. Instead, the best overall performance might be achieved by considering a portfolio of complementary solvers. This idea forms the basis for solver portfolios which have highly effective empirically recently. The concept originates from the economics concept of the same name, whereby the risk is distributed across a collection of investments.

The work presented in this dissertation considers the application of portfolio techniques to combinatorial problems such as the constraint satisfaction, satisfiability, and graphical model problems. We present a number of contributions to the field, outlined in the following section.

1.2 Thesis Statement and Contributions

In this dissertation we defend the following two sub-theses which centre around the exploitation of machine learning for solving combinatorial decision and optimisation problems. Each sub-thesis is briefly discussed and references given to where each claim is defended.

Sub-thesis 1. *To-date the application of machine learning to improving the efficiency with which combinatorial problems can be solved has focused on either selecting a solver from a portfolio of possibilities, or on tuning how a specific solver should be used. We claim that a machine learning approach can provide even greater improvement in problem solving efficiency if it can select amongst a set of alternative problem representations in addition to solver choice.*

Discussion. In Chapter 3, we employ a number of aggregation techniques from voting theory for a portfolio-based on case-based reasoning, we also employ lazy clustering methods which take runtime into account and proves highly effective in practice. Employing historical performance data, on similar instances seen previously, we explore a number of techniques to adapt the case-base knowledge to new instances, ranking solvers by expected performance. We demonstrate that combining a standard k-nearest neighbour clustering with simple aggregation heuristics is able to effectively and accurately rank solvers by expected performance on unseen instances. These contributions have appeared in the following publication:

> Barry Hurley and Barry O'Sullivan. Adaptation in a CBR-Based Solver Portfolio for the Satisfiability Problem. In *Proceedings of the 20th International Conference on Case-Based Reasoning Research and Development, ICCBR 2012*, volume 7466 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2012. doi: 10.1007/978-3-642-32986-9_13.

We propose a novel hierarchical portfolio that does not rely on a single problem representation but may transform the problem instance to an alternate representation using a portfolio of encodings. Additionally a portfolio of solvers is employed for each of the representations. We show it is necessary to take the representation into account when building the portfolio. An empirical evaluation demonstrates the complementary nature of such a portfolio and ultimately its superior performance to that of a portfolio based on a single representation. The results are presented in the following publication, and are elaborated upon in Chapter 4:

> Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2014*, volume 8451 of *Lecture Notes*

> *in Computer Science*, pages 301–317. Springer, 2014. ISBN 978-3-319-07045-2.

We extend the multi-representation portfolio to the domain of graphical models. Realising a portfolio for discrete optimisation tasks in the graphical models domain, which won the *Maximum A-Posteriori* (MAP) task at the UAI 2014 Inference Competition by employing complementary solvers and representations. Supporting material is presented in Section 4.7 and has appeared in the following publication:

> Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki, and Simon de Givry. Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints*, 21(3):413–434, 2016. ISSN 1572-9354. doi: 10.1007/s10601-016-9245-y.

Sub-thesis 2. *The complex runtime distributions exhibited by combinatorial solvers on a range of interesting problem instances pose a challenge to the standard methodology in algorithm selection and configuration which does not take a holistic view of such distributions. Considering the runtime distribution in a more holistic fashion provides greater insight into solver performance, but also presents a range of challenges that the research community should focus more purposefully upon.*

Discussion. We identify a fundamental flaw in empirical evaluations of many portfolio and runtime prediction methods. The fact that solvers can exhibit a complex runtime distribution, e.g. it might be fat- or heavy-tailed [71], has not been considered in the setting of runtime prediction, solver portfolios, or automated configuration systems. To date these methods have taken a single sample as ground-truth. We demonstrated through a large empirical analysis that the outcome of empirical competitions can vary and provide statistical bounds on such variations. We also project the fragility of state-of-the-art runtime prediction methods to runtime distributions, showing that it is insufficient to take a single sample of the runtime in the current practice. These findings are presented in Chapter 5 and have appeared in:

> Barry Hurley and Barry O'Sullivan. Statistical regimes and runtime prediction. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 318–324, 2015.

Finally, we consider an *elastic solver* which exploits the runtime distribution of a solver by launching searches in parallel, potentially on thousands of machines.

We analyse the impact of the number of cores on not only solution time, but also on energy consumption, the challenge being to find a optimal balance between the two. We highlight that although solution time always drops as the number of machines increases, the relationship between the number of machines and energy consumption is more complex. We also develop a prediction model using machine learning, demonstrating that such insights can be exploited to achieve faster solutions times in a more energy-efficient manner. Preliminary results on this topic have appear in the following technical report:

> Barry Hurley, Deepak Mehta, and Barry O'Sullivan. Elastic Solver: Balancing Solution Time and Energy Consumption. Technical Report arXiv:1605.06940, May 2016. URL `http://arxiv.org/abs/1605.06940`.

Numberjack. Much of the implementation work of this dissertation has been contributed to the Numberjack modelling platform[1]. Numberjack is an open-source tool upon which combinatorial problems can be model by a user in a high-level language and solved efficiently using a number of possible backend-solvers. Contributions to the platform throughout this dissertation were widespread. Along with general bug-fixes and maintenance work, the build-system was migrated from a series of fragile *makefiles* to Python's built-in *distutils* system, making it much easier to compile and support across different platforms and versions. Support for Python 3 was also added. The SAT encoding layer was almost entirely re-written in such a way that variables, constraints, and expressions can be translated under a number of encodings. New integrations to a number of CP, SAT, and MIP solvers were added, including Minion, Gecode, Gurobi, CPLEX, and numerous SAT solvers.

1.3 Overview of the Dissertation

This dissertation is structured as follows. Chapter 2 presents an introduction to the fields of constraint satisfaction and satisfiability. Instances of the satisfiability problem often originate from a higher-level description such as the constraint satisfaction problem, so some techniques for encoding the former to the latter are presented. Finally, we introduce the field of solver portfolios which has been

[1] `http://numberjack.ucc.ie`

highly successful at leveraging complementary-strengths among solvers, eliciting the state of the art in a number of fields.

In Chapter 3 we examine a number of adaptation methods in a portfolio based on case-based reasoning. We demonstrate that a combination of lazy-learning techniques and simple aggregation heuristics is able to achieve dramatic performance improvements.

A powerful portfolio for the constraint satisfaction problem is presented in Chapter 4 which additionally considers a portfolio of representations. A hierarchical portfolio is outlined which, in addition to a portfolio of CSP solvers, considers a portfolio of SAT encodings and subsequently a portfolio of SAT solvers. This presents a wide-range of possible representations and solution techniques. Machine learning and solver portfolio techniques are employed to choose amongst the representations and solvers. An empirical evaluation on a large, extensive set of constraint satisfaction benchmarks demonstrates that substantial performance benefits can be achieved by considering alternate, complementary representations on top of a portfolio of solvers. Furthermore, this knowledge was subsequently applied to the graphical models domain whereby a portfolio was constructed and submitted to the UAI 2014 Inference Competition, achieving first place in both the 20 and 60 minute categories. This provides independent corroboration of the benefits of a toolkit consisting of multiple representations and solvers.

Chapter 5 studies the effect of runtime distributions on state-of-the-art portfolio techniques. Underpinning many portfolios is the requirement to predict the relative performance of its component solvers. Yet the fact that solvers can exhibit a complex runtime distribution has not been considered previously. We highlight a fundamental flaw in certain types of empirical evaluations and advocate for a more probabilistic approach.

In Chapter 6, we study a system which exploits the runtime distribution of solvers in a parallel setting, which we call an *elastic* solver that considers overall energy consumption when running many searches in parallel. We analyse the impact of the number of machines (or cores) on both solution time and on energy consumption. The overall energy consumption can be reduced by running in parallel, exploiting the lower end of the runtime distribution to find a solution quicker and terminate the remaining searches.

Finally, some general concluding remarks and outlook on future extensions of the work presented in this dissertation is discussed in Chapter 7.

Chapter 2

Background and Related Work

Summary. *This chapter presents some background and related work to the field of combinatorial optimisation. First, the constraint satisfaction, satisfiability, and mixed integer programming problems are introduced. Subsequently, a more detailed presentation is given for aspects pertinent to this dissertation, such as encodings from the constraint satisfaction to satisfiability problems, and solver portfolios.*

2.1 Numberjack

This chapter will present an introduction to the field of combinatorial optimisation, specifically in the context of Numberjack[1]. Numberjack is a library, written in Python, that allows the user to model and solve combinatorial optimisation problems. It provides a common interface to a number of underlying fast C/C++ solvers across different optimisation paradigms seamlessly and efficiently, presenting an ideal base upon which the work of this dissertation can build.

Much of the implementation work of this dissertation has been contributed to the Numberjack modelling platform. The background introduction to combinatorial optimisation will thus be presented using some examples in Numberjack. This should enable the work of this dissertation to be exploited by users of the platform and make the work extensible.

In particular, research on satisfiability encodings are typically limited to a new encoding of a single constraint type, with empirical evaluations being limited to

[1] The Numberjack website: `http://numberjack.ucc.ie`

benchmarks involving solely this constraint type [4, 5, 31, 50, 60, 100, 148]. To get a broader picture of how a new encoding behaves when used in combination with other constraints, one should benchmark on a wide range of instance types. This can prove difficult since it may involve implementing encodings for many different constraints. As part of this work, Numberjack's encoding layers were revamped in order to provide flexibility to encode a problem under different encodings. The encoding layers have been implemented in a flexible manner so that it is straightforward to implement a new encoding for a constraint or variable. As a result, it should be more practical to implement the new encoding simply for the constraint type and reuse the existing encodings for the various other constraint types, including any permutation of encodings available for them.

2.2 Modelling using Numberjack

A combinatorial optimisation model of a problem consists of a declarative specification, separating in so far as possible the formulation and the search strategy. However, modelling a problem effectively can be seen as an art in itself. The difficulty lies in producing a *solvable* model, i.e. one that quickly finds optimal solutions or determines that none exist. Naturally, there are many alternative models for a single problem; often it is not clear which one is best.

The basic process of developing a model consists of first defining what constitutes the variables and their corresponding domains, i.e. what decisions need to be made and what are the possible outcomes that can be taken for each one. Next, the constraints on the relationships between the variables must be defined. If some criterion is to be optimised, an objective function needs to be specified. Finally, if the model is well defined it can be passed off directly to a solver which will search for a (optimal) solution. Often, we may need to specify some heuristics about how the solver should perform the search, such as the variable or value ordering before the solver can effectively solve the problem.

For a user, the process of developing a suitable model often requires a number of iterations, depicted in Figure 2.1. Two common issues arise in the development of a solution: the model either does not accurately represent the problem, or a solution is not found by the solver in reasonable time. The former is more of a real-world problem requiring the assistance of a domain expert, where eliciting the true constraints of the problem, which may not even be well understood, is a challenge. The latter can pose a larger challenge from a number of perspectives

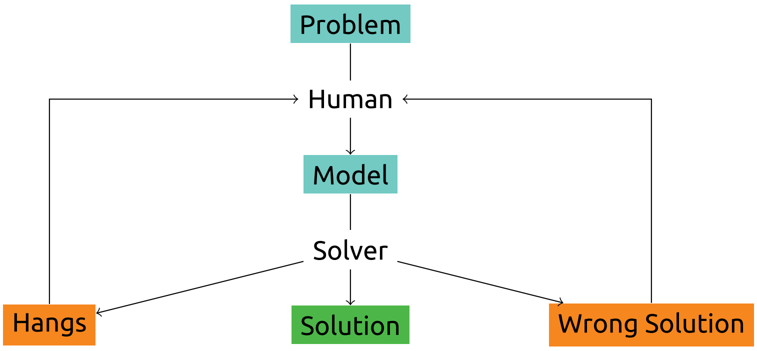


Figure 2.1: Abstract process of modelling a problem.

and may require the input of an expert in combinatorial optimisation.

Many different viewpoints can be taken in modelling a problem, so it can be easy to come up with a single model, but it may not necessarily be an efficient model. Important choices are to be made such as what are the variables, what are their domains, and what restrictions should be stated between them. Such decisions naturally affect the form of constraints which can be applied and unquestionably the effectiveness of the solver in finding a solution. Empirical performance may not be clear until it is actually evaluated.

Furthermore, solvers vary in terms of their capabilities, e.g. despite the hundreds of *global constraints* that have been developed [18], each solver typically implements a relatively small subset. Thus, the choice of which solver to use may be dictated by the global constraints required for a problem. Modelling languages lift the limitation of developing a model using a single solver. Instead, the model is implemented in a high-level solver-independent language that can be translated or encoded for a number of solvers. Nevertheless, these systems still rely on the user to produce a good model of their problem.

The next sections describe in more detail the primary components of a combinatorial optimisation model.

2.2.1 Variables

The variables constitute a fundamental component of a combinatorial problem. They are each represented by the finite set of values from which they can be assigned, often defined by a lower and upper bound. Typically these are restricted to integer values but some extensions do consider real-valued, set [63, 167], and graph [41, 44] variables. Boolean variables can take the values *true* or *false*, but are often interpreted interchangeably as 1 and 0, respectively. The ultimate task is to assign each variable to a value from its domain. The product of the variable domains defines the search space. Thus, it is important that each domain is defined as tightly as is permissible.

Some examples of how variables may be declared in Numberjack are given below. The VarArray and Matrix constructs serve as convenience methods for declaring groups of related variables.

```
Variable()         # Boolean variable
Variable('x')      # Boolean variable called 'x'
Variable(u)        # Variable with domain of [0..u−1]
Variable(l, u)     # Variable with domain of [l..u]
Variable(alist)    # Variable with domain specified as a list

VarArray(N)        # Array of N Boolean variables
VarArray(N, 'a')   # Array of N Boolean variables, called 'a'
VarArray(N, u)     # Array of N variables with domains [0..u−1]
VarArray(N, l, u)  # Array of N variables with domains [l..u]

Matrix(N, M)       # N x M matrix of Boolean variables
Matrix(N, M, 'b')  # N x M matrix of Boolean variables, called 'b'
Matrix(N, M, u)    # N x M matrix of variables with domains [0..u−1]
Matrix(N, M, l, u) # N x M matrix of variables with domains [l..u]
```

2.2.2 Constraints

Constraints define relationships between the variables, forbidding invalid solutions to the problem. A *unary constraint* is the simplest form of constraint involving a single variable and is satisfied by preprocessing the domain of the variable. *Binary constraints* relate two variables, such as saying they cannot be equal, and *global constraints* [18] involve a larger set of variables, modelling more complex relations. The combined collection of constraints that define a problem is called the constraint network. A formal definition on the constraint network and how it relates to the constraint satisfaction problem is given in Section 2.3.1.

The remainder of this section presents some common binary constraints, whereas Section 2.2.4 is devoted to the presentation of global constraints.

One of the most basic binary constraints is the disequality constraint which simply states that two variables must not be assigned the same value, for example $X \neq Y$. Inequalities such as $\langle <, \leq, \geq, > \rangle$ state a relationship which must hold between the respective assignments. In terms of their respective abilities to narrow the search space, these inequality constraints are stronger than the disequality constraint.

The *tightness* of a constraint is a measure of how many assignment tuples are forbidden, and subsequently how much of the search space is pruned. In particular, for a disequality constraint, with a tightness of $\frac{1}{d}$, we may only infer that a value may be removed from the domain of the opposite variable when one of the variables has been assigned, whereas for the inequalities, changes in the bounds or absence of certain values may reduce the domain of the other variable in the constraint. Such constraints are trivially specified in Numberjack using operator overloading, examples of which are presented in Table 2.1.

The expressivity of these binary constraints may be augmented by using expressions of the form $X + c < Y$, where c is a constant. Here, the expression $X + c$ becomes a *view* on the variable X, mirroring the offset domain without increasing the search space. Such constraints are useful in many scenarios, for example in scheduling if we would like to express the constraint that task2 starts after task1 has finished, we might specify a constraint of the form:

$$\text{task1}_{start} + \text{task1}_{duration} < \text{task2}_{start}$$

In many applications the model requires knowledge of the satisfiability of a particular constraint. In this case, we may *reify* the truth value of a constraint to a Boolean variable by writing something of the form:

z == (x < y) z <= (x < y) (x == y) != (a == b)

The first statement reifies the less-than relationship between x and y, enforcing

Table 2.1: Example binary constraint definitions in Numberjack.

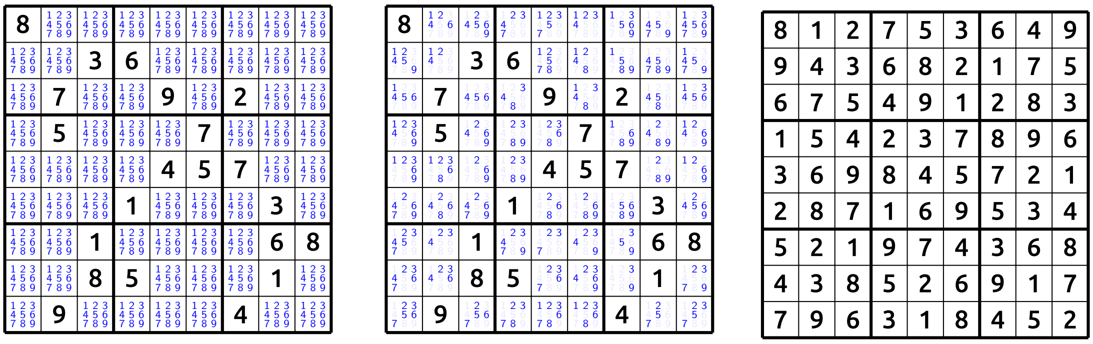
Constraint	Numberjack code
Disequality	x != z
Greater than	x > y
Less-or-equal	y <= z
Logical-or	x | y
Logical-and	y & z

that z is 1 iff x is less than y and 0 otherwise. The second example ensures that z is 0 if x is not less than y, if z is 1 then the less than relationship must hold, and other relationships are undefined. Finally, the third statement constrains the two pairs (x, y) and (a, b) such that exactly one pair must be assigned the same value and one pair must be assigned different values.

2.2.3 Inference

A central component in solving a CSP involves inferring variable information based on the constraints and the current state of the search, removing values from the domains that cannot possibly participate in any solution [116]. Based on the current partial assignment to variables during search, a value in a domain of an unassigned variable may, if assigned, violate a constraint then it is said to be *inconsistent*. Therefore it can be removed from the domain. No possible extension of the current assignment allows such a value to participate in a solution. These values are said to be *pruned* from the domain and consequently parts of the search tree will not be explored.

Figure 2.2 depicts the outcome of performing inference on a Sudoku problem which has been modelled as a CSP. Figure 2.2(a) shows the initial state of the CSP, where each cell corresponds to a single variable and its domain is the values from 1 to 9. Some cells have been pre-assigned with clues. Constraints of the problem enforce that cells within each row, column, and 3 by 3 block take unique values, i.e. a series of all-different constraints. Evidently, where an initial clue is given, no cell in the corresponding row, column, or block may take this value, and so these values can be removed from their domains. Before we start any search, inference



(a) Initial domains with preset clues.

(b) After propagating the preset clues.

(c) Complete solution.

8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

Figure 2.2: Example of CP propagation on a Sudoku instance.

can be performed based on the all-different constraints, and the information given by the present clues, to remove inconsistent values in corresponding variables. Figure 2.2(b) depicts the result of propagating this knowledge, and values which cannot participate in any solution are removed from the domains of variables, resulting in a smaller search space. During search, this process is repeated in circumstances such as when a new variable has been assigned or backtracking has occurred.

Note that iteratively propagating the constraints to the domains is typically enough to solve quintessential Sudoku problems. However, the example Sudoku presented in Figure 2.2 requires a combination of search, albeit a very small amount, and inference to find the complete solution depicted in Figure 2.2(c). We must remark that the Sudoku example depicts a rather simple aspect of consistency, nevertheless it serves to illustrate the concept. Constraint programming and other combinatorial optimisation systems offer the ability to perform much more sophisticated reasoning, some of which is discussed in the following section.

Enforcing consistency during search reduces the search space but comes at an increased computational cost at each node. A trade-off must be made between pruning the search space and searching at a faster rate. Thus, constraint programming offers different levels of consistency that can be enforced, from constraint level *local consistency* to *global consistency* [19]. Local consistency concerns individual constraints in isolation, whereas global consistency equates to a complete solution satisfying all constraints. Generally speaking, each additional level of consistency has the capability to prune larger parts of the search space but entails a higher computational complexity. A formal definition of a general consistency level, generalised arc-consistency, is given in Section 2.3.1.

2.2.4 Global Constraints

Global constraints define constraints over an arbitrarily sized set of variables, presenting many benefits for constraint programming [157]. Notably, they can succinctly convey complex relationships between variables, allowing for a concise specification of a problem. More importantly, from a pragmatic perspective, this enables higher levels of reasoning to be performed by dedicated inference algorithms, reducing the search space significantly. For example, propagation for global constraints such as *all-different* and *cardinality* constraints can be achieved in low polynomial time using flow-based algorithms [136, 137], much more efficiently than general purpose consistency algorithms.

To illustrate an example of such reasoning, consider an all-different constraint over the variables $\mathcal{X} = \{X_1, \ldots, X_5\}$, with initial domains $D(\mathcal{X}) = \{1, \ldots, 5\}$, declaring that each variable in the set must be assigned a unique value. Suppose that the domains have been reduced during search to those listed is Figure 2.3(a). Note that the domain of variables $\{X_1, X_2, X_3\}$ constitute the Hall set $\{1, 2, 5\}$, whereby these three variables must each be assigned a unique value from the Hall set. Thus, any assignment of these values to other variables in the constraint can never result in a satisfying assignment, so they can be removed from the domains of the remaining variables, $\{X_4, X_5\}$. Had the all-different constraint been decomposed into a clique of dis-equalities, then such reasoning could not have been performed.

The Global Constraint Catalogue [18] collects definitions for all global constraints defined in the CP literature, at the time of writing this listing contains over 400 constraints and is continually increasing. Such a vast catalogue provided many opportunities for the application of constraint programming, however one practical issue faced by users is in identifying which one is appropriate for their problem.

2.2.4.1 Example Global Constraints

In practice, most constraint solving libraries only provide implementations for a small number of those listed in the global constraint catalogue. This section describes some of the most prominent and widely used global constraints.

Linear Sum. This general expression constrains the dot-product linear combination of a vector of variables and a vector of coefficients. Mathematically, these constraints take the form:

$$\sum_i w_i \cdot x_i \quad \triangle \quad c$$

$$\begin{array}{lcl}
X_1 & \in & \{1, 2, 5\} \\
X_2 & \in & \{1, 2, 5\} \\
X_3 & \in & \{1, 2, 5\} \\
X_4 & \in & \{ 2, 3, 4 \} \\
X_5 & \in & \{1, 2, 3, 4, 5\}
\end{array}$$

(a) Initial domains.

$$\begin{array}{lcl}
X_1 & \in & \{1, 2, 5\} \\
X_2 & \in & \{1, 2, 5\} \\
X_3 & \in & \{1, 2, 5\} \\
X_4 & \in & \{ 3, 4 \} \\
X_5 & \in & \{ 3, 4 \}
\end{array}$$

(b) After propagating the Hall set.

Figure 2.3: Example of propagation on a Hall set $\{1, 2, 5\}$.

where w is a vector of integer or real valued weights, x is a vector of variables, $\triangle$ is a relational operator from the set $\langle <, \leq, =, \geq, > \rangle$, and c is a constant.

This is the only constraint type expressible in integer linear programming but it provides a flexible representation since a number of high-level constraints can be decomposed or encoded in this form. For example, the constraint $x > y$ can be written in linear form as $x - y > 0$. Additionally, since they only deal with problems in a standard form it enables integer programming solvers to perform high-levels of reasoning, proving extremely powerful [162].

A linear sum of variables can be expressed in a number of ways in Numberjack, for example each of the following are equivalent:

```
2*a + b + 0.5*c + 3*d          == e
Sum([2*a, b, 0.5*c, 3*d])      == e
Sum([a,b,c,d], [2,1,0.5,3])    == e
```

In general, it is expensive and difficult for a constraint programming solver to perform a large amount of reasoning on linear sum constraints, particularly if there is a large number of variables or their domains are large. For example, in a linear sum with a large number of variables, there is a huge number of possible assignment permutations in which to check for supports, at least until a number of variables are fixed. Thus, in practice, their use with constraint programming solvers is often limited to cases with a small number of variables and small domains.

All-Different. One of the most widely known, intuitive, and well studied global constraints is the *all-different* constraint [108, 136] which simply specifies that a set of variables must be assigned distinct values. Such a relation arises in many practical applications such as resource allocation, e.g. to state that a resource may not be used more than once at a single time point. An all-different constraint may be specified in Numberjack simply by passing a list of variables (or a VarArray) as follows:

```
AllDiff([x1, x2, x3, x4])
AllDiff(vararray)
```

An intuitive application of the all-different constraint is the Sudoku problem, as illustrated in Figure 2.2, whereby each row, column, and 3×3 cell is constrained to take distinct values. Such a condition can be modelled using an all-different for each row, column, and cell, giving a model with a total of 27 global constraints.

The all-different constraint may also be decomposed into a clique of dis-equalities between every pair of variables ($\forall i < j : X_i \neq X_j$). This decomposition requires $\binom{n}{2}$ binary constraints for each all-different, equating to a total of 972 (810 unique) binary disequality constraints for the Sudoku problem. However, this formulation looses the strong propagation that all-different enables, resulting in a larger search space to be explored.

Global Cardinality. The global cardinality constraint [3] places lower and upper bounds on the number of occurrences of certain values amongst a set of variables. The global cardinality constraint models restrictions in applications such as timetabling when there may be a limit on the number of consecutive activity types. For example in Numberjack, we can write the following:

```
myvariablearray = VarArray(10, 1, 5)
Gcc(myvariablearray, {3: [2, 2], 2: [0, 3], 4: [1, 10]})
```

to state that amongst the variables in ‘myvariablearray’, the value 3 must occur exactly twice, the value 2 at most three times, and the value 4 at least once.

Element. The element constraint [79] allows indexing into a variable or value array, at solving time, by the value of another variable. This can provide a very powerful modelling construct. A simple example of its use in Numberjack is:

```
myvariablearray = VarArray(10, 1, 20)
indexvar = Variable(10)
y == Element(myvariablearray, indexvar)
```

This uses the value assigned to ‘indexvar’ as an index into the variable array ‘myvariablearray’, binding the resulting variable to be equal to the variable ’y’.

Cumulative. The cumulative constraint [7] proves extremely useful in many scheduling and packing problems. Two significant and important application areas for constraint programming. For example, in a scheduling scenario with a given set of tasks, each requiring a specific quantity of resource, the cumulative constraint restricts the total consumption of the resource to not exceed a predefined limit at each time point. Tasks are allowed to overlap but their cumulative resource consumption must not exceed a predefined fixed limit. Figure 2.4 illustrates an example schedule of five overlapping tasks on a resource with a capacity of 5. Given the scheduling of task 1 at time point 0, the earliest task 2 can start is 3 since its resource consumption is 2. Task 4 on the other hand can

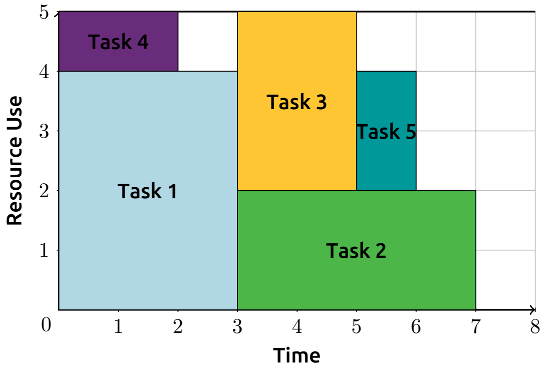


Figure 2.4: Example task assignment on a cumulative resource.

also start at 0, since its resource consumption of 1 fits within the remaining capacity. The cumulative constraint may also be viewed as modelling the packing of two-dimensional rectangles.

2.2.5 Optimisation

Numerous industrial applications of combinatorial optimisation require going beyond a single satisfiable solution. Frequently the interest is in finding good, or the absolute best, quality solution. For example, we might wish to define the *objective function* to minimise cost, wastage, loss, or to maximise profit, yield, customer satisfaction, and so on. These expressions can intuitively be specified in Numberjack as follows:

```
Minimise( openingcosts + supplycosts )
Maximise( Sum(items, weights) )
```

Different approaches are taken to solve such optimisation problems. Constraint programming can treat the objective function as another variable, performing branch and bound search on its range. It solves a series of satisfaction sub-problems, searching for a solution with an objective value below a certain threshold. On each subsequent call, the threshold is reduced until the problem is proven unsatisfiable or a resource limit has been exceeded. A satisfiability solver can similarly be used to solve some optimisation approaches, although its practicality is limited to problems where the domain of the objective function is small. Graphical model solvers perform sophisticated reasoning on the feasibility of bounds and values of local cost functions to tighten bounds on the objective. The application of the technology tends to be targeted at small, highly non-linear objective functions. Mixed integer programming solvers are most naturally suited to solving (linear) optimisation problems. The linear relaxations at their core yield effective lower-bounds. Critically, a MIP solver also examines the dual of the problem, yielding an upper-bound. Combining the two gives a precise indication of the range within which the optimal solution lies; when the two bounds are equal, optimality has been proven.

2.3 Solving Technologies

This section presents a more formal description of the aforementioned approaches to solving combinatorial problems, as well as outlining some notation used throughout the dissertation.

2.3.1 Constraint Programming

The constraint satisfaction problem (CSP) is defined by a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, defining the variables, domains, and constraints respectively. A solution to a CSP consists of a mapping from each variable in $\mathcal{X}$ to one of the values in its domain, such that all constraints in $\mathcal{C}$ are satisfied. Solutions are typically found using a combination of inference and backtracking-style search which will be covered in Section 2.4.

The graph composed of nodes representing the variables and (hyper-)edges between the nodes representing the scopes of each constraint is often referred to as the *constraint network*. A formal definition is given Definition 2.1, adapted from [19], and Definition 2.2 subsequently gives a formal definition of a constraint.

Definition 2.1 (Constraint Network)**.** *A constraint network is defined by a tuple* $\mathcal{N} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$*, where:*

- $\mathcal{X} = (X_1, \ldots, X_n)$ *is a finite sequence of integer variables,*
- $\mathcal{D} = D(X_1) \times \ldots \times D(X_n)$ *defines the domain for* $\mathcal{X}$*, where* $D(X_i) \subseteq \mathbb{Z}$ *is the finite set of values that variable* X_i *can take (the domain for* X_i*), and*
- $\mathcal{C} = (c_1, \ldots, c_e)$ *is a set of constraints.*

Definition 2.2 (Constraint)**.** *A constraint* c *is a relation defined on a sequence of variables* $X(c) = (X_i, \ldots, X_{|X(c)|})$*.* c *defines the subset of* $\mathbb{Z}^{|X(c)|}$ *that contains the combination of values (or* tuples*) that satisfy* c*.* $|X(c)|$ *is called the* arity *of* c*.*

The concept of *generalised arc-consistency* describes the relationship between the state of the domains $\mathcal{D}$ to the consistent assignments allowed by the constraints in $\mathcal{C}$. Definition 2.3, courtesy of [19], formally defines the concept of generalised arc-consistency for a constraint network.

Definition 2.3 ((Generalised) Arc-Consistency ((G)AC))**.** *Given a CSP network $\mathcal{N} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, a constraint $c \in \mathcal{C}$, and a variable $X_i \in \mathcal{X}(c)$,*

- *A value $v_i \in D(X_i)$ is consistent with c in $\mathcal{D}$ iff there exists a valid tuple τ satisfying c such that $v_i = \tau[X_i]$. Such a tuple is called a* support *for (X_i, v_i) on c.*
- *The domain $\mathcal{D}$ is (generalised) arc-consistent on c for X_i iff all the values in $D(X_i)$ are consistent with c in $\mathcal{D}$.*
- *The network $\mathcal{N}$ is (generalised) arc consistent iff $\mathcal{D}$ is (generalised) arc-consistent for all variables in $\mathcal{X}$ on all constraints in $\mathcal{C}$.*
- *The network $\mathcal{N}$ is arc inconsistent if $\emptyset$ is the only domain tighter than $\mathcal{D}$ which is (generalised) arc-consistent for all variables on all constraints.*

In so far as is possible, constraint programming attempts to separate the definition of a problem from the solving process, to the extent that it is said to represent the holy grail of programming: "the user states the problem, the computer solves it" [48].

2.3.2 Satisfiability

The satisfiability problem (SAT) [23] is one of the most prominent and long-standing areas of study in computer science, most notably by being the first problem to be proven $\mathcal{NP}$-complete and lying at the heart of the $\mathcal{P} \overset{?}{=} \mathcal{NP}$ question [33]. The problem consists of a set of Boolean variables and a propositional formula over these variables. The task is to decide whether or not there exists a truth assignment to the variables such that the propositional formula evaluates to true, and, if this is the case, to find this assignment.

SAT instances consist of a propositional logic formula, usually expressed in conjunctive normal form (CNF). The representation consists of a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a Boolean variable or its negation. Each clause is a disjunction of its literals and the formula is a conjunction of each clause. The following SAT formula is in CNF:

$$(x_1 \lor x_2 \lor \neg x_4) \land (\neg x_2 \lor \neg x_3) \land (x_3 \lor x_4)$$

This instance consists of four SAT variables. One assignment to the variables which would satisfy the above formula would be to set $x_1 = true$, $x_2 = false$, $x_3 = true$, and $x_4 = true$.

2.3.3 Mixed Integer Programming

The mixed integer programming (MIP) [162] problem consists of a set of linear constraints over integer and real-valued variables, where the goal is to find an assignment to the variables minimising a linear objective function. More formally, a MIP problem takes the form:

$$\begin{aligned} min \quad & cx + dy && (2.1)\\ s.t. \quad & Ax + By \geq 0 && (2.2)\\ & x, y \geq 0 && (2.3)\\ & y \text{ integer} && (2.4) \end{aligned}$$

where x and y are two vectors of real-valued and integer variables, respectively. c and d are vectors of coefficients defining the objective function to be minimised. The matrices A and B represent coefficients of a set of linear constraints.

Analogous to the constraint and satisfiability solving techniques seen in previous sections, modern techniques for solving a mixed integer programming problems consist of a combination of search and various forms of inference. Firstly, a number of pre-solving techniques are applied which rewrite and reduce some parts of the constraints. This maintains the same form of problem, while generally resulting in a reduced, tighter problem.

Subsequently, the space of solutions is explored using branch and bound search. At each node in the search tree, the integrality constraints on variables in y are relaxed, the resulting formulation, namely the *LP relaxation*, is solved to optimality using linear programming techniques such as the simplex algorithm [122]. If it happens that the solution also satisfies the integrality constraints, then a feasible solution has been found. The best integer solution found during search is called the *incumbent* and its objective value provides an upper-bound on the optimal solution value.

In practice however, an integer solution to the LP relaxation rarely occurs and so the fractional solution is used to guide the search. Furthermore, the objective value of the non-integral solution also provides a lower-bound on the solution of the integral problem. The distance between the best lower and upper bound is deemed the *optimality gap*, when its value reaches zero, optimality has been proved. The search procedure then branches on one of the y variables for which a non-integral value was assigned. For example, if integer variable y_i was assigned the value 2.8 in the LP relaxation solution, then two sub-problems are created with constraints $y_i \leq 2$ and $y_i \geq 3$ respectively. If the solution to the LP relaxation in any of the resulting sub-problems is infeasible or is greater than the incumbent, then that node can be dropped and another node explored. This process is repeated recursively until optimality is proven or the problem is proved infeasible.

2.3.4 Cost Function Networks

A Cost Function Network (CFN) extends a Constraint Network by using non-negative cost functions instead of constraints [120]. A CFN is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{W}, k)$ where $\mathcal{X}$ is a set of n discrete variables with a domain of possible assignments from $\mathcal{D}$, $\mathcal{W}$ is a set of non-negative functions, and k, a possibly infinite maximum cost. Like the CSP, each variable $X_i \in \mathcal{X}$ has a finite domain of values that can be assigned to it, denoted $D(X_i) \in \mathcal{D}$. A function $w_S \in \mathcal{W}$, with scope $S \subseteq \mathcal{X}$, is a function $w_S : D_S \mapsto \{\alpha \in \mathbb{N} \cup \{k\} \ : \ \alpha \leq k\}$, where D_S denotes the Cartesian product of all $D(X_i)$ for $X_i \in S$. In CFNs, the cost of a complete assignment is the sum of all cost functions. A solution has cost less than k. Therefore a cost of k denotes forbidden assignments, used in hard constraints. A solution of minimum cost is sought.

The problem is also known as the Weighted Constraint Satisfaction Problem (WCSP). It can be viewed as an optimisation version of the CSP whereby, rather than of having a hard-restriction on satisfying every constraint, a cost is associated with the violation of constraints and the cumulative sum is to be minimised. Such a formalism is suited to a number of importants practical domains such as computer vision and bioinformatics.

2.3.5 Choice is good

As the previous sections have outlined, the solution technologies for constraint programming, satisfiability, mixed integer programming, and cost function networks are all operationally different. Specifically, CP uses constraint propagation with backtracking search; SAT utilises unit-propagation, clause learning, and search; and MIP exploits linear relaxations, cutting planes, with branch and bound search. Often, it is not clear which solution technology is best suited to a particular problem so it can be worthwhile to experiment with different approaches. Fortunately, the user does not need to manually produce a different model for each approach since many problems can be encoded between CP, SAT, MIP, and CFN; a process which can be significantly simplified by using modelling frameworks such as Essence [51], MiniZinc [123], and Numberjack [78]. The following sections illustrate the performance differences between approaches on some example problems.

2.3.5.1 Example: Warehouse Location Problem

The Warehouse Location Problem [80] considers a set of existing shops and a candidate set of warehouses to be opened, the problem is to choose which warehouses are to be opened and, consequently, the respective shops which each one will supply. There is a cost associated with opening each warehouse, as well as a supply cost for each warehouse-shop supply pair, the objective being to minimise the total cost of warehouse operations and supply costs. A complete Numberjack model for the warehouse location problem is given in Figure 2.5.

Table 2.2 compares the performance of a mixed integer programming solver and a constraint programming solver, namely SCIP and Mistral respectively, on some instances of the Warehouse Location Problem. SCIP is able to solve each of instance to optimality very quickly, whereas the CP solver takes over one hour of CPU-time to find solutions of worse quality. In this case, the CP solver is not able to perform much reasoning on the objective function for this problem, a weighted linear sum, whereas the MIP solver is able to produce tight bounds very quickly and narrow the search.

```
model = Model()

# A vector of 0/1 variables for each warehouse to decide which ones to open
WareHouseOpen = VarArray(data.NumberOfWarehouses)

# A matrix of 0/1 variables for each shop (row) decide which warehouse (column) will supply it
ShopSupplied = Matrix(data.NumberOfShops, data.NumberOfWarehouses)

# Cost of running warehouses
warehouseCost = Sum(WareHouseOpen, data.WareHouseCosts)

# Cost of shops using warehouses
transpCost = Sum([Sum(varRow, costRow) for varRow, costRow in zip(ShopSupplied, data.SupplyCost)])

# Objective function
obj = warehouseCost + transpCost
model += Minimise(obj)

# Channel from store opening to store supply matrix
for col, store in zip(ShopSupplied.col, WareHouseOpen):
    model += [var <= store for var in col]

# Make sure every shop is supplied by one warehouse
for row in ShopSupplied.row:
    model += Sum(row) == 1

# Make sure that each warehouse does not exceed it's supply capacity
for col, cap in zip(ShopSupplied.col, data.Capacity):
    model += Sum(col) <= cap

# Load the model with a named solver
solver = model.load("SCIP")

# Ask the solver to solve
solver.solve()

if solver.is_sat():
    ... # print solution
elif solver.is_unsat():
    print "Unsatisfiable"
```

Figure 2.5: Model of the Warehouse Location Problem in Numberjack.

Table 2.2: Performance comparison between a mixed integer programming solver (SCIP) and a constraint programming solver (Mistral) on some instances of the Warehouse Location Problem.

Instance	SCIP			Mistral		
	Objective	Nodes	Time	Objective	Nodes	Time
`cap44`	1184690	1	0.84	1468957	10008044	>3600
`cap63`	1087190	14	1.82	1388391	10683754	>3600
`cap71`	957125	1	0.69	1297505	11029722	>3600
`cap81`	811324	1	0.65	1409091	3497095	>3600
`cap131`	954894	5	5.30	1457632	1281009	>3600

```
model = Model()

# N variables with domains 1..N representing the column of point in each row
seq = VarArray(N, 1, N)

# Points must be placed in distinct columns
model += AllDiff(seq)

# Each row of the triangular distance matrix contains no repeat distances
for i in range(N-2):
    model += AllDiff([seq[j] - seq[j+i+1] for j in range(N-i-1)])
```

Figure 2.6: Model of the Costas Array Problem in Numberjack.

2.3.5.2 Example: Highly Combinatorial Puzzles

We compare a constraint programming, a satisfiability, and a mixed integer programming solver on some benchmarks of two arithmetic puzzles. Specifically, constructing a Costas Array and constructing a Golomb ruler of minimal size. Both of these problems are parameterised by a single value specifying the size of the instance. The Costas Array problem [35] is to place n points on an $n \times n$ board such that each row and column contains only one point, and the pairwise distances between points is also distinct. This can be modelled using a vector of n variables to decide the column of each point, and enforcing all-different constraints on the vector of variables and on the triangular distance matrix. A Golomb ruler [155] is defined by placing a set of m marks at integer positions on a ruler such that the pairwise differences between marks are distinct. The objective is to find rulers of minimal length. Numberjack models for the Costas Array and Golomb Ruler problems are presented in Figures 2.6 and 2.7 respectively. Problems such as these are not limited to academic interest but do map to many real world applications.

Table 2.3 illustrates the empirical performance differences between CP, SAT, and MIP approaches on these problems. Here, the constraint programming solver (Mistral) is very effective. The satisfiability solver performs comparably well on the Costas array problem, but when dealing with the optimisation problem of the Golomb ruler, it fails to scale. However, it does outperform the mixed integer programming solver which performs very poorly on these problems.

```
model = Model()

# A vector of finite domain variables for the position of each mark
marks = VarArray(m, 2**(m-1))

# Pairwise distances are distinct
distance = [marks[i] - marks[j] for i in range(1, m) for j in range(i)]
model += AllDiff(distance)

# Symmetry breaking
model += marks[0] == 0
for i in range(1, m):
    model += marks[i-1] < marks[i]

# Minimise the position of the last mark
model += Minimise(marks[-1])
```

Figure 2.7: Model of the Golomb Ruler Problem in Numberjack.

2.4 Systematic Search

Chronological backtracking search plays a central role in the solution process for combinatorial problems. Nodes in the search correspond to variables, and branches to assignments, thus the search explores the tree of possible partial solutions. Figure 2.8 illustrates a partial example of the search tree generated by backtracking search. Initially, from the *root node*, the variable X is branched on, taking one branch for each possible value in its domain.

Modern constraint programming solvers typically perform *binary-branching* on the assignment or removal of a value from the domain. The process of *maintaining arc-consistency* (MAC) [142] during search has been shown to be highly effective. This consists of making the initial CSP arc-consistent before starting search, then again after every assignment and every backtrack. A *domain wipeout* occurs when a variable has no values remaining in its domain. When this occurs search must backtrack and explore a different path. A solution has been found when all variables have been assigned a value in their domain which is globally consistent with the constraints.

Notably, if a bad decision is made early in the search, then the resulting sub-tree may be unsatisfiable. It may take exponential time for the search to prove that no solution exists in the sub-tree, a *refutation*, before backtracking to the bad decision node [84]. The *thrashing* phenomenon occurs when the current partial

Table 2.3: Performance of a constraint programming, satisfiability, and mixed integer programming solver on two arithmetic puzzles of increasing size. Values are CPU time in seconds, '-' represents a timeout, and 'M' a memory limit of 2 GB exceeded.

Instance	Mistral	MiniSat	SCIP
Costas (11)	0.0	0.0	27.0
Costas (12)	0.0	0.0	166.0
Costas (13)	0.0	0.0	286.0
Costas (14)	1.0	0.0	1065.0
Costas (15)	9.0	0.0	2564.0
Costas (16)	52.0	16.0	-
Costas (17)	562.0	163.0	-
Costas (18)	529.0	677.0	-
Golomb (6)	0.0	0.0	2.0
Golomb (7)	0.0	0.0	17.0
Golomb (8)	0.0	2.0	59.0
Golomb (9)	0.0	34.0	1778.0
Golomb (10)	3.0	M	-
Golomb (11)	133.0	M	-
Golomb (12)	3006.0	M	M

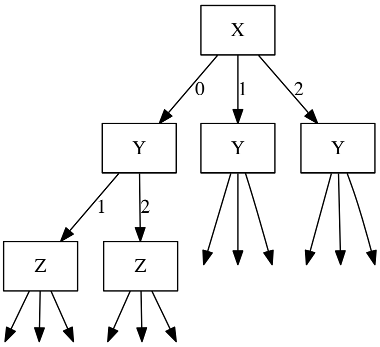


Figure 2.8: A partial example of the search tree generated by backtracking search.

assignment cannot be extended to a solution but search continues backtracking on the remaining variables, trying all possible values when the real source of inconsistency is a bad decision higher up the tree.

To avoid such worst-case behaviour, a number of methods such as randomised restarting, back-jumping, and explanation-based search have been proposed. Nevertheless, an important decision to be made arises concerning what order the tree should be explored. These topics are discussed in the following sections.

2.4.1 Search Heuristics in Constraint Programming

Two closely-related decisions which are vital for success are the choice of variable to branch on and the subsequent value it will be assigned. These decisions have a dramatic affect on the size of the search tree that will be explored. Interestingly, an oracle proposing the value ordering could lead search directly to a solution without backtracking (if the problem is satisfiable), regardless of the variable ordering. In practice however, such an oracle is implausible so heuristic methods must be used.

The CSP community has devised a number of generic, problem independent heuristics for users to choose from. Options range from static heuristics such a selecting the variables in order of their domain size or degree of connectivity in the constraint-graph, to dynamic heuristics based on the activity of the solver during search such as weighted heuristics [26], and impact-based [135] to name a few.

To avoid bad decisions early in the search tree, the variable ordering heuristic, in general, follows a *fail-first* principle [75] whereby variables likely to lead to failure should be chosen first. Effort should be focused on difficult parts of the problem likely to lead to failure, which should ideally occur early in the search. Value ordering heuristics on the other hand try to select the most promising value, one most likely to lead to a solution [56].

Choosing an effective heuristic is a highly problem dependant task, often requiring intimate knowledge of the underlying technology, an undertaking often beyond the reach of many users. Automating such a task, simplifying the barrier to entry for users, has been proposed as one of the grand challenges for constraint programming [49]. One approach to this is to use a machine learning model to automatically select the heuristic based on instance specific features [32, 61, 107, 119].

2.4.2 Restarting and Randomness

In practice, the search procedure will encounter many failures and have to backtrack. As mentioned previously, one risk occurs if a bad decision has been made early in the search process and proving that no solution exists in the sub-tree may take exponential time. One approach to avoiding such behaviour is to restart the search from the root node after a pre-defined limit on the number of failures has been reached [115].

To maintain the completeness of the search process, solvers adopt a restarting strategy whereby the failure limit eventually tends towards infinity. A restart strategy is defined by a sequence $\langle t_1, t_2, t_3, \ldots \rangle$ whereby each t_i specifies the limit on the number of failures for a particular run of the algorithm. Once the failure limit t_i is reached, the search is restarted from the root node with the new limit of t_{i+1}.

Two standard restart strategies are based on the *Luby* and *geometric* sequences. The Luby [115] sequence has the form $\langle 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \ldots \rangle$. In the context of Las Vegas algorithms [16] it is proven to be universally optimal, achieving a runtime that is only a logarithmic factor from an optimal restart strategy where the runtime distribution of the underlying algorithm is fully known, and no other universal strategy can do better by more than a constant factor [115]. Alternatively, the geometric [160] sequence increases the cutoff by a constant factor between each run.

Restarting is typically combined with randomisation in the variable and value heuristics to avoid repeatedly exploring the same search space. Such stochastic behaviour gives rise to solvers exhibiting a distribution of runtimes. In some cases, modelled by heavy- and fat-tailed distributions [71], possibly with infinite mean and variance. These distributions capture a non-negligible fraction of runs far to the right or left of the median, runs taking extremely long. *Rapid randomised restarting* [66, 70] has been shown to eliminate heavy-tails to the right of the median and can even take advantage of heavy-tails to the left of the median.

2.5 Encoding a CSP as SAT

SAT, like CSP, has a variety of practical real world applications such as hardware verification, security protocol analysis, theorem proving, scheduling, routing, planning, digital circuit design [23]. The application of SAT to many of these problems is made possible by transformations from representations like the constraint satisfaction problem [134]. This section describes, in detail, a number of encodings for representing a constraint satisfaction problem as a satisfiability problem. In each case, encodings for variable domains are presented first, followed by a description of how the constraints are encoded.

2.5.1 Direct Encoding

The following sections present the *direct encoding* [161], also known as the *sparse encoding*. The translation is intuitive in that there is a very straightforward mapping from the CSP domain to Boolean SAT variables.

2.5.1.1 Variable Domains

A CSP variable X with domain $D(X) = \{1, \ldots, d\}$ can be represented in SAT using the direct encoding by creating a Boolean SAT variable for each value in its domain: $x_1, x_2, \ldots, x_d$. If x_v is *true* in the resulting SAT formula, then $X = v$ in the CSP solution. In order to represent a solution to the CSP, exactly one of $x_1, x_2, \ldots, x_d$ must be assigned *true*. An *at-least-one* (ALO) clause is added to the SAT formula for each CSP variable as follows:

$$\forall X \in \mathcal{X} : (x_1 \vee x_2 \vee \ldots \vee x_d).$$

For each variable X, *at-most-one* (AMO) of the SAT variables can be *true*. A number of encodings exist to achieve this and there is a steady stream of increasingly sophisticated encodings [31, 50, 60, 148]. The simple *pairwise-AMO* encoding uses a quadratic number of clauses For each pair of distinct values in the domain of X, a binary clause enforces that at most one of the two can be assigned *true*. The series of these binary clauses ensure that only one of the SAT variables representing the variable will be assigned *true*, i.e.

$$\forall v, w \in \mathrm{D}(X) : (\neg x_v \vee \neg x_w).$$

2.5.1.2 Constraints

Constraints between CSP variables are represented in the direct encoding by enumerating the conflicting tuples. For binary constraints for example, binary conflict clauses are added, for each disallowed assignment, to forbid both values being used at the same time. For a binary constraint between a pair of variables X and Y, we add the conflict clause $(\neg x_v \vee \neg y_w)$ if the tuple $\langle X = v, Y = w \rangle$ is forbidden.

For intensionally specified constraints, all possible tuples must be enumerated to encode the disallowed assignments.

Disequality. To enforce that two variables are not equal, $X \neq Y$, a binary conflict clause is added for each value common between their domains. This can be achieved in $\mathcal{O}(d)$ time (assuming that variable domains are sorted) and will introduce $\mathcal{O}(d)$ binary clauses. More formally:

$$\forall v \in D(X) \cap D(Y) : (\neg x_v \vee \neg y_v)$$

Equality. Conversely, two variables can be constrained to take equal assignments, $X = Y$. For each value in the intersection of the two domains, clauses are added so that their respective Boolean variables take the same assignment. For each value outside the intersection a unary clause is added to ensure it is *false*. This encoding can be achieved in $\mathcal{O}(d)$ time and will introduce $\mathcal{O}(d)$ clauses. More formally:

$$\begin{aligned}
\forall v \in D(X) \cap D(Y) : &\quad (\neg x_v \vee \neg y_v) \\
\forall v \in D(X) \setminus D(Y) : &\quad (\neg x_v) \\
\forall v \in D(Y) \setminus D(X) : &\quad (\neg y_v)
\end{aligned}$$

Encoding the equality constraint may not seem useful since a natural option is to rewrite the problem so that the two variables are replaced by a new variable with the intersection of the two domains, but the constraint can be important in a reified scenario. Details on how each of these constraint can be reified is given in Section 2.5.6.

Inequalities. Constraints of inequality between two variables X and Y, such as $\{<, \leq, >, \geq\}$, can be generalised to the form $X + k \leq Y$. Thus, binary conflict clauses are added for each assignment which does not satisfy this relation. This can be achieved in $\mathcal{O}(d^2)$ time and will introduce $\mathcal{O}(d^2)$ binary clauses. More formally:

$$\forall i \in D(X), \forall j \in D(Y), i + k > j : (\neg x_i \vee \neg y_j)$$

Example 2.5.1 (Direct Encoding)**.** Consider a simple CSP with three variables $\mathcal{X} = \{X, Y, Z\}$, each with domain $\langle 1, 2, 3 \rangle$. The constraints consist of $Y > Z$, and all-different constraint over the variables: alldifferent(X, Y, Z), which we represent by encoding the pairwise dis-equalities. Table 2.4 shows the complete direct-encoded CNF formula for this CSP. The first 12 clauses encode the domains of the variables, the remaining clauses encode the constraints between X, Y, and Z. There is an implicit conjunction between these clauses.

Table 2.4: An example CSP encoded to SAT under the direct encoding.

Domain Clauses	$(x_1 \vee x_2 \vee x_3)$ $(\neg x_1 \vee \neg x_2)$ $(\neg x_1 \vee \neg x_3)$ $(\neg x_2 \vee \neg x_3)$ $(y_1 \vee y_2 \vee y_3)$ $(\neg y_1 \vee \neg y_2)$ $(\neg y_1 \vee \neg y_3)$ $(\neg y_2 \vee \neg y_3)$ $(z_1 \vee z_2 \vee z_3)$ $(\neg z_1 \vee \neg z_2)$ $(\neg z_1 \vee \neg z_3)$ $(\neg z_2 \vee \neg z_3)$
$X \neq Y$	$(\neg x_1 \vee \neg y_1)$ $(\neg x_2 \vee \neg y_2)$ $(\neg x_3 \vee \neg y_3)$
$X \neq Z$	$(\neg x_1 \vee \neg z_1)$ $(\neg x_2 \vee \neg z_2)$ $(\neg x_3 \vee \neg z_3)$
$Y \neq Z$	$(\neg y_1 \vee \neg z_1)$ $(\neg y_2 \vee \neg z_2)$ $(\neg y_3 \vee \neg z_3)$
$Y > Z$	$(\neg y_1 \vee \neg z_1)$ $(\neg y_1 \vee \neg z_2)$ $(\neg y_1 \vee \neg z_3)$ $(\neg y_2 \vee \neg z_2)$ $(\neg y_2 \vee \neg z_3)$ $(\neg y_3 \vee \neg z_3)$

2.5.2 Support Encoding

Related to the direct encoding is the *support encoding* [58, 101] which shares the same domain representation but differs in how constraints are encoded. Instead of clauses for the conflicting assignments, the support encoding specifies the supported values for a given partial assignment. An interesting property of the encoding is that if a constraint has no consistent values in the domain of the corresponding variable, a unit-clause will be added to prune the values from the domain that cannot exist in any solution. Also, a solution to a SAT formula without the *at-most-one* constraints on variable domains, represents an arc-consistent assignment to the CSP. Unit propagation on this SAT instance establishes arc-consistency in optimal worst-case time for establishing arc-consistency [58].

2.5.2.1 Variable Domains

The support encoding uses the same mechanism as the direct encoding to encode CSP domains into SAT – each value in the domain of a CSP variable is encoded as a SAT variable which represents whether or not it takes that value. See Section 2.5.1.1 for details on the domain encoding.

2.5.2.2 Constraints

The support encoding differs from the direct encoding in terms of how the constraints between variables are encoded. Unlike the direct encoding which encodes the conflicting assignments, the support encoding generates clauses with literals that support a partial assignment. Given a constraint between two variables X and Y, for each value v in the domain of X, let $S_{Y,X=v} \subset D(Y)$ be a strictly smaller subset of the values in the domain of Y which are consistent with assigning

$X = v$. Either x_v is *false* or one of the consistent assignments from $y_1 \ldots y_d$ must be true. This is encoded in the support clause

$$\neg x_v \vee \left(\bigvee_{i \in S_{Y,X=v}} y_i \right).$$

Conversely, for each value w in the domain of Y, a support clause is added for the supported values in X which are consistent with assigning $Y = w$. In the case wwere all values are supported, such that $S_{Y,X=v} = D(Y)$, then no clause needs to be generated, since it would be subsumed by the at-least-one clause of the variable Y. The support encoding in general requires $\Theta(2d)$ clauses, each of size $\mathcal{O}(d)$.

Example 2.5.2 (Support Encoding)**.** Table 2.5 gives the complete support-encoded CNF formula for the simple CSP given in Example 2.5.1. The first 12 clauses encode the domains and the remaining ones the support clauses for the constraints. There is an implicit conjunction between clauses. Notice that for the constraint $Y > Z$, two unit-clauses are generated, which prune inconsistent values from the domains.

Table 2.5: An example CSP encoded to SAT under the support encoding.

Domain Clauses	$(x_1 \vee x_2 \vee x_3)$ $(\neg x_1 \vee \neg x_2)$ $(\neg x_1 \vee \neg x_3)$ $(\neg x_2 \vee \neg x_3)$ $(y_1 \vee y_2 \vee y_3)$ $(\neg y_1 \vee \neg y_2)$ $(\neg y_1 \vee \neg y_3)$ $(\neg y_2 \vee \neg y_3)$ $(z_1 \vee z_2 \vee z_3)$ $(\neg z_1 \vee \neg z_2)$ $(\neg z_1 \vee \neg z_3)$ $(\neg z_2 \vee \neg z_3)$
$X \neq Y$	$(\neg x_1 \vee y_2 \vee y_3)$ $(\neg x_2 \vee y_1 \vee y_3)$ $(\neg x_3 \vee y_1 \vee y_2)$ $(\neg y_1 \vee x_2 \vee x_3)$ $(\neg y_2 \vee x_1 \vee x_3)$ $(\neg y_3 \vee x_1 \vee x_2)$
$X \neq Z$	$(\neg x_1 \vee z_2 \vee z_3)$ $(\neg x_2 \vee z_1 \vee z_3)$ $(\neg x_3 \vee z_1 \vee z_2)$ $(\neg z_1 \vee x_2 \vee x_3)$ $(\neg z_2 \vee x_1 \vee x_3)$ $(\neg z_3 \vee x_1 \vee x_2)$
$Y \neq Z$	$(\neg y_1 \vee z_2 \vee z_3)$ $(\neg y_2 \vee z_1 \vee z_3)$ $(\neg y_3 \vee z_1 \vee z_2)$ $(\neg z_1 \vee y_2 \vee y_3)$ $(\neg z_2 \vee y_1 \vee y_3)$ $(\neg z_3 \vee y_1 \vee y_2)$
$Y > Z$	$(\neg y_1)$ $(\neg y_2 \vee z_1)$ $(\neg y_3 \vee z_1 \vee z_2)$ $(\neg z_1 \vee y_2 \vee y_3)$ $(\neg z_2 \vee y_3)$ $(\neg z_3)$

2.5.3 Full Regular Encoding

Unlike the direct and support encoding, which model $X = v$ as a SAT variable, the *full regular encoding* [10] encoding creates Boolean SAT variables to symbolise $X \leq v$. This encoding is also known as the *order encoding* in the literature and also with the opposite semantics $X \geq v$. It has been shown to maintain tractability when encoding certain tractable classes of CSPs, which is not the case when encoding under the direct or support encoding [132, 133].

2.5.3.1 Variable Domains

A Boolean SAT variable $x_{\leq v}$ is created for each value v in the domain of a variable X. If X is less than or equal to v, then X must also be less than or equal to $v+1$ ($x_{\leq v+1}$). Therefore, clauses are added to enforce this consistency across the domain:

$$\forall v \in D(X) : (\neg x_{\leq v} \vee x_{\leq v+1}).$$

This linear number of clauses is all that is needed to encode the domain of a CSP variable into SAT under the order encoding. In contrast, the direct and support encodings require a quadratic number of clauses in the domain size. In practice the final variable $x_{\leq d}$ can be omitted since it must necessarily be true if $x_{\leq d-1}$ is false.

2.5.3.2 Constraints

The order encoding is naturally suited to modelling inequality constraints. To state $X \leq 3$, the unit clause ($x_{\leq 3}$) can be posted. The constraint $X = v$, can be rewritten as $(X \leq v \wedge X \geq v)$; $X \geq v$ can then be rewritten as $\neg X \leq (v-1)$. To state that $X = v$ in the order encoding, we would encode $(x_{\leq v} \wedge \neg x_{\leq v-1})$. A conflicting tuple between two variables, for example $\langle X = v, Y = w \rangle$ can be written in propositional logic and simplified to a CNF clause using De Morgan's Law as follows:

$$\begin{gathered}
\neg((x_{\leq v} \wedge x_{\geq v}) \wedge (y_{\leq w} \wedge y_{\geq w})) \\
\neg((x_{\leq v} \wedge \neg x_{\leq v-1}) \wedge (y_{\leq w} \wedge \neg y_{\leq w-1})) \\
\neg(x_{\leq v} \wedge \neg x_{\leq v-1}) \vee \neg(y_{\leq w} \wedge \neg y_{\leq w-1}) \\
(\neg x_{\leq v} \vee x_{\leq v-1} \vee \neg y_{\leq w} \vee y_{\leq w-1})
\end{gathered}$$

Example 2.5.3 (Full Regular Encoding)**.** Table 2.6 gives the complete full-regular encoded CNF formula for the simple CSP specified in Example 2.5.1. There is an implicit conjunction between clauses in the notation.

Table 2.6: An example CSP encoded to SAT under the full regular encoding.

Domain Clauses	$(\neg x_{\leq 1} \vee x_{\leq 2})$ $(\neg x_{\leq 2} \vee x_{\leq 3})$ $(x_{\leq 3})$ $(\neg y_{\leq 1} \vee y_{\leq 2})$ $(\neg y_{\leq 2} \vee y_{\leq 3})$ $(y_{\leq 3})$ $(\neg z_{\leq 1} \vee z_{\leq 2})$ $(\neg z_{\leq 2} \vee z_{\leq 3})$ $(z_{\leq 3})$
$X \neq Y$	$(\neg x_{\leq 1} \vee \neg y_{\leq 1})$ $(\neg x_{\leq 2} \vee x_{\leq 1} \vee \neg y_{\leq 2} \vee y_{\leq 1})$ $(\neg x_{\leq 3} \vee x_{\leq 2} \vee \neg y_{\leq 3} \vee y_{\leq 2})$
$X \neq Z$	$(\neg x_{\leq 1} \vee \neg z_{\leq 1})$ $(\neg x_{\leq 2} \vee x_{\leq 1} \vee \neg z_{\leq 2} \vee z_{\leq 1})$ $(\neg x_{\leq 3} \vee x_{\leq 2} \vee \neg z_{\leq 3} \vee z_{\leq 2})$
$Y \neq Z$	$(\neg y_{\leq 1} \vee \neg z_{\leq 1})$ $(\neg y_{\leq 2} \vee y_{\leq 1} \vee \neg z_{\leq 2} \vee z_{\leq 1})$ $(\neg y_{\leq 3} \vee y_{\leq 2} \vee \neg z_{\leq 3} \vee z_{\leq 2})$
$Y > Z$	$(\neg y_{\leq 1})$ $(z_{\leq 1} \vee \neg y_{\leq 2})$ $(z_{\leq 2} \vee \neg y_{\leq 3})$ $(z_{\leq 3})$

2.5.4 Regular Encoding

The full regular encoding can be augmented with the additional direct encoding representation of variable domains to produce the *regular encoding* [10]. A variable's domain is encoded in both representations and clauses are added to chain between them:

$$\forall v \in D(X) : x_v \rightarrow x_{\leq v} \wedge \neg x_{\leq v-1} \quad .$$

Chaining the two domain representations in this way allows for the at most one clauses from the direct encoding to be omitted, it is enforced through the chained representations. Thus only a linear number of clauses are required to encode a domain, rather than a quadratic number, albeit requiring twice the number of SAT variables.

Having both encodings gives flexibility in the representation of each constraint. For example, for inequalities the regular encoding can be used since it is naturally suited, but for a (dis)equality the direct encoding would be more natural. Throughout this dissertation, where the regular encoding is used, the encoding which gives the most compact formula is chosen, like in the previous example.

Example 2.5.4 (Regular Encoding)**.** Table 2.7 gives the complete regular-encoded CNF formula for the simple CSP specified in Example 2.5.1. There is an implicit conjunction between clauses in the notation.

Table 2.7: An example CSP encoded to SAT under the regular encoding.

Domain Clauses	$(x_1 \vee x_2 \vee x_3)$ $(\neg x_1 \vee x_{\leq 1})$ $(\neg x_2 \vee x_{\leq 2})$ $(\neg x_2 \vee \neg x_{\leq 1})$ $(\neg x_3 \vee x_{\leq 3})$ $(\neg x_3 \vee \neg x_{\leq 2})$ $(\neg x_{\leq 1} \vee x_{\leq 2})$ $(\neg x_{\leq 2} \vee x_{\leq 3})$ $(x_{\leq 3})$
	$(y_1 \vee y_2 \vee y_3)$ $(\neg y_1 \vee y_{\leq 1})$ $(\neg y_2 \vee y_{\leq 2})$ $(\neg y_2 \vee \neg y_{\leq 1})$ $(\neg y_3 \vee y_{\leq 3})$ $(\neg y_3 \vee \neg y_{\leq 2})$ $(\neg y_{\leq 1} \vee y_{\leq 2})$ $(\neg y_{\leq 2} \vee y_{\leq 3})$ $(y_{\leq 3})$
	$(z_1 \vee z_2 \vee z_3)$ $(\neg z_1 \vee z_{\leq 1})$ $(\neg z_2 \vee z_{\leq 2})$ $(\neg z_2 \vee \neg z_{\leq 1})$ $(\neg z_3 \vee z_{\leq 3})$ $(\neg z_3 \vee \neg z_{\leq 2})$ $(\neg z_{\leq 1} \vee z_{\leq 2})$ $(\neg z_{\leq 2} \vee z_{\leq 3})$ $(z_{\leq 3})$
$X \neq Y$	$(\neg x_1 \vee \neg y_1)$ $(\neg x_2 \vee \neg y_2)$ $(\neg x_3 \vee \neg y_3)$
$X \neq Z$	$(\neg x_1 \vee \neg z_1)$ $(\neg x_2 \vee \neg z_2)$ $(\neg x_3 \vee \neg z_3)$
$Y \neq Z$	$(\neg y_1 \vee \neg z_1)$ $(\neg y_2 \vee \neg z_2)$ $(\neg y_3 \vee \neg z_3)$
$Y > Z$	$(\neg y_{\leq 1})$ $(z_{\leq 1} \vee \neg y_{\leq 2})$ $(z_{\leq 2} \vee \neg y_{\leq 3})$ $(z_{\leq 3})$

2.5.5 Additional Encodings

Many more encodings have been proposed in the literature citing various alternate benefits. One notable encoding, namely the *full-log encoding* [57, 97, 161], encodes the binary representation of an integer variable. Encoding the domain of each CSP variable is highly compact in that it requires only $\lceil \log_2 d \rceil$ Boolean variables, however the encoding of constraints can be more intricate. The full-log encoding may also be combined with the direct encoding to produce the log encoding. Channeling the domain representations removes the need for the at-most-one clauses in the direct encoding and allows constraints to be encoded using either representation. This opens the door to many possibilities such as producing a more compact representation or one which encodes information from multiple perspectives.

DPLL-based SAT solvers obtain propagation similar to forward checking on a direct encoded instance, and maintaining arc-consistency on a support encoded instance. One disadvantage to the log encoding is that such properties are lost. The log-support encoding [53] aims to recover some higher level of propagation by combining the log and the support encodings.

2.5.6 Encoding Reified Constraints

Encodings for a number of CSP constraints can be reified with the addition of a slack variable to each clause which encodes the base-constraint. For example, to reify $Z \Leftrightarrow (X = Y)$, the clauses representing $X = Y$ are added as described in previous sections, but an additional literal $(\neg z)$ is also added to each clause; additionally clauses representing $X \neq Y$ must be added and should include the additional literal (z). Thus, if z is *true*, the literal $(\neg z)$ is falsified and the $X = Y$ clauses must be satisfied as normal, but the $X \neq Y$ clauses are satisfied by the additional literal (z).

2.6 Solver Portfolios

In many empirical comparisons, such as the CSP and SAT competitions, the winner is determined by the total number of instances solved across a heterogeneous benchmark set. Solvers thus compete to be crowned as the best general purpose solver for the benchmark set by solving the most problems within the time limit. However, this is often not representative of the state of the art for the field. Many solvers are often more efficient on specific classes of benchmarks, possibly by being designed purposely for them. They may not solve the most instances overall but may be the most efficient on a subset of instances. Thus, the state of the art may be elicited by taking the most efficient solver for each individual instance, not simply the best general purpose solver.

Solver portfolios [67, 83, 106] aim to exploit this property by replacing a single solver with a set of complementary solvers, along with a mechanism for selecting a subset to use on a particular problem. By making decisions at an instance-specific level, it is possible to make significant performance gains over any of the individual component solvers. Solver portfolios have been used with great success for solving both SAT and CSP instances in systems such as SATzilla [165], ISAC [98], and CPHydra [128].

Solver portfolios are an instance of the *Algorithm Selection Problem* [138] where the goal is to select the most appropriate algorithm for solving a particular problem. Figure 2.9 depicts an abstract model of the algorithm selection problem for combinatorial search. The model consists of a problem space $x \in \mathcal{P}$, an algorithm space $A \in \mathcal{A}$, a performance measure $p(A, x) \in \mathcal{R}^n$, and a selection model S. The goal of the selection model is to map each instance to the best algorithm according to the performance measure.

The majority of modern portfolio approaches employ some form of machine learning to take the role of the selection model. This involves a training phase whereby a reference set of instances, a domain-specific feature description, a candidate set of algorithms, and a performance metric are defined. Feature descriptions for each instance and performance data of each algorithm on each instance is recorded. The machine learning model is built such that the performance metric is maximised on this training data. Numerous additional machine learning techniques such as cross-validation, parameter tuning, feature normalisation or transformations, etc., may be employed at this stage to further improve the performance of the model. Subsequently, to apply this trained model to a new test instance at runtime, first the feature description must be computed and passed to the model to make a solver section. The chosen solver is then applied to the problem instance. Additionally some portfolio approaches may interrupt the chosen solver in place of another, based on the observed performance, illustrated by the dashed feedback arc in the figure.

The model illustrated in Figure 2.9 is by no means a comprehensive depiction of the real-world application of a solver portfolio, a number of practical issues must be accounted for. Firstly, we may encounter many easy instances for which it would be wasteful to spend time computing features and making a solver decision when it could be trivially solved directly. Thus, it is practical to have a static schedule of pre-solvers which will be run briefly before any intelligent selection takes place. Conversely, for certain larger instances we may be unable to compute features due to timeouts or memory limits, so it is useful to have a backup-solver which may be run in this scenario. Additionally, if the chosen solver is unable to handle the instance or crashes, then a successive solver may be run. Despite such practical issues, the figure serves to exemplify the concept at the heart of modern model-based solver portfolios. In the following sections, a number of different forms of these core selection models are discussed.

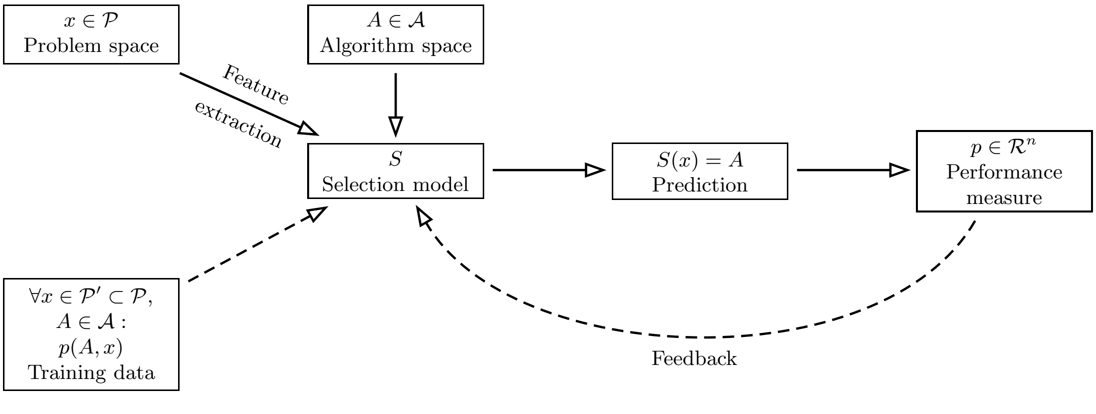


Figure 2.9: Model of the algorithm selection problem. Figure courtesy of [106].

2.6.1 Selection Models

At the heart of the system is the selection model, which typically consists of some form of machine learning. The machine learning model is built from some historical training data, such as performance information for the algorithms on a set of reference instances, along with appropriate feature descriptions. A model is fitted mapping the feature vector to the performance metric.

Eliciting a representative feature description for instances is crucial to the effective application of the machine learning models. Features for combinatorial search instances comprise of both static and dynamic features extracted from the instance. Static features range from simple statistics such as the number of variables, constraints, types of constraints, etc; to intricate graph statistics extracted from the constraint and variable graph representations. Dynamic features are recorded from short runs of solver to collect information such as probing data, or the distribution of constraint weights.

Once a meaningful feature vector has been built, there are typically three main approaches to using machine learning to build solver selection models, namely using classification, clustering, and regression techniques [106]. The following sections briefly explain the respective methods and mention some relevant work.

2.6.1.1 Classification for Solver Selection

The problem of predicting the best solver can be treated as a classification problem where the label to predict is the solver that will be run. For each instance, solvers are ranked based on their performance and the target prediction label becomes the solver with the best performance.

On the surface, this sounds like a natural model and a straightforward mapping to the application. However, one issue with this approach is the lack of consideration for the relative performance differences between solvers, whether two solvers are almost identical or orders of magnitude different is not considered. One alternative is to build an ensemble of pairwise classification models, whereby each model assumes the role to predict solver A versus B, A versus C, B versus C, and so on. The winner is chosen by the solver with the most votes. Additionally, this may be augmented by directly including the relative performance difference by weighting the classification task. This approach proved highly effective as the core mechanism in the latest iteration of SATzilla [165], winning numerous medals at recent SAT competitions.

2.6.1.2 Clustering for Solver Selection

Alternatively, based on the hypothesis that a solver's performance is consistent across closely-related instances, instances can be clustered based on their features. The solver with the best performance on a particular cluster is assigned as the cluster's label. The cluster membership of any new data decides the solver to use.

This clustering approach is adopted by Instance Specific Algorithm Configuration (ISAC) [98], a successful portfolio system for the SAT. Instances are clustered using the g-means algorithm, a variation of k-means with automated selection of the number of clusters parameter, k. The best performing solver in each cluster is then run through an automated algorithm configuration system to optimise its performance on that cluster of instances. At run time, cluster membership of the instance decides the solver to be run.

Such an approach depends on a very close relationship between the feature description and the performance. It relies on instances that look similar from the feature description perspective to have a similar performance by the respective solvers. Taking the performance metric into account during clustering is not something which has been tackled in the literature, to the best of our knowledge. Nevertheless, the clustering method has proved to be quite successful in empirical competitions and in the literature.

A lazy form of clustering, such as k-nearest neighbour clustering, may also be applied. At runtime, the k most similar instances are recalled. A number of aggregation techniques may then be applied in order to choose a solver, or produce a schedule of solvers. A notable application of this is CPHydra [128] which aggregates this information by solving an optimisation model, deciding on a schedule of the solvers which maximises the probability of solving the instance within the time limit.

2.6.1.3 Regression for Solver Selection

Finally, perhaps surprisingly, regression models can be trained to predict the performance of each portfolio solver in isolation. For each solver, a model learns a function mapping the feature vector to its performance, and the solver with the best predicted performance is chosen. When solver runtime is used as the performance metric, the typical practice in the literature is to predict the log of the runtime. We may also employ a collection of models in order to predict pairwise performance differences. The chosen solver is subsequently the one with the most votes from the pairwise models.

Solver portfolios based on regression models have been highly successful in empirical competitions. Most notably, initial versions of *SATzilla* [164] built ridge regression models to predict the log of the runtime for each component solver. Their analysis noted that the predicted runtimes can often be off by orders of magnitude but that the relative ordering of the predictions among solvers is generally correct, which is sufficient for the purposes of building a portfolio.

2.6.2 Non-Model Based Portfolios

The portfolio methods highlighted so far use machine learning models to make a prediction at runtime of the solver to be run, an alternative is to build a static schedule of solvers without the need for a selection mechanism at runtime. One strength of non-model based approaches is their applicability to new domains without the need for a domain-specific feature description.

One notable non-model based portfolio is *ppfolio* [140], which proved surprisingly successful, winning 16 prizes at the 2011 SAT Competition. The idea is quite simple, essentially consisting of a number of handcrafted rules to launch different sets of solvers depending on the number of cores available. Hoos et al. subsequently took a more refined approach by modelling the choice of solver schedule as a combinatorial problem in the *aspeed* [82] system. Based on empirical data, a multi-objective answer set programming model is solved to decide the optimal schedule of solvers, including the ability to generate parallel schedules exploiting multiple cores.

2.6.3 Disparity of Approaches

Many entries in the algorithm selection literature aim to tackle problems that are fundamentally similar but exist in different domains, for example portfolios for the CSP, SAT, MaxSAT, etc. However, there is a disparity between approaches such that no single approach has proved dominant. Recently, *AutoFolio* [114] was proposed as a generic meta-portfolio to tackle this issue. The decision of which algorithm selection approach to use is treated as a configuration problem. It constructs portfolios in a form resembling that of SATzilla, with initial pre-solvers, followed by feature and machine learning model selection. Automated configuration methods are used to make such decisions like the choice of machine learning model and subsequently to tune its parameters. Additionally, the aforementioned *aspeed* is employed to build its pre-solving component.

For further reading, an extensive survey of the wide-range of literature on the algorithm selection problem was recently published [106], with an additional tabulated form online. [2]

2.7 Chapter Summary

This chapter reviews the literature relevant to this dissertation. Combinatorial decision and optimisation problems such as the constraint satisfaction, satisfiability, and mixed integer programming have been introduced in the context of the Numberjack system. Some encodings from CSP to SAT have been described, and various methods for building solver portfolios have been presented. Subsequent technical chapters will present more focused background material, where relevant.

[2] Online Algorithm Selection Survey at `http://larskotthoff.github.io/assurvey/`

Chapter 3

Case-Based Reasoning for Solver Selection

Summary. *This chapter examines a number of adaptation methods for a portfolio for solving constraint satisfaciton problems based on case-based reasoning. Employing the knowledge of performance on previously seen similar instances, we explore a number of techniques to adapt to the new instance and select the candidate solver.*

The objective of the work reported in this chapter is to study a simple case-based reasoning approach to a portfolio for the Boolean satisfiability (SAT) problem. We present three large case-bases of problem-solving experience with a large number of modern SAT solvers in three distinct domains, including one comprising almost 1200 industrial problems (Section 3.1). We focus primarily on the problem of adaptation of problem-solving experience to solve new cases, having retrieved a suitable set of similar experiences involving problems similar to the one we wish to solve (Sections 3.2 and 3.3). Our results (Section 3.4) demonstrate that a case-based reasoning approach would perform close to oracle performance on the domains we evaluate, exhibiting a potential "killer" application domain for case-based reasoning. These results are consistent with the belief held in the SAT community that experience plays a key role in selecting a good solver for a problem instance [106].

3.1 Building Case-bases for SAT Solving

We summarise the representation, cases and similarity measure used in our three case-bases for SAT solving. Our case-bases relate to three domains: industrial instances, hand-crafted instances, and randomly generated instances.

3.1.1 Feature Representation

We employed the same set of SAT instance features as those used in SATzilla[1]. SATzilla is a successful algorithm portfolio for SAT, i.e. a system that uses machine learning techniques to select the fastest SAT solver for a given problem instance. That system uses a total of 48 features, summarised in Figure 3.1 [165].

These features can be summarised under nine different categories: problem size features; variable-clause graph features; variable graph features; balance features; proximity to Horn formula; DPLL probing features; and local search probing features. The first category of features are self explanatory, and simply relate to the number of variables and clauses in the SAT instance. The next two categories relate to two different graph representations of a SAT instance. The variable-

Problem Size Features:
1. **Number of clauses**: denoted c
2. **Number of variables:** denoted v
3. **Ratio:** c/v

Variable-Clause Graph Features:
4-8. **Variable nodes degree statistics:** mean, variation coefficient, min, max and entropy.
9-13. **Clause nodes degree statistics:** mean, variation coefficient, min, max and entropy.

Variable Graph Features:
14-17. **Nodes degree statistics:** mean, variation coefficient, min and max.

Balance Features:
18-20. **Ratio of positive and negative literals in each clause:** mean, variation coefficient and entropy.
21-25. **Ratio of positive and negative occurrences of each variable:** mean, variation coefficient, min, max and entropy.
26-27. **Fraction of binary and ternary clauses**

Proximity to Horn Formula:
28. **Fraction of Horn clauses**
29-33. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max and entropy.

DPLL Probing Features:
34-38. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256.
39-40. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes.

Local Search Probing Features:
41-44. **Number of steps to the best local minimum in a run:** mean, median, 10th and 90th percentiles for SAPS.
45. **Average improvement to best in a run:** mean improvement per step to best solution for SAPS.
46-47. **Fraction of improvement due to first local minimum:** mean for SAPS and GSAT.
48. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean over all runs for SAPS.

Figure 3.1: A summary of the features used to describe SAT instances in our case-base. These are the same features which used in SATzilla and which have been show to have a good correlation with instance hardness [165].

[1] `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/`

clause graph is a bipartite graph with a node for each variable, a node for each clause, and an edge between them whenever a variable occurs in a clause. The variable graph has a node for each variable and an edge between variables that occur together in at least one clause. The balance features are self explanatory and relate, primarily, to the distribution of positive and negative literals within the SAT instance. Another category measures the proximity to a Horn formula. This captures how close the SAT instance is to an important polynomial class of SAT that can be solved using the unit-propagation inference method used in all systematic SAT solvers. The DPLL probing features are related to statistics that a standard systematic search algorithm gathers while testing the difficulty of the instance [38]. The local search features are the non-systematic analogue of the latter category.

3.1.2 Case-Bases

We built three case-bases from the training data used by the SATzilla system [165].[2] Each case in the case-base represents one SAT problem instance and the individual performance of a set of solvers when applied to it. For each benchmark instance in the dataset, there is a record of whether each solver solved the instance within a specified cut-off time (1 hour), the time taken by each to solve the instance, and whether the solver crashed during execution. Each solver is run independently of each other. Thus, each case is a pair, (F, S), where F is a set of feature values and S is a set of pairs encoding the performance of each solver on the query instance.

The problem instances were originally taken from the benchmarks suites associated with the annual International SAT Competition.[3] We combine the instances from each year of the SAT Competition into a single combined dataset. Each instance is assigned to one of three categories based on the instance origin: handcrafted (HAN), industrial (IND) and randomly generated (RAN) instances. The instances are additionally separated into what SATzilla classified as satisfiable and unsatisfiable instances. For our evaluation, instances from these two classifications were combined into a single dataset. Table 3.1 gives the resulting size of each case-base.[4] Table 3.1 also shows the number of solvers in each. Note that the random category contains eight additional solvers, purposely designed for random instances, that are not present in the dataset for the handcrafted and industrial categories.

[2] SATzilla data: `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/`

[3] `http://www.satcompetition.org`

[4] The case-bases are available at `http://osullivan.ucc.ie/datasets/iccbr2012/`

Table 3.1: The number of problem instances and the number of solvers associated with each of our three case-bases.

Case-base	**# Instances**	**# Solvers**
Handcrafted (HAN)	1181	19
Industrial (IND)	1183	19
Random (RAN)	2308	27

3.1.3 Similarity Metric

The features that encode the cases are all numeric. Therefore, we will be assuming that the similarity between two cases is computed using the unweighted normalised Euclidean distance. Feature values are normalised to the interval $[0, 1]$. Specifically, for the ith feature of case γ, we compute the normalised value $\eta(\gamma[i])$ of feature value $\gamma[i]$ as follows:

$$\eta(\gamma[i]) = \frac{\gamma[i] - min(i)}{max(i) - min(i)} \quad ,$$

where $min(i)$ and $max(i)$ are the minimum and maximum values respectively for feature i across all cases.

When evaluating a test case, one finds the k cases with highest similarity (smallest Euclidean distance) in the case-base. The challenge is then, given the set of performance data for each solver, how should one adapt this experience to the current problem instance. We frame this problem as a label ranking problem [52], which we will discuss in greater detail in the following section.

3.2 Adaptation Strategies

The specific task of adaptation in our portfolio context is to decide which solver should be used to solve a given instance. We will consider a setting in which we are allocated one hour to solve each instance. The objective is, given a set of problem instances, to solve as many of them as possible within the cutoff in the shortest time. In other words, we lexicographically order these two objectives: maximise the number of solved instances, and tie-break by running time, which we prefer to minimise. In our setting, each nearest neighbour can be seen as giving an ordering over the performance of the available solvers. We can interpret this order as a ranking, possibly time-weighted.

In traditional classification, we are interested in assigning one or more labels from a finite set to a case. In contrast, label ranking deals with assigning a total ordering of labels to a case. This ranking of labels can be much more useful than assigning a single label, e.g. rank aggregation methods have been used to combine query results from multiple search engines [42]. In this chapter, the labels represent different algorithms in our portfolio solver and the ranking of labels represents the expected order of run time on an instance.

Label ranking may also be seen as a generalization of multi-label classification. Instead of classifying a case with a subset of the classes, we instead, assign a totally ordered ranking of the classes. Multi-label ranking is the task whereby in addition to producing a total ordering of the labels for an instance, the task is to also identify a partition of the labels into relevant and irrelevant labels [28, 30]. This introduces an additional layer of complexity to the task. Methods for learning pairwise preferences between labels have been proposed [52]. It has been shown that case-based label ranking compares well against model-based approaches [29]. An in depth survey of additional label ranking methods is given in [158].

We will consider a variety of voting based approaches for label ranking and consensus ranking; we refer the reader to the literature for further details of the various methods [130]. We will use examples throughout, based on the sample data presented in Table 3.2. In this table, we present an example of the retrieval set from our case-based system, but do not present the running times. Instead we simply order the solvers by running time, using the notation $a \prec b$ to annotate the relative performance.

3.2.1 Kendal-Tau Distance

To compare two rankings, we define a function to compute the distance between them. The Kendal-Tau distance between two rankings A and B is the number of

Table 3.2: An example list of label rankings, which we will use as a running example. Each ranking contains a total ordering of the labels a, b, c, d. The operator $\prec$ can be read as 'faster'.

Name	Label Ranking
A	$a \prec b \prec c \prec d$
B	$c \prec b \prec a \prec d$
C	$b \prec a \prec d \prec c$
D	$a \prec c \prec d \prec b$
E	$b \prec a \prec c \prec d$

discordant pairs, i.e. the number of pairs who's relative ranking is different. Let L be the set of all labels and let A and B be two complete rankings of these labels. More formally, the Kendal-Tau distance is defined as:

$$\text{KT-distance}(A, B) = \sum_{c,d \in L,\, c \neq d} \begin{cases} 1 & \text{if } A \text{ and } B \text{ rank } c \text{ and } d \text{ in a different order} \\ 0 & \text{otherwise}\,. \end{cases}$$

Example 3.2.1. Using the example rankings given in Table 3.2, the Kendal-Tau Distance between rankings A and B is:

$$\text{KT-distance}(A, B) = 3 \quad .$$

This is because the relative ranking between pairs of candidates (a, b), (a, c), and (b, c) is different amongst the two rankings. A and B both rank the other pairs, e.g. (c, d) , in the same order. □

3.2.2 Kemeny Consensus Ranking

The Kemeny Score of a ranking, R, is the sum of all the Kendal Tau distances from R to all rankings among the set of votes, V.

$$\text{Kemeny-Score}(R, V) = \sum_{v \in V} \text{KT-distance}(R, v) \quad .$$

Example 3.2.2. Let $R = \langle a \prec c \prec b \prec d \rangle$. If we take all the rankings from Table 3.2 as the votes, then the Kemeny Score of R is 9. This is the sum of:

$$\text{KT-distance}(R, A) = 1 \quad \text{KT-distance}(R, B) = 2 \quad \text{KT-distance}(R, C) = 3$$
$$\text{KT-distance}(R, D) = 1 \quad \text{KT-distance}(R, E) = 2$$

□

The Kemeny Consensus is the ranking of the labels that minimises the Kemeny Score. This may also be referred to as the optimal Kemeny ranking. It is the ranking which minimises, among the votes, the number of disagreements on the pairwise preference between every pair of labels.

Aggregating multiple rankings into a single optimal Kemeny ranking is NP-hard [42]. Section 3.3 formulates the problem of computing the optimal Kemeny ranking as a combinatorial optimisation problem and illustrates that such an approach can be highly efficient and practical.

Example 3.2.3. For the votes given in Table 3.2, the optimal Kemeny ranking is $\langle b \prec a \prec c \prec d \rangle$ with a Kemeny Score of 7. All other possible permutations of the labels have a higher Kemeny Score than this. In this case the Kemeny Optimal ranking matches one of the rankings in the votes, but this may not necessarily be the case. □

The Kemeny Consensus ranking is said to satisfy the Condorcet criterion. This states that if a candidate is preferred by most voters to any other candidate, then it should be ranked first in the aggregation ranking [25]. It expresses no condition on the remainder of the positions, however.

3.2.3 Borda Voting

Borda Voting is a polynomial time approximation scheme for the Kemeny Consensus ranking. Each of the k-nearest neighbours will vote for a label in the order for which that solver performed on that case. A label in position p receives $n-p+1$ points based on its position in the ranking. The points for each candidate are summed up. The ranking is produced by sorting these tallies in decreasing order. Borda Voting does not satisfy the Condorcet criterion.

Example 3.2.4. For vote B in Table 3.2, candidates c, b, a and d would receive 4, 3, 2 and 1 points respectively. If we sum up the points from all the votes in this table, the a would have a score of 16 points, b of 15, c of 12 and d of 7. The resulting ranking would be $\langle a \prec b \prec c \prec d \rangle$. □

3.2.4 Weighted Borda Voting

Weighted Borda Voting takes extra information about each neighbour into account during the Borda voting. The vote for a particular solver is multiplied by the weight in one of the two weighting schemes we consider. In distance-weighted borda voting (DW-BV), the weight W_D is given as $W_D = \frac{1}{1+d}$ where d is the Euclidean distance between the neighbour and the test instance.

In time-weighted borda voting (TW-BV) the weight W_T is given as $W_T = \frac{\text{cutoff}-t}{\text{cutoff}}$, where ‘cutoff’ is the cut-off execution time limit and t is the time taken for the solver on a given neighbour. Time-weighted borda voting gives a large weight to solvers that take very little time to solve the instance and a weight of zero to any that timeout or do not solve the instance. In theory this would seem to suit our goal of choosing the solver which will perform fastest for a given instance.

3.2.5 Copeland Voting

Copeland Voting looks at every pair of labels (a, b) and counts the difference between the number of votes that prefer a to b and those that prefer b to a. The label with the higher number of preferences gets one added to its score. The label with the lower number of preferences, gets one deducted from its score. The resulting ranking of the labels is obtained by sorting by their respective scores.

Example 3.2.5. Given the votes in Table 3.2, the label ranking produced by Copeland Voting would be $\langle b \prec a \prec c \prec d \rangle$. The scores for each label would be: (a, 1), (b, 3), (c, -1), (d, -3). □

3.2.6 Bucklin Voting

Bucklin Voting is a means of choosing the label with the best median ranking. The algorithm first attempts to select the label which has a majority of first preference votes. The number of first preferences for each label is counted across the votes. If one of the labels has a majority, then that label is the winner. If no label has a majority, then the second preference votes are added to the first. Again, if there is a label that has a majority of votes, then that label is the winner. There may be multiple labels with a majority. In this case, the winner is the one with the highest vote tally.

Since this voting mechanism only outputs the top ranked candidate, it is excluded from our area under the curve comparison as described in Section 3.4.2. It is included in our portfolio evaluation in Section 3.4.3.

Example 3.2.6. Given the votes in Table 3.2, the count of first preferences for labels a, b, c, and d would be 2, 2, 1, and 0 respectively. Since no label has a majority of first preferences, we add the seconds preferences. This gives a tally of 4, 4, 2, and 0 for labels a, b, c, and d respectively. Since two labels share the majority, we add the third preference from each of the votes. This gives tallies of 5, 4, 4, and 2, indicating a as the winner of the Bucklin Voting election. □

3.2.7 Coomb's Voting

Coomb's Voting is similar to Bucklin Voting in that it first attempts to select the label which has the majority of first preference votes. If no label has a majority, a separate election is held between the labels which are ranked last in the votes. The label with the most last-place votes is then removed from all votes. A tally of the first preference votes is taken again and this is repeated until there is a label with a majority.

Example 3.2.7. Again, we use the votes from Table 3.2 for this example. As with Buckling voting, the count of first preferences for labels a, b, c, and d would be 2, 2, 1, and 0 respectively. Looking at the label ranked last in each vote, we see that d occurs most frequently so d is removed from all votes. The tallies for first preference are reset and counted again. Since removing d does not change the tallies for first preferences, we hold another election to remove the last ranked candidate. Since c is now ranked last in 3 of the votes, it is removed from every vote. This bumps label b to the top in ranking B, alterting the number of first preference votes, leaving a with 2 and b with 3. Signifying b as the winner of the Coomb's Voting election.

3.2.8 Instant Runoff Voting

In Instant Runoff Voting (IRV), we again stop if there is a label with a majority of first place votes. If not, then the label with the fewest first preference votes is eliminated. This label is removed from each of the votes. For each vote where the eliminated label held a first place preference, the next preference votes are added to their respective label's tally. This is repeated until there is a candidate with a majority of votes. This voting scheme is similar to Coomb's Voting except instead of eliminating the label that is ranked last, we eliminate the label that has the fewest first preference votes.

Example 3.2.8. We use the votes from Table 3.2 for this example. The tallies of first preference votes for labels a, b, c, and d are again 2, 2, 1, and 0 respectively. Since there is no vote where d is the first preference, it is removed from all votes. This leave us with first preference tallies of 2, 2, 1 for a, b, and c respectively, without a majority winner. Of the remaining labels, c now has the fewest first preferences so is removed from all votes. This bumps b to the top of ranking B, altering the number of first preference votes to give a 2 and b 3, electing b as the winner of the Instant Runoff Voting.

3.2.9 Best Average Score

Among the data for the k nearest neighbour instances is the run time for each solver on that neighbour instance. The borda voting and distance weighted borda voting methods above do not take this valuable data into account when performing their aggregation. Another strategy to get a ranking of the solvers from this data is to order them by their average score across these k instances. This gives us an ordering of the solvers by their performance, averaged across the k nearest neighbours. We refer to this aggregation as ordering by Best Average Score.

3.2.10 Very Best Ranking

The Very Best Ranking (VBR) is the ranking produced by an oracle, who knows the best ranking for each instance by solver performance. We use this ranking as a benchmark to compare the rankings produced by the aggregation methods.

3.2.11 Summary of Voting Methods

Table 3.3 summarises the aggregate rankings produced by the voting methods described in previous sections. Since each voting method is an approximation of the Kemeny Consensus, it may differ in terms of the ranking, as can be seen with Borda and Bucklin voting. Additionally, Bucklin, Coomb's, and Instant Runoff voting each only elect a single winner but it may not necessarily be the highest-ranked label in the Kemeny Consensus ranking.

Table 3.3: Summary of aggregate rankings.

Voting Method	**Label Ranking**
Kemeny Consensus	$\langle b \prec a \prec c \prec d \rangle$
Borda Voting	$\langle a \prec b \prec c \prec d \rangle$
Copeland Voting	$\langle b \prec a \prec c \prec d \rangle$
Bucklin Voting	$\langle a \rangle$
Coomb's Voting	$\langle b \rangle$
Instant Runoff Voting	$\langle b \rangle$

3.3 An Exact Method for Computing Optimal Kemeny Rankings

The ranking methods presented in the previous section, with the exception of the optimal Kemeny and VBR rankings, are heuristics. In this section we compute an exact aggregate ranking by formulating the problem as a combinatorial optimisation problem. We present a Mixed Integer Programming (MIP) model for computing the optimal Kemeny ranking from a set of rankings (votes), in our case the k nearest neighbors of a query SAT instance. This model was implemented using Numberjack with SCIP[5] as the underlying MIP solver.

Let L be the set of all labels. Let V be the set of votes from each of the k nearest neighbors. We encode each ranking as a list where each label takes the value of the number of labels ranked higher than it. For example, if we are given a ranking of the labels $c \prec a \prec d \prec b$, then this would be converted to $\langle 1, 3, 0, 2 \rangle$ because a has 1 candidate ahead of it, b has 3, and so on. This simplifies the process of finding the index of a label within a ranking for the MIP model. We define the MIP model as follows:

- R is the array of the rank indexes in the Kemeny Consensus ranking. R_i states the number of labels that are ranked before label i in the aggregation ranking. The domain of values that each position in R can take is therefore $0 \ldots m-1$. This array contains all the decision variables.
- We add the constraint AllDiff(R) because only one candidate can occupy each position. There is no AllDifferent constraint in MIP but Numberjack allows us to encode the problem at a high level like this and it will automatically decompose the constraint into inequalities.
- For each pair of labels i and j, we have a Boolean variable r_{ij} which is encoded to take the value 1 if i is ranked higher than j in the target ranking R, zero otherwise.
- For each pair of labels i and j in each vote V_k we have a Boolean variable v_{kij} which takes the value 1 if label i is ranked higher than label j in vote V_k, zero otherwise.
- For each pair of labels i and j in each vote V_k we have the Boolean variable D_{kij} which is the exclusive-or between r_{ij} and v_{kij}. This means D_{kij} will

[5] SCIP website: `http://scip.zib.de/`

take the value 1 if R ranks i and j in a different order to V_k.

- The Kendal Tau distance to vote V_k from R is KT_k, which is the sum over all D_{kij} for every pair of candidates i and j.
- The Kemeny Score of the target ranking R is $\sum KT_k$. We attempt to minimise this value.

For a set of votes among 19 labels, this MIP model is able to solve the difficult aggregation problem in a matter of seconds. Consider that a greedy naive algorithm for commuting the Kemeny Optimal ranking may need to examine every possible permutation of the labels, which is $\mathcal{O}(n!)$. It must compute the Kemeny Score for each permutation and choose the ranking which minimises this function. This approach quickly becomes infeasible.

Model Improvements

This section presents some practical improvements to the MIP model by reducing the domain of the variables, upper-bounding the Kemeny Score, and linearising the exclusive-or constraint. These allow the model to scale to larger problems and solve instances more effectively.

Domain of variables in R. The domain of positions that each candidate can take can be bounded by the each candidates average position plus/minus the average KT-distance [20]. The average KT-distance, $\overline{d}$, is defined as:

$$\overline{d} = \frac{1}{|L|(|L|-1)} \cdot \sum_{l_1,l_2 \in L,\, l_1 \neq l_2} \text{KT-distance}(l_1, l_2)$$

Upper-bound on Kemeny Score. Each ranking in V could be chosen as the target ranking. Although it may not be the Kemeny Consensus ranking, it may be used to compute an upper-bound on the Kemeny Score. The Kemeny Score of each ranking in V can be computed against all rankings in V with the smallest value for this constituting an upper-bound on the Kemeny Score. This may be time consuming to compute if the number of votes is large as it takes n^2 calls to the KT-distance function.

Encoding Exclusive-or to MIP. Since we are limited to linear constraints in a MIP formulation, the exclusive-or constraint must be encoded to a linear form before it can be solved by the MIP solver. Let x be a Boolean variable which will take the value of the exclusive-or between two other Boolean variables a and b: $x = a \oplus b$. An auxiliary variable h will be introduced to replace the expression $2 \cdot a \cdot b$ and additional constraints will be imposed on h as follows:

$$\begin{aligned}
x &= a(1-b) + b(1-a) && (3.1)\\
x &= a + b - 2ab \\
x &= a + b - 2h && (3.2)\\
0 &\leq h \\
0 &\leq a - h \\
0 &\leq b - h \\
0 &\geq a + b - h - 1
\end{aligned}$$

Effectiveness. By bounding the domain of the variables by the candidates average position plus/minus the average KT-distance, we reduce the domain size of our decision variables by an average of 13.5%, 13.7%, and 1.2% for Industrial, Handcrafted, and Random instances respectively. This may not seem like much, particularly in the case of Random instances, but it is enough to reduce the domains from a complete overlap. When combined with the upper-bound on the Kemeny Score, it can be the difference between solving the instance in milliseconds or not at all.

3.4 Evaluation of a CBR-based Portfolio for SAT

We present an evaluation of both the quality of the adaptation strategies for ranking solvers by run time (Section 3.4.2), and the performance of our case-based reasoning-based algorithm portfolio for SAT (Section 3.4.3). We use the case-bases described in Section 3.1. In terms of the quality of the rankings, we show that rankings that consider running time, rather than relative position in the rank, give better performance. This is somewhat unsurprising, but it is interesting to see that the effort spent in finding the optimal Kemeny ranking is not worthwhile.

Of much greater significance is our demonstration that our CBR portfolio outperforms all of its constituent solvers by a considerable margin. In fact, the superiority of the CBR approach is observed regardless of the adaptation scheme used. Again, rankings that consider time are superior to all others, and compare well in terms of performance against the oracle (VBR) that always selects the best solver for a particular SAT instance.

3.4.1 Methodology

In all experiments we used a 10-fold cross validation approach, studying each of our three case-bases (Section 3.1) separately. We report averages, where appropriate. We always seek five nearest neighbours (5-NN), having observed that setting k to this value gave good typical-case performance. For the purpose of this chapter, unweighted normalised Euclidean distance is used as a similarity metric throughout. All adaption methods use the same distance measure, therefore each are tasked with aggregating the same set of neighbours.

3.4.2 Evaluation of the Adaptation Schemes

Given a ranking of the solvers, using a particular adaptation scheme, and their respective execution times, we can plot the cumulative time of the execution time of each solver against its position in the ranking. Let $s(i)$ be the solver ranked in position i of a ranking, and $t(\alpha)$ be the time taken by solver α to solve the instance. The plot of the cumulative time of the solvers in a ranking is given by:

$$f(x) = \sum_{i=1}^{x} t(s(i)).$$

If the solvers are ordered in strict order of increasing run time, the area under the curve in this plot will be minimised. On the other hand, the ranking which is as poor as possible will have maximum area. We compare each of our adaptation strategies that produce a ranking in this way. Figure 3.2 shows an example plot of the curve for each of the label rankings produced by an adaptation method.

Paired t-tests were performed to compare two label ranking methods on the basis of the area under the curves in our ranking plots. Such a paired t-test was performed between every pair of adaptation methods on every instance across the 10 splits in each dataset category. Table 3.4 gives the complete table of these results showing the 95% confidence interval and the p-value. In this table a confidence interval with negative lower and upper bounds, which is highlighted in

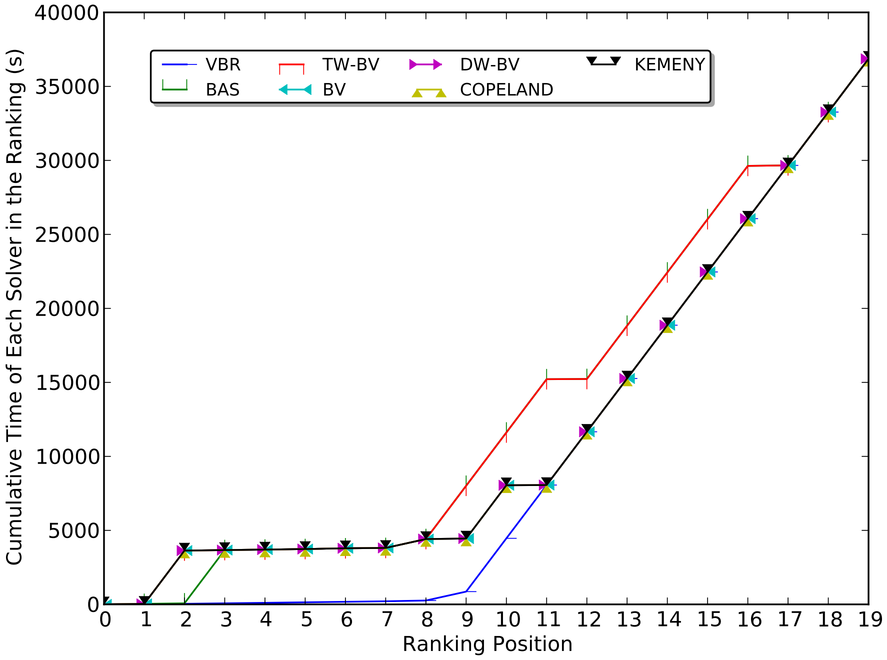


Figure 3.2: An illustration of the curve produced by accumulating the time taken by each solver in a ranking. Each line represents a curve produced by the rankings from each of the adaptation methods that produce a ranking.

bold, signifies that the ranking on the left is statistically significantly better than the ranking on the right.

On hand-crafted and industrial problems, which are really the most interesting from a practical viewpoint, the best-average-solver (BAS) and three variants of Borda voting out-perform all other methods; the statement is almost also true in the random category. It is clear, and not unsurprising, that the methods that take running time into account, out-perform all others. The Kemeny ranking never out-performs another method.

While comparing the rankings is interesting in itself, the more important question is how effective are these rankings in a CBR-based algorithm portfolio for SAT. We study this below.

3.4.3 A CBR-based Solver Portfolio for SAT

A variant of the CBR-based solver portfolio was implemented using each adaptation scheme in turn. We name these using the acronym of the adaptation scheme used. Given a SAT instance, the portfolio system will apply the relevant adaptation scheme to the set of cases retrieved using our 5-NN method. The highest ranked solver is selected. We record the total number of instances solved by the chosen solvers and the cumulative time summed over all solved instances, given a cut-off time of 1 hour per instance, in a 10-fold cross validation setting. This setup is very similar to that of the International SAT Competition.

Table 3.4: Table of paired t-tests comparing the area under the curve for each method. Results with a negative interval and p-value below the threshold of 0.05 are highlighted in **bold**. This means that the first method is statistically significantly better than the second method for that dataset. Results with a positive interval and p-value below the threshold of 0.05 are highlighted in *italics*.

		Handcoded			**Industrial**			**Random**		
		95% Conf. Int.		p-value	95% Conf. Int.		p-value	95% Conf. Int.		p-value
VBR	BAS	**-8615.4**	**-6366.5**	**0.000**	**-7186.6**	**-5286.6**	**0.000**	**-23538.6**	**-19300.2**	**0.000**
VBR	TW-BV	**-9425.4**	**-7119.7**	**0.000**	**-7857.2**	**-5864.8**	**0.000**	**-27900.0**	**-23160.4**	**0.000**
VBR	DW-BV	**-14736.7**	**-11755.3**	**0.000**	**-10613.7**	**-8388.4**	**0.000**	**-53803.9**	**-46149.2**	**0.000**
VBR	BV	**-14902.9**	**-11866.3**	**0.000**	**-11065.8**	**-8671.7**	**0.000**	**-53795.7**	**-46126.8**	**0.000**
VBR	COPELAND	**-17103.5**	**-13394.1**	**0.000**	**-12239.6**	**-9423.4**	**0.000**	**-55608.2**	**-46523.2**	**0.000**
VBR	KEMENY	**-17173.4**	**-13463.0**	**0.000**	**-11854.5**	**-9303.8**	**0.000**	**-70833.4**	**-61940.3**	**0.000**
BAS	TW-BV	**-1196.0**	**-367.2**	**0.000**	**-876.0**	**-372.8**	**0.000**	**-5223.4**	**-2998.3**	**0.000**
BAS	DW-BV	**-7054.4**	**-4455.7**	**0.000**	**-4141.7**	**-2387.2**	**0.000**	**-31689.3**	**-25425.0**	**0.000**
BAS	BV	**-7218.2**	**-4569.1**	**0.000**	**-4600.1**	**-2664.1**	**0.000**	**-31675.3**	**-25408.4**	**0.000**
BAS	COPELAND	**-9420.1**	**-6095.6**	**0.000**	**-5787.9**	**-3401.8**	**0.000**	**-33551.5**	**-25741.1**	**0.000**
BAS	KEMENY	**-9499.7**	**-6154.9**	**0.000**	**-5396.9**	**-3288.2**	**0.000**	**-48835.7**	**-41099.2**	**0.000**
TW-BV	DW-BV	**-6203.8**	**-3743.0**	**0.000**	**-3514.4**	**-1765.7**	**0.000**	**-27354.7**	**-21538.0**	**0.000**
TW-BV	BV	**-6368.2**	**-3855.9**	**0.000**	**-3965.6**	**-2049.9**	**0.000**	**-27347.5**	**-21514.5**	**0.000**
TW-BV	COPELAND	**-8552.0**	**-5400.5**	**0.000**	**-5143.7**	**-2797.1**	**0.000**	**-29182.7**	**-21888.2**	**0.000**
TW-BV	KEMENY	**-8630.7**	**-5460.7**	**0.000**	**-4761.6**	**-2674.6**	**0.000**	**-44543.8**	**-37169.4**	**0.000**
DW-BV	BV	**-260.6**	**-16.7**	**0.026**	**-558.1**	**-177.3**	**0.000**	-136.7	167.2	0.844
DW-BV	COPELAND	**-2639.3**	**-1366.3**	**0.000**	**-1839.8**	**-821.0**	**0.000**	-2622.9	444.6	0.164
DW-BV	KEMENY	**-2719.4**	**-1425.0**	**0.000**	**-1595.1**	**-561.1**	**0.000**	**-18057.6**	**-14763.0**	**0.000**
BV	COPELAND	**-2459.8**	**-1268.6**	**0.000**	**-1361.5**	**-564.0**	**0.000**	-2625.3	416.4	0.155
BV	KEMENY	**-2543.1**	**-1324.2**	**0.000**	**-1251.7**	**-169.1**	**0.010**	**-18060.9**	**-14790.3**	**0.000**
COPELAND	KEMENY	-222.4	83.6	0.374	-339.2	843.8	0.403	**-16663.1**	**-13979.1**	**0.000**

Table 3.5: Leader board for the Handcrafted category of problem instances.

	Solver Name	Nr. Solved	Cumulative Time on Solved Instances (s)
1	VBR	114.9 (± 1.4)	24703.4 (± 4972.5)
2	TW-BV	110.5 (± 2.2)	25668.9 (± 5855.8)
3	BAS	109.5 (± 2.7)	26147.9 (± 7106.6)
4	DW-BV	109.3 (± 1.8)	26239.2 (± 6779.6)
5	BV	109.2 (± 1.8)	25561.7 (± 6541.9)
6	IRV	108.0 (± 1.8)	23872.8 (± 6121.4)
7	COOMBS	107.9 (± 2.5)	24553.9 (± 6317.8)
8	KEMENY	107.8 (± 2.4)	24163.2 (± 6683.3)
9	BUCKLIN	107.6 (± 2.5)	22892.7 (± 6783.0)
10	COPELAND	107.6 (± 2.5)	24060.7 (± 6140.9)
11	minisat20SAT07	88.1 (± 4.1)	33700.5 (± 9455.9)
12	mxc08	86.2 (± 3.9)	30312.6 (± 9620.1)
13	march_dl2004	85.1 (± 3.5)	22245.3 (± 7770.6)
14	picosat846	84.0 (± 3.6)	28713.1 (± 8167.2)
15	minisat2.0	82.5 (± 4.3)	32019.3 (± 7527.4)
16	vallst	73.1 (± 5.2)	18904.4 (± 4995.6)
17	rsat20	67.0 (± 4.4)	20652.4 (± 5934.4)
18	zchaff_rand	62.6 (± 4.5)	17362.4 (± 6541.4)
19	march_pl	44.2 (± 4.8)	6817.7 (± 2814.7)
20	tts	43.1 (± 5.3)	6360.1 (± 3461.8)
21	SATenstein_r3sat	25.7 (± 4.7)	4278.6 (± 5091.6)
22	ranov	25.5 (± 5.6)	3789.9 (± 3098.3)
23	adaptg2wsatplus	24.3 (± 5.0)	1572.6 (± 1804.6)
24	gnoveltyplus	24.3 (± 4.6)	3433.5 (± 2446.2)
25	adaptg2wsat0	23.4 (± 4.7)	1987.7 (± 2177.6)
26	kcnfs04SAT07	22.5 (± 4.2)	10022.1 (± 3416.3)
27	SATenstein_qcp	22.2 (± 3.8)	2347.4 (± 2197.6)
28	SATenstein_swgcp	22.2 (± 3.8)	2347.8 (± 2196.6)
29	SATenstein_hgen	18.4 (± 3.8)	2179.7 (± 1570.3)

We compare this to the Very Best Ranking (VBR), which chooses the best solver for the instance given. We report the average number of instances solved and the average run time, with standard deviation in both cases. In our results tables, Tables 3.5, 3.6 and 3.7, we sort the variants in terms of number of instances solved, and then by run time. The VBR, the oracle, is therefore always ranked at the top.

The overall result here is that the best SAT solvers are out-performed in every problem class by each of the CBR-based portfolios. The CBR portfolio compares very well against the oracle (VBR) in each category. For example, in the random

Table 3.6: Leader board for the Industrial category of problem instances.

	Solver Name	**Nr. Solved**	**Cumulative Time on Solved Instances (s)**
1	VBR	113.1 (± 2.5)	24561.0 (± 5164.6)
2	BAS	110.3 (± 3.3)	30003.9 (± 4674.8)
3	TW-BV	109.8 (± 3.0)	27742.0 (± 4430.4)
4	KEMENY	105.4 (± 3.7)	26500.6 (± 5287.4)
5	DW-BV	105.1 (± 3.4)	25699.3 (± 4611.6)
6	BV	105.0 (± 3.5)	26364.8 (± 4243.5)
7	COPELAND	104.5 (± 4.1)	26650.9 (± 5209.8)
8	COOMBS	104.2 (± 3.9)	26758.2 (± 6411.9)
9	IRV	103.6 (± 3.7)	26317.2 (± 5540.3)
10	BUCKLIN	102.6 (± 4.5)	25574.9 (± 5617.2)
11	mxc08	101.8 (± 3.9)	30144.6 (± 6091.5)
12	picosat846	96.4 (± 3.7)	29688.2 (± 6551.8)
13	rsat20	93.8 (± 4.8)	34573.7 (± 6666.6)
14	minisat20SAT07	89.8 (± 3.2)	31467.8 (± 8382.5)
15	minisat2.0	87.2 (± 2.4)	34332.8 (± 7641.0)
16	zchaff_rand	80.6 (± 5.3)	33037.2 (± 4045.4)
17	vallst	65.9 (± 4.8)	29617.0 (± 3576.8)
18	march_dl2004	43.3 (± 4.4)	6027.4 (± 1751.3)
19	adaptg2wsatplus	21.3 (± 5.3)	3492.1 (± 1987.2)
20	march_pl	21.2 (± 3.6)	3210.8 (± 2758.8)
21	SATenstein_r3sat	21.1 (± 4.0)	2816.8 (± 1925.5)
22	tts	20.2 (± 3.6)	2342.4 (± 2489.1)
23	adaptg2wsat0	20.1 (± 4.7)	2854.1 (± 2210.5)
24	SATenstein_swgcp	16.8 (± 4.4)	2547.4 (± 2406.1)
25	SATenstein_qcp	16.8 (± 4.4)	2552.4 (± 2412.5)
26	SATenstein_hgen	15.4 (± 4.0)	2576.1 (± 2164.6)
27	gnoveltyplus	14.3 (± 4.2)	3507.7 (± 2883.7)
28	kcnfs04SAT07	9.1 (± 3.0)	1026.4 (± 1441.9)
29	ranov	8.7 (± 3.0)	1672.0 (± 1517.6)

problem category, the CBR portfolio solves 33% more instances than the best SAT solver on its own, and solves within 5% of the instances solved by the oracle.

Both BAS and TW-BV portfolios perform consistently well, which would not be obvious a-priori in this setting in which it is most important to solve instances within a cut-off. Once again, the Kemeny ranking is not competitive amongst the CBR-based portfolios.

Notably, in all categories, the cumulative time on solved instances is lower for all portfolio approaches than the single best solver, despite solving many more

Table 3.7: Leader board for the Random category of problem instances.

	Solver Name	**Nr. Solved**	**Cumulative Time on Solved Instances (s)**
1	VBR	227.6 (± 1.4)	28960.0 (± 5745.6)
2	BAS	216.7 (± 3.2)	30463.4 (± 5284.6)
3	TW-BV	211.8 (± 2.6)	25250.1 (± 4005.0)
4	COOMBS	206.2 (± 3.1)	24303.7 (± 4554.8)
5	IRV	206.1 (± 3.9)	25405.7 (± 5249.5)
6	COPELAND	205.8 (± 3.1)	24590.6 (± 5769.9)
7	DW-BV	205.6 (± 3.5)	24019.9 (± 4382.9)
8	BV	205.4 (± 3.5)	23753.5 (± 4606.5)
9	BUCKLIN	203.2 (± 3.9)	24111.5 (± 5774.5)
10	KEMENY	194.0 (± 4.5)	27713.1 (± 4651.7)
11	march_dl2004	149.8 (± 6.3)	38318.5 (± 4934.1)
12	gnoveltyplus	148.1 (± 6.0)	21884.0 (± 6398.8)
13	SATenstein_T7	146.8 (± 6.4)	23124.2 (± 3713.5)
14	ranov	146.0 (± 6.4)	19454.8 (± 4909.5)
15	SATenstein_swgcp	142.7 (± 7.5)	16795.1 (± 5327.9)
16	SATenstein_qcp	142.7 (± 7.5)	16801.1 (± 5325.8)
17	SATenstein_2P8	141.8 (± 6.6)	15242.8 (± 4268.1)
18	SATenstein_L5	141.6 (± 6.0)	9293.6 (± 3807.0)
19	SATenstein_T3	141.6 (± 7.0)	12863.9 (± 4687.7)
20	adaptg2wsat0	139.8 (± 6.7)	23592.5 (± 6028.6)
21	SATenstein_2P7	139.5 (± 6.9)	12376.5 (± 3575.3)
22	SATenstein_2P9	138.4 (± 7.5)	12624.7 (± 4294.2)
23	adaptg2wsatplus	138.4 (± 6.4)	26437.6 (± 10217.9)
24	SATenstein_r3sat	138.1 (± 6.5)	12813.8 (± 3148.8)
25	SATenstein_T5	137.9 (± 6.9)	16251.8 (± 4443.1)
26	SATenstein_hgen	121.5 (± 8.8)	9396.6 (± 1917.0)
27	mxc08	115.4 (± 6.7)	40052.7 (± 6010.6)
28	SATenstein_L3	115.2 (± 6.6)	11852.6 (± 2615.7)
29	minisat2.0	115.1 (± 6.2)	38166.2 (± 6136.8)
30	minisat20SAT07	114.0 (± 7.3)	36482.2 (± 10084.8)
31	picosat846	112.6 (± 7.8)	35309.7 (± 5667.0)
32	march_pl	89.1 (± 4.4)	11450.7 (± 3655.8)
33	rsat20	78.4 (± 7.6)	19388.4 (± 9251.8)
34	kcnfs04SAT07	75.8 (± 4.8)	20189.9 (± 5584.9)
35	vallst	74.9 (± 6.8)	31666.5 (± 6851.4)
36	zchaff_rand	62.1 (± 6.3)	18108.5 (± 4398.4)
37	tts	25.1 (± 4.5)	5520.7 (± 3062.5)

instances. This demonstrates the significant performance gains that may be achieved by employing a portfolio of solvers that are potentially solver on average than the single best solver.

These results are consistent with the expectations of experts in the field of SAT. It is regarded as a challenge to be able to select a good performing solver for a given instance, and the choice is heavily reliant on the experience of the user who makes this choice. Therefore, this domain is perfect for CBR, and the results demonstrate that it is also a very useful technique to use here.

3.5 Chapter Summary

In this chapter we studied a variety of adaptation schemes for a family of CBR-based algorithm portfolios for the SAT problem. The results clearly demonstrate that the choice of adaptation scheme is important for performance with schemes that consider run time rather than relative ranking giving superior performance.

We clearly demonstrated that a CBR approach to this task is extremely competitive, and out-performs individual high-performing SAT solvers in a wide variety of disciplines. A feature of the domain of SAT, and constraint solving in general, is that experience is important.

Chapter 4

A Hierarchical Portfolio of Representations and Solvers

Summary. *A hierarchical portfolio is outlined which, in addition to a portfolio of CSP solvers, considers a portfolio of SAT encodings and subsequently a portfolio of SAT solvers. We demonstrate the complementary nature of such a portfolio and ultimately its superior performance to that of a portfolio based on a single representation.*

Additionally, we apply the portfolio to the domain of graphical models, demonstrating that significant performance gains can also be achieved in this domain.

4.1 Introduction

The pace of development in both CSP and SAT solver technology has been rapid. Combined with portfolio and algorithm selection technology, impressive performance improvements over systems that have been developed only a few years previously have been demonstrated. Constraint satisfaction problems and satisfiability problems are both NP-complete and, therefore, there exist polynomial-time transformations between them. We can leverage this fact, whereby in addition to a set of CSP solvers, we may choose to convert CSPs into SAT problems and solve them using SAT solvers.

In this chapter we exploit the fact that different SAT solvers have different performances on different encodings of the same CSP. In fact, the particular choice of encoding that will give good performance with a particular SAT solver is dependent on the problem instance to be solved. We show that, in addition to using dedicated CSP solvers, to achieve the best performance for solving a CSP the best course of action might be to translate it to SAT and solve it using a SAT solver. We name our approach Proteus, after the Greek god Proteus, the shape-shifting water deity that can foretell the future.

Our approach makes a series of decisions – whether a problem should be solved as a CSP or a SAT problem, which encoding should be used for converting into SAT, and finally which solver should be assigned to tackle the problem. Approaches that make a series of decisions are usually referred to as hierarchical models. Hierarchical models have also been used in the context of a SAT portfolio [73, 163]. They first predict whether the problem to be solved is expected to be satisfiable or not and then choose a solver depending on that decision. Our approach is closer to [62], which first predicts what level of consistency the all-different constraint should achieve before deciding on its implementation.

To the best of our knowledge, no portfolio approach that potentially transforms the representation of a problem in order to be able to solve it more efficiently exists at present.

The remainder of this chapter is organised as follows. Section 4.2 presents some background material and discusses some related work. In Section 4.3 we motivate the need to choose the representation and solver in combination. Details of the hierarchical portfolio is presented in Section 4.4. A detailed empirical evaluation of the portfolio is presented in Section 4.5. Section 4.6 undertakes some analysis comparing the empirical performances of CSP versus SAT. Proteus is extended and applied to the domain of graphical models in Section 4.7. Finally, a summary of the chapter is presented in Section 4.8.

4.2 Background and Related Work

This section provides some background and discusses some work related to this chapter. First, we discuss some related portfolio approaches, followed by some related approaches that have used SAT to solve CSPs, and finally, we describe some details of the randomly generated benchmarks that will be used for the initial investigation.

4.2.1 Related Portfolios

The approach presented in this chapter employs algorithm selection techniques to dynamically choose whether to translate to SAT, and if so, which SAT encoding and solver to use, otherwise it selects which CSP solver to use. There has been a great deal of research in the area of algorithm selection and portfolios; we refer the reader to Section 2.6 and to a recent survey of this work [106].

Specifically, we note three contrasting example approaches to algorithm selection for the constraint satisfaction and satisfiability problems, namely CPhydra (CSP), SATzilla (SAT), and isac (SAT). CPhydra [128] contains an algorithm portfolio of CSP solvers which partitions CPU time between components of the portfolio in order to maximize the probability of solving a given problem instance within a fixed time limit. SATzilla [165], at its core, uses cost-sensitive decision forests that vote on the SAT solver to use for an instance. In addition to that, it contains a number of practical optimisations, for example running a pre-solver to quickly solve the easy instances. isac [98] is a cluster-based approach that groups instances based on their features and then finds the best solver for each cluster.

The Proteus approach is not a straightforward application of portfolio techniques. In particular, there is a series of decisions to make that affect not only the solvers that will be available, but also the information that can be used to make the decision. Because of this, the different choices of conversions, encodings and solvers cannot simply be seen as different algorithms or different configurations of the same algorithm.

4.2.2 Solving a CSP using SAT

The approach presented in this chapter offers a novel perspective on using SAT solvers for constraint solving. The idea of solving CSPs as SAT instances is not new; the solvers `Sugar`, `Azucar`, and `CSP2SAT4J` are three examples of SAT-based CSP solving which have been successful.

`Sugar` [152] has been very competitive in recent CSP solver competitions. It converts the CSP to SAT using a specific encoding, known as the order encoding, which was presented in detail in Section 2.5.4. `Azucar` [153] is a related SAT-based CSP solver that uses the compact order encoding. However, both `Sugar` and `Azucar` use a single predefined encoding and solver to solve the encoded CSP instances. Our work does not assume that conversion using a specific encoding to SAT is the best way of solving a problem, but considers multiple candidate encodings and solvers.

CSP2SAT4J [109] uses the SAT4J library as its SAT back-end and a set of static rules to choose either the direct or the support encoding for each constraint. For intensional and extensional binary constraints that specify the supports, it uses the support encoding. For all other constraints, it uses the direct encoding. The Proteus approach does not have predefined rules but instead chooses the encoding and solver based on features of the problem instance to solve.

4.2.3 Uniform Random Binary CSPs

This chapter will employ uniform random binary (URB) CSPs [59] in order to study the performance of a SAT solver on a number of encodings from CSP to SAT. Artificially generated random benchmarks have commonly been used to benchmark the development of new algorithms and solvers. Although the generated instances lack some of the structure inherent in industrial instances, they provide a controlled manner with which a sufficient number of benchmarks can be constructed in order to make meaningful observations. Large numbers of instances can be generated with similar characteristics such as domain size, constraint tightness and density, and so on. Such control enables researchers to study the behaviour of approaches in various conditions.

A uniform random binary CSP is described by a tuple $\langle n, d, c, t\rangle$, representing the number of variables, the uniform domain size, the density of the constraint graph, and the constraint tightness respectively. Intuitively, the number of variables and their domain size correspond to the size of the problem. The constraint density specifies the ratio of possible constraint pairs, i.e. $|C| = \lceil c \cdot \binom{n}{2} \rceil$. The tightness corresponds to the fraction of disallowed value-pairs for each constraint, i.e. the number of forbidden tuples for each constraint is $t \cdot d^2$.

A number of methods can be used for determining how the random selection of constraint and value tuples are chosen, namely models A, B, C, D, and E [6, 68]. Each differs subtly in how the edges and constraints are selected, such as selecting each of the $n(n-1)/2$ possible edges independently with probability c, or selecting exactly $c \cdot n(n-1)/2$ edges. This chapter will use the model B method, which selects exactly $c \cdot n(n-1)/2$ constraints, and exactly $t \cdot d^2$ forbidden pairs of incompatible values. This model has been widely used in the constraint programming literature as it enables problems to be generated with a precise and consistent size, allowing empirical performance to be studied in a highly controlled manner [68].

4.3 Multiple Encodings and Solvers

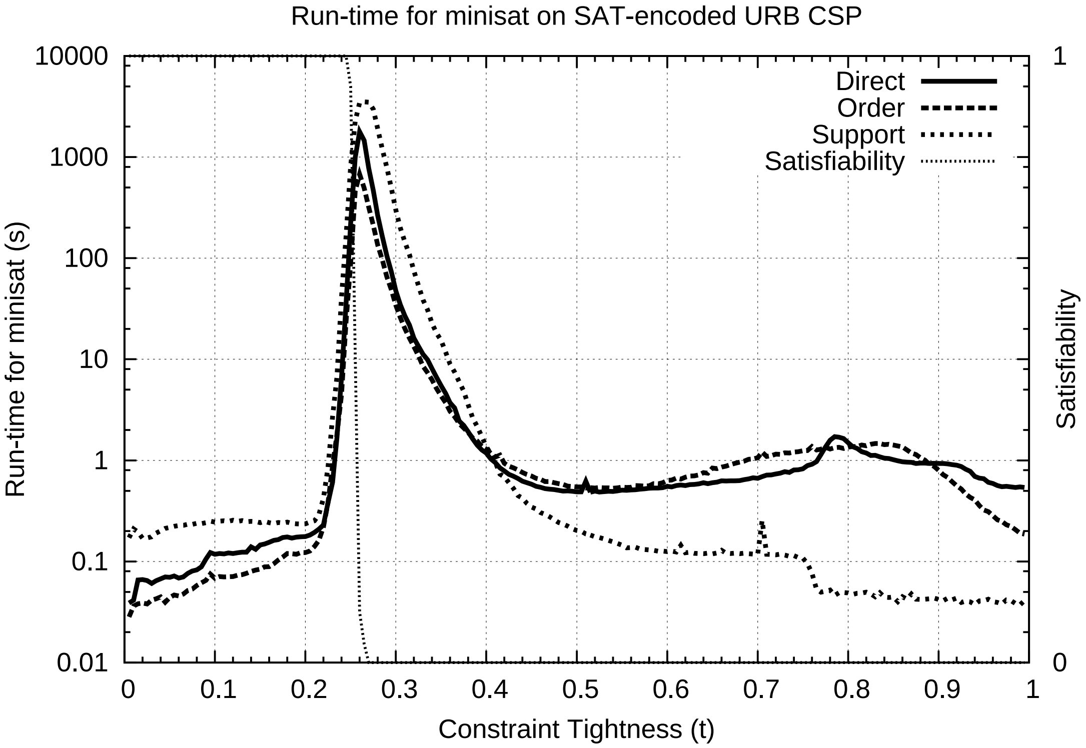
This section analyses the performance of multiple encodings and solvers in two contrasting scenarios. Firstly, a controlled experimental evaluation using uniform random instances studies the performance behaviour across the phase transition. A subsequent evaluation employs structured CSP Competition instances, demonstrating how the additional encodings and solvers can be exploited to achieve significant performance gains.

4.3.1 Uniform Random Instances

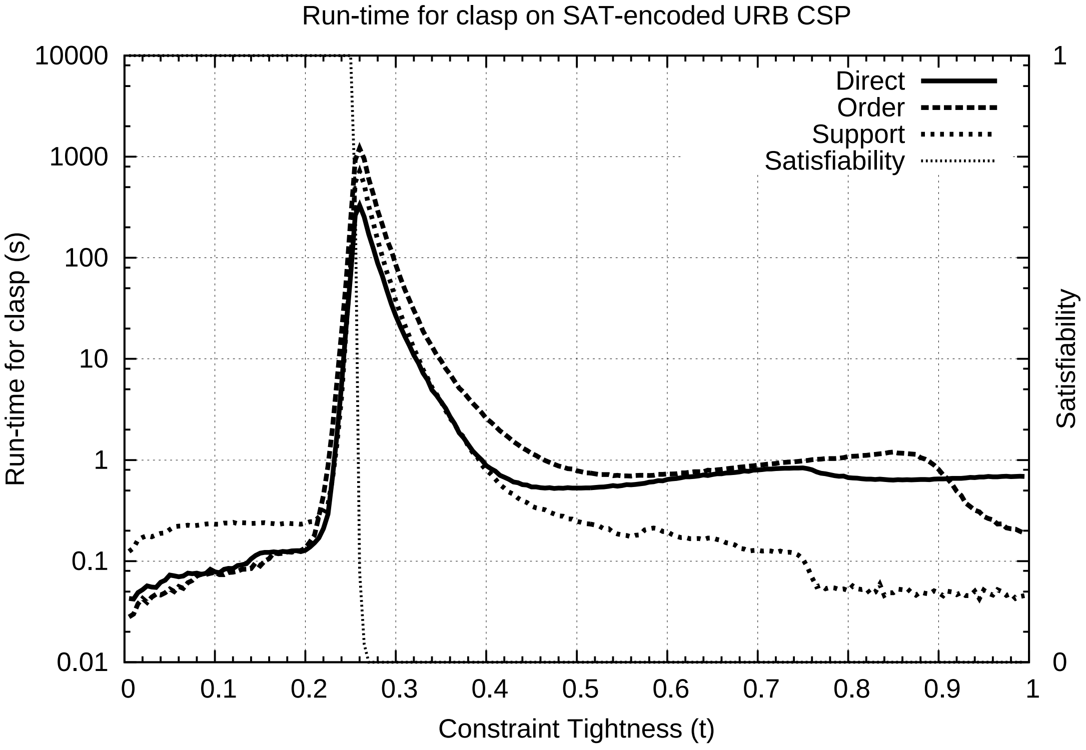
To motivate our work, we performed a detailed investigation for two solvers to assess the relationship between solver and problem encoding with features of the problem to be solved. For this experiment we considered uniform random binary CSPs with a fixed number of variables, domain size and number of constraints, and varied the constraint tightness. Tightness was varied from 0 to 1, where 0 means that all assignments are allowed, in increments of 0.005. At each tightness the mean runtime of the solver on 100 random CSP instances is reported. Each instance contains 30 variables with domain size 20 and 300 constraints, resulting in instances with a constraint density of 0.69. This allows us to study the performance of SAT encodings and solvers across the phase transition.

Figure 4.1 plots the runtime for `MiniSat` and `Clasp` on uniformly random binary CSPs that have been translated to SAT using three different encodings. Observe that in Figure 4.1(a) there is a distinct difference in the performance of `MiniSat` on each of the encodings, sometimes an order of magnitude. Before the phase transition, we see that the order encoding achieves the best performance and maintains this until the phase transition. Beginning at constraint tightness 0.41, the order encoding gradually starts achieving poorer performance and the support encoding now achieves the best performance.

Notably, if we rank the encodings based on their performance, the ranking changes after the phase transition. This illustrates that there is not just a single encoding that will perform best overall and that the choice of encoding matters, but also that



(a) Performance using `MiniSat`.



(b) Performance using `Clasp`.

Figure 4.1: `MiniSat` and `Clasp` on random binary CSPs.

this choice is dependent on problem characteristics such as constraint tightness.

Around the phase transition, we observe contrasting performance for `Clasp`, as illustrated in Figure 4.1(b). Using `Clasp`, the ranking of encodings around the phase transition is direct $\succ$ support $\succ$ order; whereas for `MiniSat` the ranking is order $\succ$ direct $\succ$ support. Note also that the peaks at the phase transition differ in magnitude between the two solvers. These differences underline the importance of the choice of solver, in particular in conjunction with the choice of encoding – making the two choices in isolation does not consider the interdependencies that affect performance in practice. Thus, in the context of a portfolio it is not sufficient to consider the choice of solver or problem representation independently, instead they should be considered in tandem.

4.3.2 CSP Competition Instances

In addition to the random CSP instances, our analysis also comprises 1493 challenging benchmark problem instances from the CSP solver competitions that involve global and intensional constraints. The empirical setup is described in detail in Section 4.5. Figure 4.2 illustrates the respective performance of the virtual best CSP-based and SAT-based methods on these instances. The virtual best CSP method relates to the best performance of any of the considered CSP solvers. Likewise, the virtual best SAT method equates to the absolute best performance amongst any of the encoding and solver combinations. The figure's axes show runtime (in seconds) to solve the instance, in log-scale. Each point of the scatter plot represents the time in seconds of the two approaches. A point below the dashed line indicates that the virtual best SAT portfolio was quicker, whereas a point above means the virtual best CSP portfolio was quicker. Equivalent performance would place the point along the dashed line.

Unsurprisingly the dedicated CSP methods often achieve the best performance. There are, however, numerous cases where considering SAT-based methods has the potential to yield significant performance improvements. In particular, there are a number of instances that are unsolved by any CSP solver but can be solved quickly using SAT-based methods, illustrated by the points along the right hand edge. Conversely, there are many instances unsolved by the SAT methods which are solved quickly by a CSP solver, as seen by the points along the top edge of the plot. Moreover, in cases where both paradigms are able to solve the instance, often one approach can be orders of magnitude faster than the other.

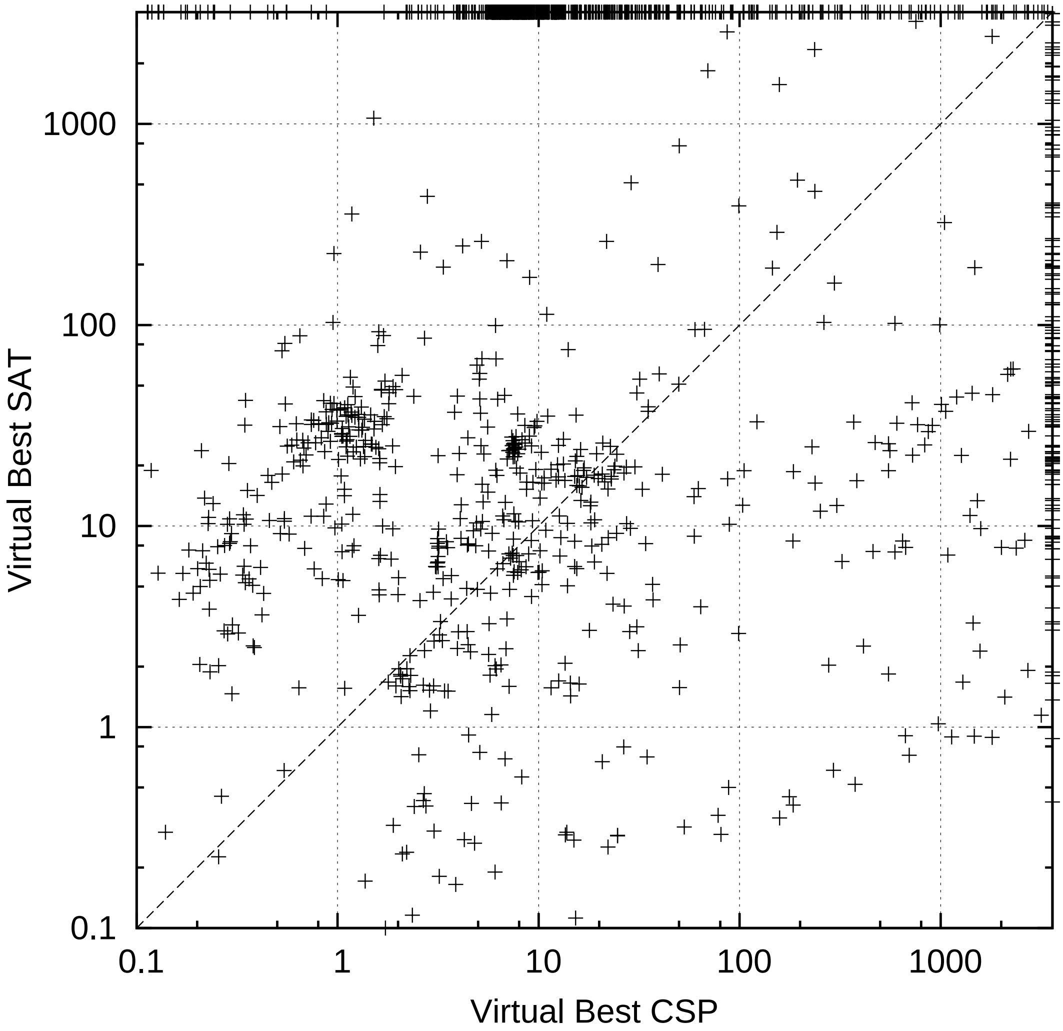


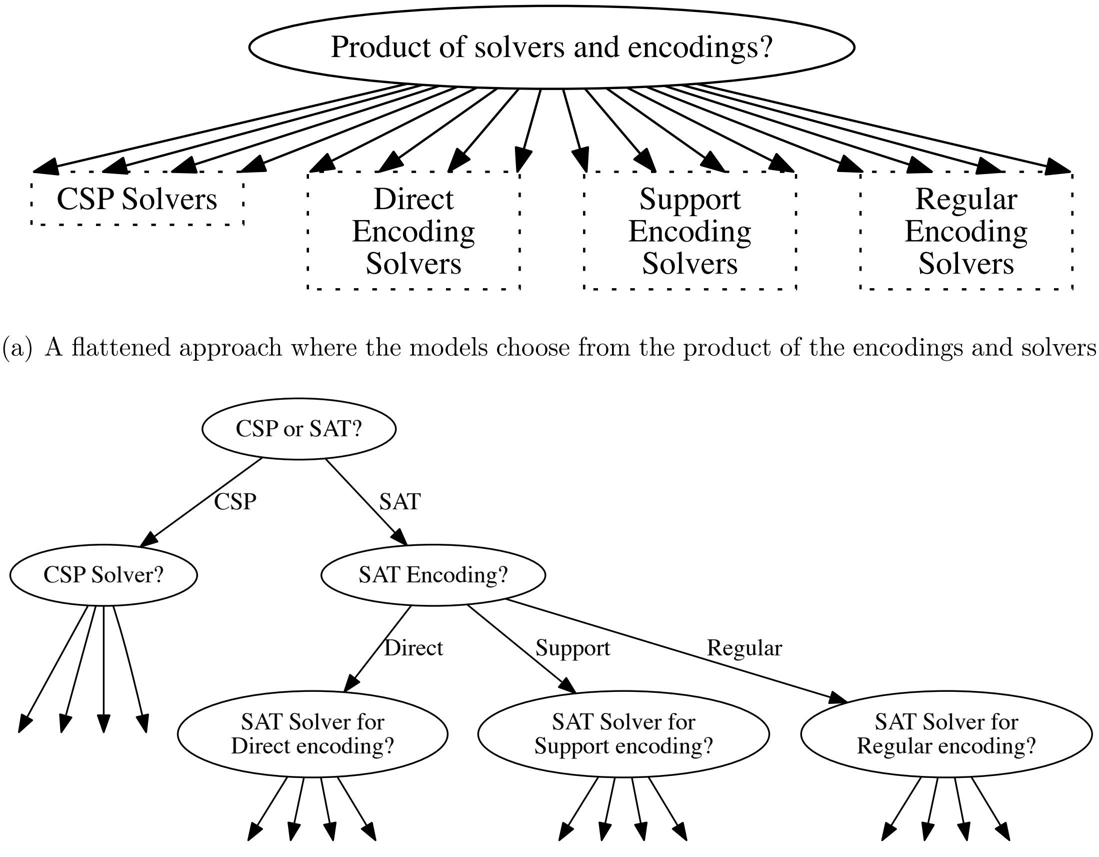
Figure 4.2: Performance of the virtual best CSP portfolio and the virtual best SAT-based portfolio.

Clearly the two approaches are complementary: there are numerous instances for which a SAT-based approach does not perform well or fails to solve the instance but a CSP solver does extremely well, and vice-versa. Additionally, both paradigms boast a number of instances that can be solved orders of magnitude faster, showing their highly complementary nature. The hierarchical portfolio presented in the following sections, named Proteus, aims to unify the best of both worlds and take advantage of the substantial potential performance gains that can be achieved.

4.4 Proteus: A Hierarchical Portfolio for CSPs

In order to exploit the potential performance gains presented in the previous section, a portfolio could be built using multiple CSP solvers, multiple SAT encodings, and multiple SAT solvers. Such a portfolio could be modelled as a flat decision by taking the CSP solvers along with the Cartesian product of the SAT encodings and SAT solvers, as depicted in Figure 4.3(a). One disadvantage of this approach however is that there may be a large number of choices for the portfolio to choose from. This can hinder some machine learning models which encounter difficulties such as the inability to generalise or make meaningful deductions when the number of outcomes is large [121]. Subsequently, a portfolio based on this flattened approach will not scale if the set of representations, encodings, or solvers is extended.

Conversely, our approach, proposed under the name Proteus, makes a series of decisions – first choosing a representation of the problem such as whether a problem should be solved as a CSP or a SAT problem, subsequently which encoding should be used, and finally which solver should be assigned to tackle the



(a) A flattened approach where the models choose from the product of the encodings and solvers.

(b) The hierarchy makes a series of decisions: first which paradigm, secondly which encoding, and finally which solver.

Figure 4.3: A contrasting example of a flattened versus hierarchical approach to selecting from a set of four CSP solvers, three SAT encodings, and four SAT solvers.

problem. An example of this hierarchy is depicted in Figure 4.3(b).

This hierarchical approach offers a number of advantages over the flattened setting. Even though there are a larger number of decisions to be made, the set of choices at each is more reasonable. Each decision is easily extensible with additional paradigms, solvers, encodings, feature sets, and machine learning models. Moreover, we may decide to use different machine learning models for each of the decisions. Since some decisions may be binary, we might choose a model which performs well on binary decisions. Furthermore, different feature sets may be used for each decision, possibly providing further insight into the problem. For example, for choosing the SAT solver to run under the direct encoding we may compute features of the SAT instance [165] in order to make a better decision.

Caution must be taken however, since machine learning models can be liable to making imperfect decisions, which could in turn be multiplied in the Proteus hierarchy. However, in practice this did not occur in our empirical evaluations, rather the hierarchical model was more reliable than any flattened approach, as will be demonstrated in Table 4.2.

4.5 Experimental Evaluation

4.5.1 Setup

The hierarchical model we present in this chapter consists of a number of layers to determine how the instance should be solved. At the top level, we decide whether to solve the instance using as a CSP or using a SAT-based method. If we choose to leave the problem as a CSP, then one of the dedicated CSP solvers must be chosen. Otherwise, we must choose the SAT encoding to apply, followed by the choice of SAT solver to run on the SAT-encoded instance.

Each decision of the hierarchical approach aims to choose the direction which has the potential to achieve the best performance in that sub-tree. For example, for the decision to choose whether to solve the instance using a SAT-based method or not, we choose the SAT-based direction if there is a SAT solver and encoding that will perform faster than any CSP solver would. Whether this particular encoding-solver combination will be selected subsequently depends on the performance of the algorithm selection models used in that sub-tree of our decision mechanism. For regression models, the training data is the best performance of any solver under that branch of the tree. For classification models, it is the label of the sub-branch with the virtual best performance.

This hierarchical approach presents the opportunity to employ different decision mechanisms at each level. We consider 6 regression, 19 classification, and 3 clustering algorithms, which are listed below. For each of these algorithms, we evaluate the performance using stratified 10-fold cross-validation. The dataset is split into 10 equally sized partitions with approximately the same distribution of the best solvers. One partition is used for testing and the remaining 9 partitions as the training data for the model. This process is repeated with a different partition considered for testing each time until every partition has been used for testing. This enables the entire dataset to be considered in the evaluation while maintaining a consistent training and testing isolation.

Solver performance is measured in terms of the PAR10 score (penalised average runtime). The PAR10 score for an instance is the time it takes the solver to solve the instance, unless the solver times out. In this case, the PAR10 score is ten times the timeout value. The sum over all instances is divided by the number of instances. In essence, the PAR10 measure quantifies the coefficients in our overall multi-objective problem, whereby we first want to minimise the number of unsolved instances, and subsequently the overall CPU-time. It offers a more holistic and discriminative measure than just CPU-time or number of instances solved alone. The factor of ten for unsolved instances is rather arbitrary but has typically been the standard in the portfolio and related communities for a number of years.

Instances. In our evaluation, we consider CSP problem instances from the CSP solver competitions [1]. Of these, we consider all instances defined using global and intensional constraints that are not trivially solved during 2 seconds of feature computation. We also exclude all instances that were not solved by any CSP or SAT solver within the time limit of 1 hour. Altogether, we obtain 1,493 non-trivial instances from problem classes such as Timetabling, Frequency Assignment, Job-Shop, Open-Shop, Quasi-group, Costas Array, Golomb Ruler, Latin Square, All Interval Series, Balanced Incomplete Block Design, and many others. This set includes both small and large arity constraints and all of the global constraints used during the CSP solver competitions: all-different, element, weighted sum, and cumulative. Note that the benchmark set is comprised entirely of satisfaction instances.

The optimisation problems mentioned above, such as the Golomb ruler problem, were modelled in the CSP Competitions as a satisfaction problem with an objective bound as a constraint.

For the SAT-based approaches, Numberjack [78] was used to translate a CSP instance specified in XCSP format [141] into SAT (CNF) and pass the resulting instance to the SAT solver.

Features. A fundamental requirement of any machine learning algorithm is a set of representative features. We explore a number of different feature sets to train our models: *i*) features of the original CSP instance, *ii*) features of the direct-encoded SAT instance, *iii*) features of the support-encoded SAT instance, *iv*) features of the direct-order-encoded SAT instance and *v*) a combination of all four feature sets. These features are described in further detail below.

We computed the 36 features used in CPHydra for each CSP instance using Mistral, listed below. The set includes static features like statistics about the types of constraints used, average and maximum domain size; and dynamic statistics recorded by running Mistral for 2 seconds: average and standard deviation of variable weights, number of nodes, number of propagations and a few others. Instances which are solved by Mistral during feature computation are filtered out from the dataset.

- Average Domain Continuity,
- Average Predicate Arity,
- Average Predicate Shape,
- Average Predicate Size,
- Dynamic - Log Average Constraint Weight,
- Dynamic - Log Number of Nodes Visited,
- Dynamic - Log Number of Propagations,
- Dynamic - Log Std. Dev. Constraint Weight,
- Log Number of Bits,
- Log Number of Booleans,
- Log Number of Constants,
- Log Number of Constraints,
- Log Number of Extra Bits,
- Log Number of Extra Booleans,
- Log Number of Extra Ranges,
- Log Number of Extra Values,
- Log Number of Lists,
- Log Number of Ranges,
- Log Number of Search Variables,
- Log Number of Values,
- Maximum Arity,
- Minimum Continuity,
- Number of All-Differents,
- Perc. of Constraints - AllDiff,
- Perc. of Constraints - Binary Extensional,

- Perc. of Constraints - Cumulative,
- Perc. of Constraints - Dec. Predicate,
- Perc. of Constraints - Element,
- Perc. of Constraints - Extensional,
- Perc. of Constraints - GAC Predicate,
- Perc. of Constraints - Global,
- Perc. of Constraints - Large Extensional,
- Perc. of Constraints - N-ary Extensional,
- Perc. of Constraints - Weighted Sum,
- Sqrt. Average Domain Size, and
- Sqrt. Maximum Domain Size

In addition to the CSP features, we computed the 54 SAT features used by SATzilla [165] for each of the encoded instances and different encodings. The features encode a wide range of different information on the problem such as problem size, features of the graph-based representation, balance features, the proximity to a Horn formula, DPLL probing features and local search probing features.

We also consider the super-set of all 198 available features, 36 CSP plus 3 × 54 SAT features. This extracts the largest amount of information from the instance, but comes at a much higher computational cost.

Constraint Solvers. Our CSP models are able to choose from four complete CSP solvers:

- Abscon [111],
- Choco [154],
- Gecode [55], and
- Mistral [76].

Satisfiability Solvers. We considered the following six complete SAT solvers:

- `clasp` [54],
- `cryptominisat` [151],
- `glucose` [15],
- `lingeling` [21],
- `riss` [117], and
- Minisat [43].

Learning Algorithms. We evaluate a number of regression, classification, and clustering algorithms using WEKA [74]. All algorithms, unless otherwise stated, use the default parameters. The regression algorithms we used were LinearRegression, PaceRegression, REPTree, M5Rules, M5P, and SMOreg. The classification algorithms were BayesNet, BFTree, ConjunctiveRule, DecisionTable, FT, HyperPipes, IBk (nearest neighbour) with 1, 3, 5 and 10 neighbours, J48, J48graft, JRip, LADTree, MultilayerPerceptron, OneR, PART, RandomForest, RandomForest with 99 random trees, RandomTree, REPTree, and SimpleLogistic. For clustering, we considered EM, FarthestFirst, and SimplekMeans. The FarthestFirst and SimplekMeans algorithms require the number of clusters to be given as input. We evaluated with multiples of 1 through 5 of the number of solvers in the respective data set given as the number of clusters. The number of clusters is represented by $1n$, $2n$ and so on in the name of the algorithm, where n stands for the number of solvers.

We use the LLAMA toolkit [105] to train and test the algorithm selection models.

4.5.2 Portfolio and Solver Results

The performance of each of the six SAT solvers was evaluated on the three SAT encodings of 1,493 CSP competition benchmarks with a time-out of 1 hour and limited to 2GB of RAM. The four CSP solvers were evaluated on the original CSPs. Our results report the PAR10 score and number of instances solved for each of the algorithms we evaluate. The PAR10 is the sum of the runtimes over all instances, counting 10 times the timeout if that was reached. Data was collected on a cluster of Intel Xeon E5430 Processors (2.66Ghz) running CentOS 6.4. This data is available online[1] and has been integrated into ASlib [24].

The performance of a number of hierarchical approaches is given in Table 4.1. The hierarchy of algorithms which produced the best overall results for our dataset involves M5P regression with CSP features at the root node to choose SAT or CSP, M5P regression with CSP features to select the CSP solver, LinearRegression with CSP features to select the SAT encoding, LinearRegression with CSP features to select the SAT solver for the direct encoded instance, LinearRegression with CSP features to select the SAT solver for the direct-order encoded instance, and LinearRegression with the direct-order features to select the SAT solver for the support encoded instance. The hierarchical tree of specific machine learning approaches we found to deliver the best overall performance on our data set is labelled Proteus and is depicted in Figure 4.4.

We would like to point out that in many solver competitions the difference between the top few solvers is fewer than 10 additional instances solved. In the 2012 SAT Challenge for example, the difference between the first and second place single solver was only 3 instances and the difference among the top 4 solvers was only 8

[1] Proteus runtime data: `http://4c.ucc.ie/~bhurley/proteus/`

Table 4.1: Performance of the learning algorithms for the hierarchical approach. The 'Category Bests' consists of the hierarchy of algorithms where at each node of the tree of decisions we take the algorithm that achieves the best PAR10 score for that particular decision.

Classifier	Mean PAR10	Number Solved
VBS	97	1493
Proteus	1774	1424
M5P with CSP features	2874	1413
Category Bests	2996	1411
M5Rules with CSP features	3225	1398
M5P with all features	3405	1397
LinearRegression with all features	3553	1391
LinearRegression with CSP features	3588	1383
MultilayerPerceptron with CSP features	3594	1382
lm with CSP features	3654	1380
RandomForest99 with CSP features	3664	1379
IBk10 with CSP features	3720	1377
RandomForest99 with all features	3735	1383

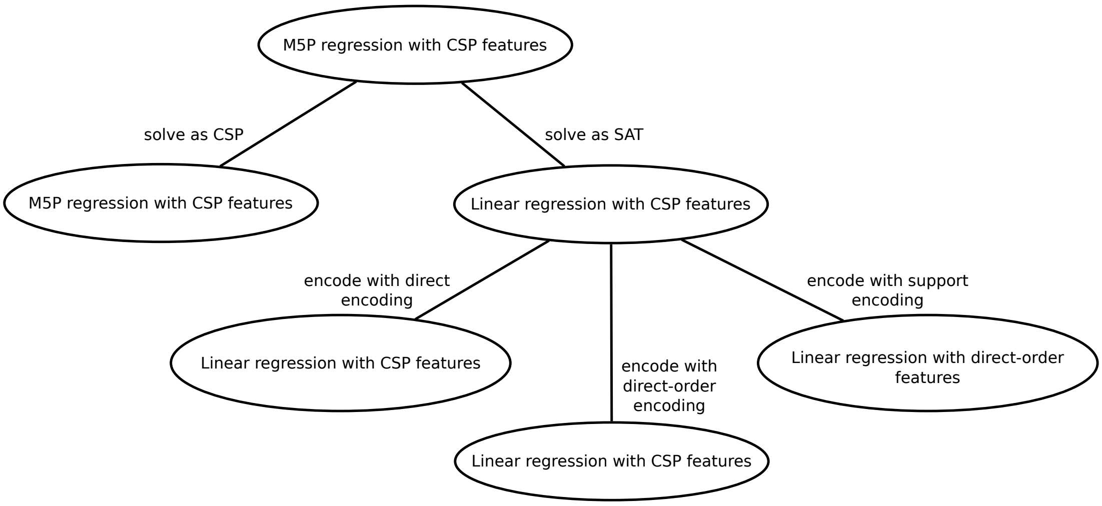

M5P regression with CSP features
solve as CSP
solve as SAT
M5P regression with CSP features
Linear regression with CSP features
encode with direct encoding
encode with support encoding
Linear regression with CSP features
encode with direct-order encoding
Linear regression with direct-order features
Linear regression with CSP features

Figure 4.4: Overview of the machine learning models used in the hierarchical approach.

instances. The results we present in Table 4.1 are, therefore, very significant in terms of the gains we are able to achieve.

Our results demonstrate the power of Proteus. The performance it delivers is very close to the virtual best (VBS), that is the best performance possible if an oracle could identify the best choice of representation, encoding, and solver, on an instance by instance basis. The improvements we achieve over other approaches are similarly impressive. The results conclusively demonstrate that having the option to convert a CSP to SAT does not only have the potential to achieve significant performance improvements, but also does so in practice.

An interesting observation is that the CSP features are consistently used in each of the top performing approaches. One reason for this is that it is quicker to compute only the CSP features instead of the CSP features, then converting to SAT and computing the SAT features in addition. Computing the SAT features may elicit more information about the instance structure, but it incurs increased overhead, which may only be worthwhile in some cases. For example, the LinearRegression model gives its best performance using all feature sets, despite the overhead of translating and computing features of each encoding. Note that for the best tree of models (cf. Figure 4.4), it is better to use the features of the direct-order encoding for the decision of which solver to choose for a support-encoded SAT instance despite the additional overhead.

Table 4.2: Ranking of each classification, regression, and clustering algorithm to choose the solving mechanism in a flattened setting. The portfolio consists of all possible combinations of the 3 encodings and the 6 SAT solvers and the 4 CSP solvers for a total of 22 solvers.

Classifier	Mean PAR10	Number Solved
VBS	97	1493
Proteus	1774	1424
LinearRegression with all features	2144	1416
M5P with csp features	2315	1401
LinearRegression with csp features	2334	1401
lm with all features	2362	1407
lm with csp features	2401	1398
M5P with all features	2425	1404
RandomForest99 with all features	2504	1401
SMOreg with all features	2749	1391
RandomForest with all features	2859	1386
IBk3 with csp features	2877	1378

We also compare the hierarchical approach to that of a flattened setting with a single portfolio of all solvers and encoding solver combinations. The flattened portfolio includes all possible combinations of the 3 encodings and the 6 SAT solvers and the 4 CSP solvers for a total of 22 solvers. Table 4.2 shows these results. The regression algorithm LinearRegression with all features gives the best performance using this approach. However, it is significantly worse than the performance achieved by the hierarchical approach of Proteus.

4.5.3 Greater than the Sum of its Parts

Given the performance of Proteus, the question remains whether a different portfolio approach that considers just CSP or just SAT solvers could do better. Table 4.3 summarizes the virtual best performance that such portfolios could achieve. We use all the CSP and SAT solvers for the respective portfolios to give us VB CSP and VB SAT, respectively. The former is the approach that always chooses the best CSP solver for the current instance, while the latter chooses the best SAT encoding/solver combination. VB Proteus is the portfolio that chooses the best overall approach/encoding. We show the actual performance of Proteus for comparison. Proteus is better than the virtual bests for all portfolios that consider only one encoding. This result makes a very strong point for the need to consider encoding and solver in combination.

Proteus outperforms four other VB portfolios. Specifically, the VB CPHyrda is the best possible performance that could be obtained from that portfolio if a perfect choice of solver was made. Neither SATzilla nor ISAC-based portfolios consider different SAT encodings. Therefore, the best possible performance either of them could achieve for a specific encoding is represented in the last four lines of Table 4.3.

These results not only demonstrate the benefit of considering multiple CSP solving techniques, but also eliminate the need to compare with existing portfolio systems since we are computing the best possible performance that any of those systems could theoretically achieve. Proteus impressively demonstrates its strengths by significantly outperforming oracle approaches that use only a single encoding.

4.6 CSP versus SAT Analysis

This section presents a more detailed study of the crucial decision of when to choose CSP or SAT. We first examine the performance differences between CSP and SAT by breaking it out by instance category. Subsequently, we attempt to explain the performance of the portfolio, how it is able to exploit the performance differences between the two paradigms so effectively.

4.6.1 Empirical Setup

Using the empirical performance data of the 1493 instances collected in Section 4.5, we take the time in seconds of the virtual best CSP solver t_{CSP} and the virtual best SAT solver t_{SAT}. The absolute difference in orders of magnitude is $\delta = |\log(1 + t_{CSP}) - \log(1 + t_{SAT})|$. Instances where δ is less than one order of magnitude are excluded, giving 1249 instances where the difference between the virtual best CSP and virtual best SAT is significant. Each instance is labeled CSP or SAT, corresponding with the paradigm giving the virtual best performance, under any encoding and solver. This gives a set which is biased towards CSP, 976 versus 273.

Table 4.3: Virtual best performances ranked by PAR10 score.

Method	Mean PAR10	Number Solved
VB Proteus	97	1493
Proteus	1774	1424
VB CSP	3577	1349
VB CPHydra	4581	1310
VB SAT	17373	775
VB DirectOrder Encoding	17637	764
VB Direct Encoding	21736	593
VB Support Encoding	21986	583

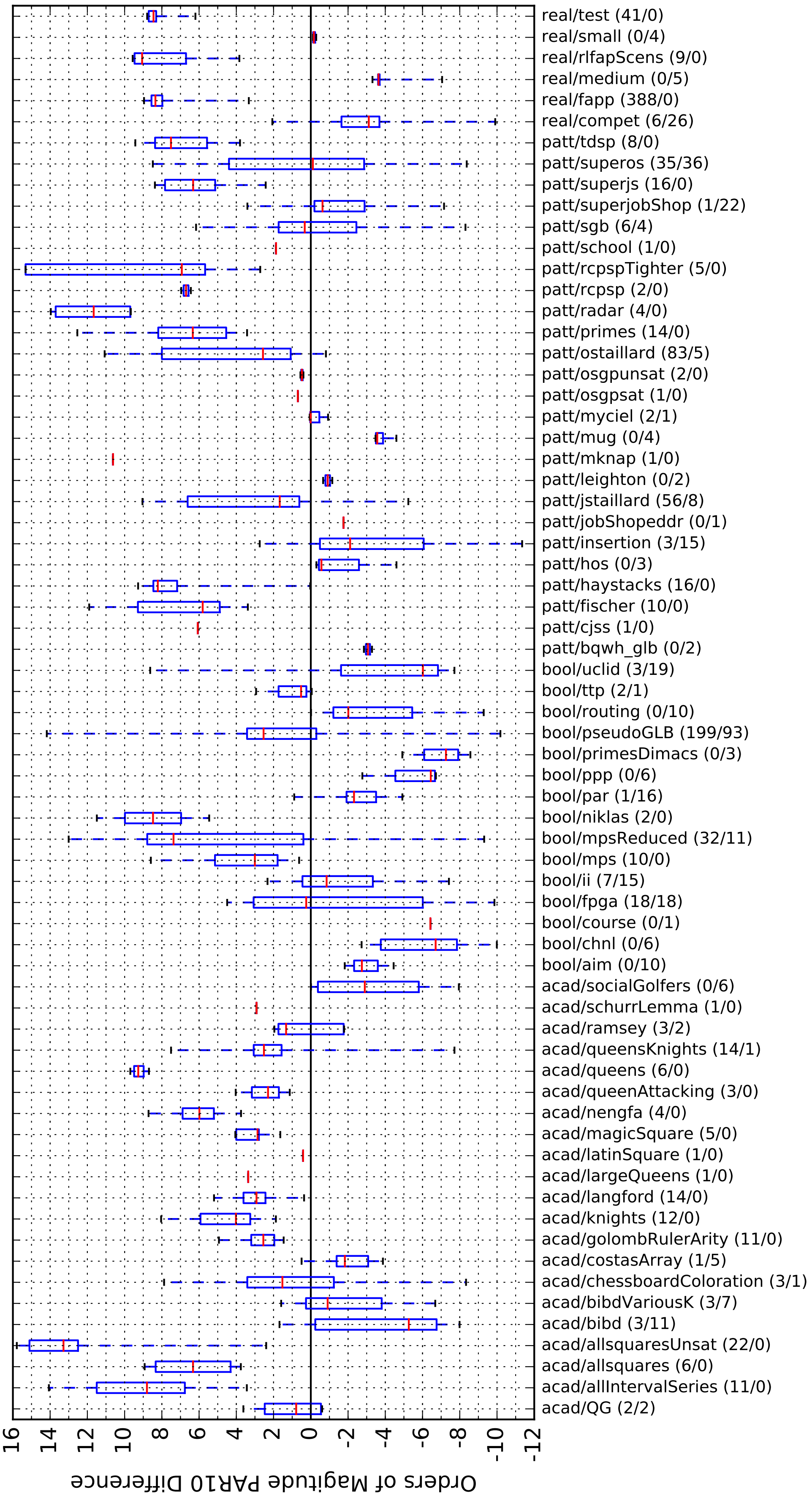


Figure 4.5: Box-plot of the orders of magnitude PAR10 difference between the virtual best CSP and virtual best SAT solvers. A positive (resp. negative) value signifies that CSP (resp. SAT) was faster. The values in brackets after the instance category name shows the number of instances where CSP/SAT was faster.

4.6.2 Orders of Magnitude Differences

Figure 4.5 summarises the complete performance difference between the virtual best CSP solver and the virtual best SAT solver. The difference between the two is shown in the form of a box-plot, where the central box ranges between the first and third quartiles, with whiskers extending to the minimum and maximum values, and the central red line plotting the median. Instances are grouped by their respective problem category, as defined on their source [1]. It is difficult to garner concrete insights from some instance categories which only contain a handful of instances but we present some from the reasonably sized categories.

First, note that there are a number of problem categories where all instances are clearly suited more to CSP, and likewise there are categories clearly more suited to SAT. For example, the real/fapp category has a large number of instances which are clearly more suited to a CSP solver. These are instances of the *Frequency Assignment Problem with Polarization*, having relatively large, sparse domains (in the hundreds of values) and complex, non-linear expressions as constraints. The SAT encoding of such problems is quite large, requiring a large number of variables and auxiliary encodings of the intermediate non-linear expressions.

Many of the bool categories, containing instances with only Boolean domains, perform relatively well in SAT. These instances will elicit quite compact SAT representations so it would seem natural that the SAT solvers would perform well. Interestingly, this category also includes some pseudo-Boolean instances which contain linear expressions over the Boolean variables. SAT encodings of linear expressions is a highly-active research topic as it has huge implications [4, 5, 31, 50, 60, 100, 148]. In this work, we have employed a naïve encoding whereby the linear expression is decomposed, chaining the cumulative sum of each variable to a new variable. Therefore, it is encouraging to still see the SAT solvers achieve relatively good performance on these instances. If a more sophisticated encoding were used for these expressions, the performance may have been even better.

Contrary to this, there are several categories which have a mixture of instances where CSP is faster and some where SAT is the faster choice, such as bool/fpga, bool/pseudoGLB, patt/jstaillard, and patt/superos. In particular, the patt/superos category has an even split between CSP and SAT being faster. These are instances of Taillard's Open Shop Scheduling problem which have been modified to include constraints for finding super-solutions [77]. The problems consist of the original disequality constraints from the scheduling problem and additional constraints of the form $(X_1 + X_2 \leq X_3 \vee X_0 + X_3 \leq X_1)$ and $(X_0 + X_1 \leq X_2 \vee X_1 + X_2 \leq X_0)$ to find super-solutions. Given that these problems contain variables with domain size in the range of hundreds to thousands, it was unexpected to see the SAT solvers perform so well. We could not distinguish any specific characteristics about the individual instances which would indicate the expected performance one way or the other, in particular the performance differences were not tied to instance satisfiability nor size. There is certainly scope for a more detailed investigation of this behaviour, which we consider as potential future work.

4.6.3 Explaining Portfolio Performance

At the top of the Proteus hierarchy is the decision to choose the CSP or SAT route, aiming to choose the paradigm which is most likely to be the fastest. A bad decision at this point could have a detrimental effect which is compounded by decisions later on in the hierarchy. This section studies the top-level CSP or SAT decision, demonstrating that a relatively simple machine learning model can be highly accurate at choosing the best paradigm.

For the purpose of this analysis, a decision tree classifier will be employed to make the CSP/SAT decision. It is not the model which formed part of the overall best hierarchical portfolio, but it is more transparent, allowing us to study instance features which the model considered most important to distinguishing CSP versus SAT.

Table 4.4 summarises the results of a stratified 10-fold cross-validation whereby the dataset is split into 10 evenly sized folds with a consistent distribution of the labels across the folds. For each fold, the table lists the size of the training and test sets, broken out by the number of instances where CSP/SAT was faster, accuracy, precision, recall, and Matthews Correlation Coefficient. The first four metrics are susceptible to bias in the dataset, for example simply predicting the majority label of CSP gives an accuracy of 78.1%. On the other hand, *Matthews Correlation Coefficient* (MCC) is designed to measure the quality of binary classifiers, aiming to be a balanced measure even when the classes are of very different sizes. A MCC of +1 means perfect prediction, −1 an inverse prediction, and 0 an average random prediction. Predicting the majority label of CSP gives a MCC of 0. Also, a uniformly random-classifier and one that predicts according to the training set distribution gives values very close to 0 also.

Table 4.4: Stratified 10-fold cross validation predicting CSP or SAT by a decision tree classifier using CSP features.

	CSP/SAT						
Fold	Train	Test	Accuracy	Precision	Recall	F1-Score	Matthews
1	879/245	97/28	0.9680	0.9697	0.9897	0.9796	0.9066
2	879/245	97/28	0.9440	0.9787	0.9485	0.9634	0.8467
3	879/245	97/28	0.9440	0.9688	0.9588	0.9637	0.8412
4	878/246	98/27	0.9520	0.9600	0.9796	0.9697	0.8554
5	878/246	98/27	0.9440	0.9789	0.9490	0.9637	0.8430
6	878/246	98/27	0.9440	0.9691	0.9592	0.9641	0.8371
7	878/246	98/27	0.9200	0.9151	0.9898	0.9510	0.7524
8	878/246	98/27	0.9440	0.9596	0.9694	0.9645	0.8326
9	878/246	98/27	0.9440	0.9691	0.9592	0.9641	0.8371
10	879/246	97/27	0.9839	1.0000	0.9794	0.9896	0.9549
Overall		976/273	0.9488	0.9663	0.9682	0.9672	0.8496

Table 4.5: Mean feature importance for stratified 10-fold cross validation predicting CSP or SAT by a decision tree classifier using CSP features.

Feature	Gini Importance
Average Predicate Arity	0.28357
Log Number of Values	0.12687
Average Predicate Size	0.10092
Log Number of Ranges	0.08386
Maximum Arity	0.05512
Perc. of Constraints - Weighted Sums	0.05190
Log Number of Bits	0.04287
Log Number of Booleans	0.04166
Log Number of Search Variables	0.04008
Log Number of Constraints	0.03872
Sqrt. Average Domain Size	0.03555
Sqrt. Maximum Domain Size	0.02569
Log Number of Extra Booleans	0.02037
Perc. of Constraints - Element	0.01321
Log Number of Extra Ranges	0.00982

Achieving accuracy of 95% overall and a MCC of 0.85, all metrics clearly show that a single decision tree is highly effective at choosing when to solve an instance as CSP or SAT. Next, we examine what instance specific features the decision tree employs in order to effectively make its choice.

Table 4.5 presents the set of features used by the decision tree, sorted by their Gini importance. The Gini importance measure is the accumulated sum of the reduction in Gini-impurity that the feature contributed, i.e. its effectiveness at separating different instances. The arity of predicates is by far the most important feature for the decision tree, accounting for 28% of the reduction in Gini-impurity. The next two most important features, the number of values and average predicate size, also relate to the size of constraint expressions. As an aside, if a single rule classifier is built, i.e. a decision tree with depth limited to one node, it also chooses the predicate arity as the distinguishing feature in all folds, achieving an accuracy of 84% overall and MCC of 0.501.

For illustration purposes, we limit the decision tree to a node-depth of 3 and present the resulting decision tree, trained on fold number 10, in Figure 4.6. Based on the set of features that the decision tree chooses to distinguish CSP versus SAT, there appears to be a consistent pattern emerging. In general terms, it appears that instances involving smaller arity expressions, or smaller domains are more suited to SAT. This seems like a natural explanation of SAT's superior performance, but there are also some large arity constraints, with small domains

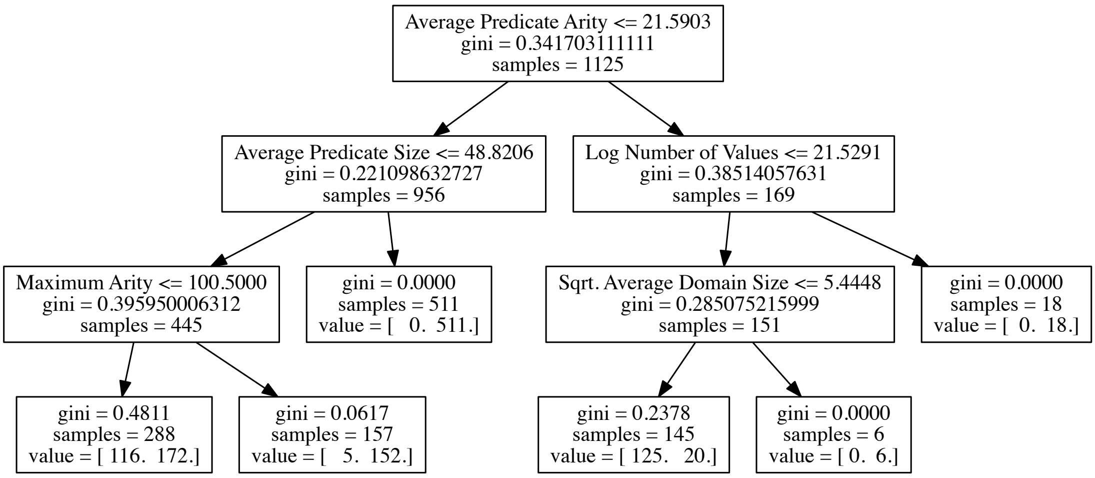


Figure 4.6: An example decision tree (limited to depth 3) choosing SAT or CSP. Each node is labelled with the feature and threshold value to decide a split, values below (resp. above) the threshold take the left (resp. right) branch. The values in brackets on leaf nodes denote the number of instances in the training set labelled SAT and CSP respectively.

where SAT performs well. CSP seems to be the paradigm of choice for larger, more complex expressions. Additionally, looking at the left most branch in Figure 4.6, containing instances with smaller predicates and lower maximum arity, there is an almost even split of CSP versus SAT, suggesting it not simply the small instances where SAT dominates.

The set of CSP instance features we consider here did not include features studying the detailed structure, such as summarising the constraint graph, tree width, etc. The features we employ in this chapter emanate from their successful application in the CPHydra and Proteus portfolios. If additional structural features were considered they might provide further insight into the relative empirical performances. This may be considered as an avenue for future work.

4.7 Proteus for Graphical Model Optimisation

A number of paradigms closely-related to constraint programming present new challenges and ample opportunity to apply Proteus-like techniques to potentially improve the state-of-the-art. In particular, Graphical Models are an important and widely studied area of artificial intelligence and statistics. They provide the capability to tackle some highly-important applications, ranging from image-processing to bioinformatics. It benefits from an active research community, dedicated annual conferences, and a semi-regular solver competition. Unlike mixed integer programming, which has a very small collection of high-performing solvers, the graphical model community has a wide range of closely-related complementary paradigms as well as competitive solvers.

The field presents an excellent opportunity for the application of portfolio techniques, yet has remained relatively untouched to-date. Nevertheless, in order to apply such techniques, a number of significant challenges must be faced. Specifically, with the switch to an optimisation problem, not only does the validity of the solution need to be considered, but also the quality of the solution, and whether it was proven to be optimal, can be quite important. Additionally, a set of empirically distinguishing features is lacking in the area.

The remainder of this section presents an application of some Proteus-like techniques to the graphical models domain, with its effectiveness independently verified through a first place entry to the UAI 2014 Probabilistic Inference Competition. The contents of this section have been published in [93], we present here the details pertinent to this thesis. For further details on areas such as background, please refer to the journal article.

4.7.1 Graphical Models Background

Weighted variants of constraint networks such as Cost Function Networks (CFNS), also known as Weighted Constraint Satisfaction Problems (WCSPs), aim at finding an assignment to all variable that minimises the sum of local cost functions [120]. With a language restricted to Boolean variables and clausal form constraints, the weighted partial Max-SAT (WPMS) problem has the same target [120].

In artificial intelligence and statistics, probabilistic graphical models [102] use the same idea to concisely represent probability distributions over random variables. These models include Bayesian Networks and Markov Random Fields (MRFs). The problem of identifying a variable assignment that has maximum probability is called the *Maximum Probability Explanation* in Bayesian networks, or *Maximum A-Posteriori* (MAP) in MRF. By a simple $(-\log)$ transformation, these problems can be reduced to CFNs. Graphical Models can also be easily encoded as 0/1 Linear Programming (01LP) problems, a standard language for Operations Research (OR). Two encodings will be considered, namely the *direct* encoding, and the *tuple* encoding which is based on the so-called *local polytope* [64, 104, 143].

4.7.1.1 Combinatorial Optimisation Languages

We briefly describe the combinatorial optimisation languages that will be used. Each language has emanated from individual applications, with dedicated algorithms for solving instances of each being developed over the decades. However, it is possible to translate problems between these languages, something which will be presented in the subsequent section.

[CFN] Cost Function Networks extend Constraint Networks by using non-negative cost functions instead of constraints [120]. A CFN is a triple (X, W, k) where $X = \{1, \ldots, n\}$ is a set of n discrete variables, W is a set of non-negative functions, and k, a possibly infinite maximum cost. Each variable $i \in X$ has a finite domain D_i of values that can be assigned to it. A function $w_S \in W$, with scope $S \subseteq X$, is a function $w_S : D_S \mapsto \{\alpha \in \mathbb{N} \cup \{k\} \ : \ \alpha \leq k\}$, where D_S denotes the Cartesian product of all D_i for $i \in S$. In CFNs, the cost of a complete assignment is the sum of all cost functions. A solution has cost less than k. Therefore a cost of k denotes forbidden assignments, used in hard constraints. A solution of minimum cost is sought.

[MRF] Markov Random Fields define a pair (X, Φ) where $X = \{1, \ldots, n\}$ is a set of n random variables, and Φ is a set of potential functions. Each variable $i \in X$ has a finite domain D_i of values that can be assigned to it. A potential function $\phi_S \in \Phi$, with scope $S \subseteq X$, is a function $\phi_S : D_S \mapsto \mathbb{R} \cup \{\infty\}$. We consider the MAP query that aims at finding a complete assignment of maximum probability (or equivalently, minimum energy).

[WPMS] Weighted Partial MaxSAT problems are CFNs restricted to Boolean domains and a language of weighted clauses [23]. An instance is defined as a set of pairs $\langle C, w \rangle$ and an upper bound k. Each C is a clause and w is a number in $\mathbb{N} \cup \{k\}$, the *weight* of clause C. A clause is a disjunction of literals, and a literal is a Boolean variable or its negation. A clause with weight $\geq k$ is a *hard* clause, otherwise it is *soft*. The objective is to find an assignment to the variables appearing in the clauses that minimises the sum of the weights of all falsified clauses, which should be of cost $< k$.

[01LP] A 0/1 Linear Program is defined by a linear objective function over a set of 0/1 variables to minimise under a conjunction of linear equalities and inequalities [162].

[CP] Constraint Programming problems are defined by a set of discrete variables and a set of constraints. The aim is to minimise the value of a given objective variable while satisfying all constraints [139].

4.7.2 Translations Between Formalisms

Table 4.6 summarises for each input formalism the different translations used to produce instances in the corresponding output formalism.

Additive MRFs can be reduced to CFNs using a fixed decimal point representation of energies which are scaled to integers and shifted to enforce non-negativity.

Table 4.6: Summary of translations between graphical model formalisms.

In/Out	MRF (UAI)	CFN (WCSP)	WPMS (WCNF)	01LP (LP)	CP (MINIZINC)
MRF	-	$-\log(prob)$	Through CFN	Through CFN	Through CFN
CFN	$\exp(-cost)$	-	Direct/tuple encoding	Direct/tuple encoding	Extra cost vars & table cons.
WPMS	Through CFN	Direct translation	Direct encoding only	Through CFN	Extra cost vars & reified logical *or*
CP	Through CFN	Decomposed objective & global constraints	Through CFN	Through CFN	-

Multiplicative MRFs can be transformed to additive MRFs using a simple $(-\log)$ transform, and then to CFNs.

As weighted partial MaxSAT is a CFN with Boolean variables and a language of clauses, thus a WPMS instance is already a CFN. For a CFN, we consider two encodings to WPMS based on CSP to SAT encodings: the *direct* encoding [14], and the *tuple* encoding introduced by Bacchus [17].

The 01LP encodings of CFNs are similar to those for WPMS, using the direct and tuple encodings but with 0/1 variables. The additional expressivity of linear constraints enables some simplifications.

Translating CFNs into crisp CSPs involves adding additional cost variables and table constraints for each cost function. Likewise for translating WPMs to CSP but using reified Boolean expressions. The converse translation of CP models with a cost variable into a CFN (and then MRFs and WPMSs) that does not use cost variables is a complex task requiring local cost functions to be identified, starting from the objective variable, while removing intermediate cost variables. Global constraints are decomposed into ternary cost functions in extension, which limits the translation to instances with small domain sizes.

For a more detailed description of these translations, see [93].

In the portfolio evaluation to follow, we consider a subset of the benchmarks and the solvers such that all the instances could be translated to all the solvers, i.e. we exclude the WPMS and CP benchmarks, and the GECODE solver due to the prohibitive size of the encoded problem. The information for these languages are presented here for completeness, performance details outside the context of the portfolio evaluation are available in [93].

4.7.3 Graphical Model Benchmarks

An extensive set of benchmarks representing optimisation problems from various areas was collected from different sources including deterministic (CFN, MaxCSP, WPMS), probabilistic (MRF, BN), as well as CP collections. Each collection contains several categories of instances, each category corresponding to a specific class of problems.

Together, these benchmark resources contain problems offering a large variety in terms of size, maximum arity, domain size, and cost range. WPMS and CVPR categories have the highest number of variables (close to 1 million variables for WPMS/TimeTabling, half a million for CVPR/PhotoMontage and ColorSeg). The WPMS benchmark also has the largest arities (a weighted clause on 580 variables appears in Haplotyping). For the other benchmarks, maximum arity varies from 2 to 5. Graph connectivities are usually very small for MRF & CVPR (often based on grid graphs where vertices represent pixels in images) and WPMS benchmarks. MRF/ObjectDetection, CFN/ProteinDesign, MaxCSP/Langford, and CVPR/Matching have complete graphs. MRF/ProteinFolding has the largest domain size (503 values). Most CVPR instances have very large cost ranges (8-digit precision), whereas MaxCSP instances contain only 0/1 costs. The emphasis between optimisation and feasibility also varies a lot among the problems: almost all deterministic GM categories, except MaxCSPs and CFN/CELAR, contain forbidden (k) tuples in their cost functions. On the contrary, probabilistic GMs usually have no forbidden tuples (except for MRF/Linkage and DBN).

Table 4.7 reports the number of instances per benchmark resource and its gzipped size for the seven formulations. The UAI format appears to be the most compact to express local functions as tables. It relies on a complete ordered table of costs which does not require describing tuples whereas the other formats explicitly describe tuples associated to non-zero costs. The price to pay for this conciseness is the inability of the UAI format to represent large arity functions with a few non-zero costs (such as large weighted clauses). As seen before, the tuple encoding is usually larger than the direct one, except for MRF/CVPR LPs where the local polytope is a good choice since there are almost no zero costs. CP instances

Table 4.7: Number of instances and their total compressed (gzipped) size per format for each benchmark resource

Benchmark	Nb.	UAI	WCSP	LP (direct)	LP (tuple)	WCNF (direct)	WCNF (tuple)	MINIZINC
MRF	319	187MB	475MB	2.4G	2.0GB	518MB	2.9GB	473MB
CVPR	1461	430MB	557MB	9.8GB	11GB	3.0GB	15GB	N/A
CFN	281	43MB	122MB	300MB	3.5GB	389MB	5.7GB	69MB
MaxCSP	503	13MB	24MB	311MB	660MB	73MB	999MB	29MB
WPMS	427	N/A	387MB	433MB	N/A	717MB	N/A	631MB
CP	35	7.5MB	597MB	499MB	1.2GB	378MB	1.9GB	21KB
Total	3026	0.68GB	2.2GB	14GB	18GB	5GB	27GB	1.2GB

benefit from global constraints in the MINIZINC language, which are decomposed in large tables in the other formats.

4.7.4 Experimental Setup

We consider state-of-the-art MRF solvers DAOOPT [129], winner of PIC 2011, and TOULBAR2 version 0.9.8 [39, 45] (including Virtual Arc Consistency (VAC) as preprocessing [34], dominance rule pruning [40], and hybrid best-first search [9]), winner of MaxCSP 2008 and UAI 2010 & 2014 Evaluations, against WPMS MAXHS solver [36, 37], winner of crafted WPMS MaxSAT 2013, the CP solver GECODE, winner of MiniZinc Challenges 2012, and IBM-ILOG CPLEX 12.6.

All computations were performed on a single core of AMD Opteron 6176 at 2.3 GHz and 8 GB of RAM with a 1-hour CPU time limit.

4.7.5 Graphical Model Instance Features

In order to build a prediction model for the portfolio, a set of distinguishing features is required. To describe graphical model instances, we consider the following feature set:

1. the input file size,
2. the CPU time to read the instance,
3. an initial upper bound on the solution,
4. the time to compute the initial upper bound,
5. the number of variables,
6. the number of cost functions.
7. The ratio of unary,

8. binary, and
9. ternary cost functions, i.e. the fraction of the total number of cost functions of each arity.
10. The ratio of cost functions which have arity 4 or greater.
11. Finally, a number of statistics such as the mean, standard deviation, coefficient of variation, minimum, and maximum for domain size, and
12. cost function arity.

By no means does this list constitute a comprehensive list of possible features for graphical models, nevertheless in initial evaluations these proved effective and have the benefit of being relatively cheap to compute.

Table 4.8 presents the Gini importances [27] of the above features according to a decision tree classifier aiming to predict the fastest solver. The Gini importance measure is the accumulated sum of the reduction in Gini-impurity that the feature contributed, i.e. its effectiveness at separating different instances. The most important features are the ratio of binary cost functions, the minimum domain size, and the value of the initial upper bound. These features enable instances with differing fastest solvers to be separated, accounting for between 10-15% of the reduction in the Gini-impurity. The next three most valuable features, namely the time to read, the time to compute the initial upper bound, and the file size may be viewed as a proxy for the size of the problem.

Table 4.8: Gini importances of graphical model features, truncated at 1%.

	Feature	Gini importance
1	Ratio of binary cost functions	14.445%
2	Minimum domain size	13.928%
3	Initial upper bound	10.988%
4	Time to read	9.211%
5	Time to compute upper bound	8.393%
6	File size	8.305%
7	Coefficient of variation of constraint arity	7.546%
8	Stadard deviation of constraint arity	7.149%
9	Mean constraint arity	6.555%
10	Number of variables	4.304%
11	Number of cost functions	3.875%
12	Mean domain size	1.634%

Table 4.9: Summary of portfolio approaches sorted by decreasing number of problems solved over the 2,564 instances.

Solver	Solved time (sec.)		Num.	Num.	Misclass. pen.	
	Mean	Std. dev.	solved	best	solved	total time
VBS(6)	93.0	385.1	2,321			
M5P regression	91.5	376.1	2,298			
J48 classification	84.7	368.1	2,294			
Random Forest	74.6	327.6	2,279			
k-means clustering	66.9	301.4	2,259			
TOULBAR2	105.2	408.3	2,220	1,863	224	28,000.1
CPLEX$_{tuple}$	55.4	316.6	1,852	27	3	10,345.3
DAOOPT	535.1	340.1	1,812	3	0	3,236.8
MAXHS$_{tuple}$	140.0	414.5	1,551	3	1	8.4
MAXHS	199.0	565.4	1,078	208	4	9,261.4
CPLEX	127.7	433.4	1,002	217	36	14,381.9

4.7.6 Offline Evaluation Results

Table 4.9 presents an offline evaluation of a simple portfolio approach based on 6 graphical model solvers listed in Section 4.7.4. We consider a subset of the benchmarks and the solvers such that all the instances could be translated to all the solvers, i.e. we exclude the WPMS and CP benchmarks, and the GECODE solver from our portfolios and evaluation. For a detailed breakdown of their respective contributions overall see [93].

The portfolio is built using LLAMA [105], with 10-fold stratified cross validation. This involves splitting the dataset into 10-equally sized folds with an equal distribution of the best solver across folds. For brevity, we present results only for the best performing regression, classification, and clustering methods, plus the Random Forest classifier. The *Virtual Best Solver* (VBS) corresponds to an oracle deciding the best solver for each instance. The table lists the mean and standard deviation of CPU time on the solved instances, the number of instances solved to optimality in less than 1 hour, the number of times each solver was the fastest. In addition, the misclassification penalty shows the contribution of each solver to the portfolio, i.e. the number of instances that were not solved by any other solver, and, where another one solved the instance, the additional CPU time needed by the next best solver.

From these statistics alone, it is clear that each of the component solvers (except MAXHS$_{tuple}$) play a valuable contribution to the portfolio both in terms of being able to solve more instances, and reducing the overall CPU time needed. All six solvers contribute to improving the performance of the virtual best solver. Additionally, each of the portfolio methods are able to outperform the single best solver and close most of the gap to the virtual best solver. The single best solver, toulbar2, is quite dominant on its own, yet there are 101 additional instances solved by the remaining solvers, as well as a significant amount of CPU-time to be saved by faster solvers.

4.7.7 UAI 2014 Probabilistic Inference Competition – Portfolio Entry

Based on the offline evaluation of Section 4.7.6, a portfolio was trained and submitted to the UAI 2014 Inference Competition (MAP task). The source code to train and run the portfolio is available online[2]. It was built from five constituent solvers: i) toulbar2, ii) a version of toulbar2 taking a starting solution from an initial run of the incop [124] local search solver, iii) the Message Passing Linear Programming mplp2 solver [149, 150], iv) cplex using the direct encoding, and v) cplex with the tuple encoding. These solvers were selected based on their complementary performances in previous empirical evaluations.

The submitted portfolio was built using a random forest classifier from scikit-learn [131] with default parameters except increasing the number of classifier trees to 99, and using the feature set listed in Section 4.7.5. Additional classifiers such as a pairwise random-forest and decision trees were also evaluated offline but, for our purposes, were no better than a single random-forest.

4.7.7.1 Offline Evaluation

In order to gauge the respective performance and contributions of the component solvers, we undertook an offline evaluation, considering the 2,564 benchmark instances described in Section 4.7.3. Table 4.10 summarises the performance of the portfolio and its component solvers.

Firstly, the virtual best solver is composed of all five solvers, with each solver playing a non-negligible part. incop+toulbar2 is a highly effective combination, solving more instances than any other solver, and having a mean solution time faster than toulbar2 alone. It solved 23 instances which went unsolved by

[2] Source code for UAI-Proteus portfolio `https://github.com/9thbit/uai-proteus`

any other solver, and improves on the next faster time of any other solve to a cumulative 82,616 seconds. TOULBAR2 alone is a close second, solving a similar number of instance to optimality, and notably being the fastest overall on 1,449 occasions. MPLP2 pays little contribution to the portfolio in this scenario, only helping to improve the virtual best cumulative CPU-time by 1,183 seconds.

Somewhat surprisingly, CPLEX$_{direct}$ has a relatively poor performance overall, solving the lowest number of instances to optimality, less than half that of the best single solvers. However, it is able to solve 46 instances that went unsolved by any other solver, twice that of the next-best unique contribution.

Finally, the fact that there is a total of 88 instances uniquely solved, and 163,000 seconds of CPU-time to be saved, shows strong potential for a portfolio. The UAI'14 portfolio is able to exploit the majority of these complementary strengths, solving the most instances overall and closing 56% of the gap between the best single and the virtual best solvers.

4.7.7.2 Competition Results

The effectiveness of this multi-language portfolio was independently verified in the UAI 2014 Inference Competition, achieving two first places in the MAP task under both the 20 and 60 minute timeouts[3].

The UAI 2014 Inference Competition used the cumulative solver rankings as the overall evaluation metric. For each instance, solvers are ranked by their relative performance, in terms of solution quality, and the respective ranks are added across instances. We note that ties may occur in instance rankings, and lower is

Table 4.10: Offline evaluation of the UAI 2014 portfolio on 2,564 instances

Solver	Solved time (s)		Num.	Num.	Misclass. pen.	
	Mean	Std. dev.	solved	best	solved	time (s)
VBS(5)	63.5	276.3	2,315			
UAI'14 portfolio	71.8	312.4	2,276			
INCOP+TOULBAR2	87.6	361.2	2,227	352	23	82,616.2
TOULBAR2	105.2	408.3	2,220	1,449	13	56,339.3
CPLEX$_{tuple}$	55.4	316.6	1,852	27	6	9,584.7
MPLP2	66.2	424.6	1,537	198	0	1,183.3
CPLEX$_{direct}$	127.7	433.4	1,002	289	46	13,276.0

[3] See MAP/Proteus entry at `http://www.hlt.utdallas.edu/~vgogate/uai14-competition/leaders.html`.

better in this metric. Figure 4.7 plots the cumulative ranking across instances for the MAP category, to which the *Proteus* portfolio described in this section was entered.

Three of the portfolio's component solvers were submitted to the same competition as independent entries. The two 01LP encodings performed extremely well on certain instances but extremely poorly on the remaining[4]. Based on the competition's overall evaluation metric, the cumulative sum of a solver's rank on each instance, the 01LP encodings did not rank high overall but were the top-ranked solvers in a number of cases. Likewise, the INCOP+TOULBAR2 solver was the highest ranked in some cases but ranked in mid-field in many others[5].

The UAI 2014 portfolio solver was highly successful in deciding when to run these solvers or not, achieving first place overall. This independent empirical evaluation supports the findings demonstrated in this dissertation, that significant speedups can be achieved by exploiting various encodings to related languages.

4.8 Chapter Summary

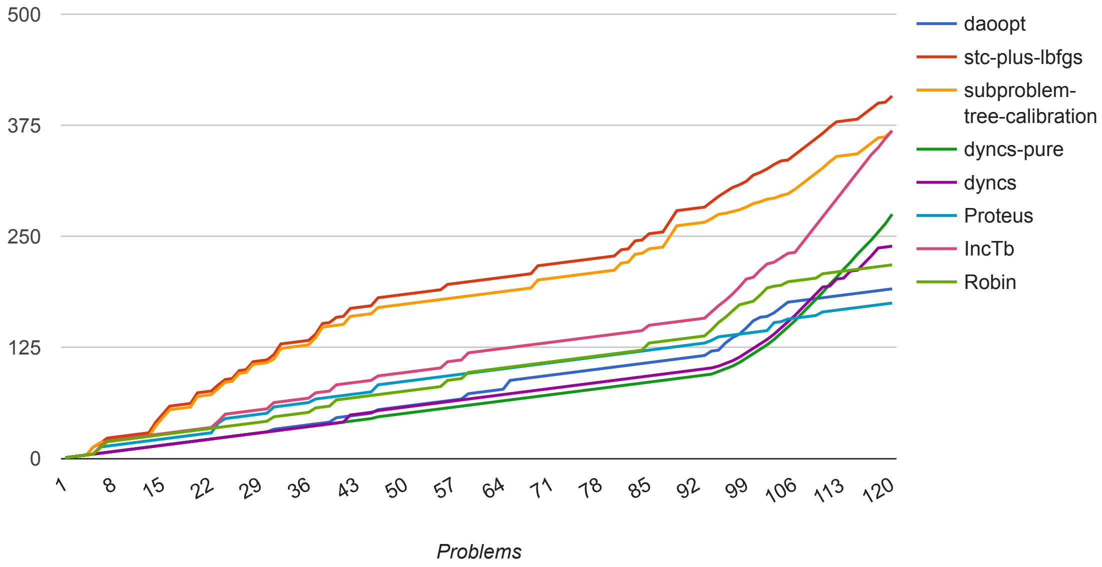
This chapter presented a hierarchical portfolio, named Proteus. The portfolio was first applied to the constraint satisfaction problem, considering a portfolio of CSP solvers, as well as a number of encodings to SAT and subsequently a portfolio of SAT solvers. Detailed empirical evidence across the phase-transition established that it is not sufficient to consider the decisions of which representation or solver to use in isolation, but that they must be considered in tandem.

The hierarchical nature of the portfolio makes it highly extensible with additional encodings and solvers. Moreover, different models and features may be used to make different decisions in the hierarchy, providing greater flexibility to exploit additional knowledge such as features of the chosen representation. We empirically demonstrated the complementary nature of such a portfolio and ultimately its superior performance to that of a portfolio based on a single representation.

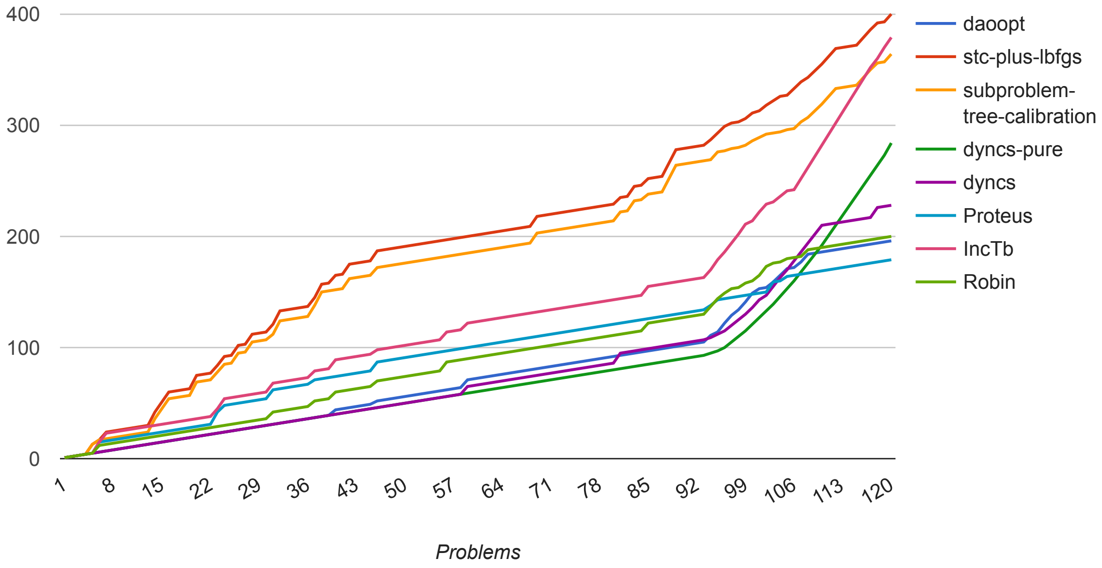
The relationship between CSP and SAT, in terms of empirical performance, was studied in detail. Specifically, the faster paradigm is not necessarily distinguishable by the instance category. We also provided some insight to highlight certain characteristics of the instance which hint at the preferred paradigm.

[4] See MAP/MIP-UAI and MAP/MIP-T-UAI entries.

[5] See MAP/IncTb entry.



(a) 20 minute timeout category.



(b) 60 minute timeout category.

Figure 4.7: Cumulative solver ranking of UAI 2014 MAP solvers. Figures courtesy of `http://www.hlt.utdallas.edu/~vgogate/uai14-competition/`

Proteus was also applied to the domain of graphical models. A number of languages were considered, employing several translations between them, and subsequently a set of solvers. We demonstrated the complementary nature between languages in this domain, and that it can be exploited by a portfolio to achieve significant empirical gains. The effectiveness of such a portfolio was proven through a winning entry to the UAI 2014 Inference Competition.

Chapter 5

Runtime Distributions and Solver Selection

Summary. *This chapter studies the effect of runtime distributions on state-of-the-art portfolio techniques. Underpinning many portfolios is the use of machine learning to predict the performance of its component solvers. Yet the fact that a randomised solver exhibits a runtime distribution on a problem instance has not been considered in this setting. We highlight a fundamental flaw in certain types of related empirical evaluations and make recommendations for a more holistic approach.*

5.1 Introduction

Modern combinatorial search solvers contain many elements that are stochastic in nature, such as randomised variable and value selection, tie-breaking in heuristics, and random restarting [65, 66]. These elements add a degree of robustness to the solver, by helping it to avoid worst-case behaviour. Nevertheless, this stochasticity results in variations in runtime between repeated runs of the solver on a problem instance, a fact which is often not considered. This chapter illustrates the consequences that such assumptions can have when comparing solver performance.

In the field of solver portfolios [67, 106], which has consolidated and advanced the state-of-the-art across a broad range of problem domains such as Constraint

Programming (CP) [89, 128], Satisfiability (SAT) [99, 165], MaxSAT [12], Planning [144], and many more. The effectiveness of a portfolio relies on their ability to identify the best solver for each individual instance. At its core, training a portfolio involves collecting the runtime performance of each of its constituent solvers on a collection of benchmark instances [67, 106]. From this, machine learning techniques are used to choose amongst the solvers based on instance features. Existing work in this area has only considered a single sample of the solver runtime for each instance, and makes the assumption that the behaviour is consistent over repeated runs of the solver. However, as we will see, the performance of a solver can vary by many orders of magnitude between repeated runs. This behaviour is related to fat- and heavy-tailed distributions [66, 69, 71].

In a related field, automated solver configuration tools [11, 47, 95] have successfully been able to tune solvers to surpass their default configuration. Their effectiveness relies on obtaining consistent, representative performance data for a given parameterisation on a sample of input instances. Crucially, these methods will not typically make allowances for any stochastic behaviour in the underlying solvers.

Related to the above is the task of solver runtime prediction [96]. This focuses on building an empirical hardness model to predict the performance of a solver based on instance-specific features. Such models often form the basis of the portfolio and configuration techniques previously mentioned.

However the fact that a solver can exhibit a runtime distribution that does not have finite mean and/or variance has largely been ignored in each of these fields. Instead the typical approach is to characterise the behaviour by taking a single sample runtime. This chapter demonstrates that state-of-the-art runtime prediction methods do not account for the stochastic aspect of solver runtime and, as such, can have a substantial effect on their accuracy.

5.2 Background

This section discusses some studies of runtime distributions relevant to the current chapter.

A large body of work, particularly in the CP and SAT communities, has studied runtime variations when running deterministic backtracking algorithms on distributions of random instances, but also by repeated runs of randomised backtracking algorithms on a individual instances. The seminal work of Gomes et al. modelled such performance variations under heavy- and fat-tailed distributions [69, 71]. Unlike standard distributions such as normal, exponential, and Weibull that have exponentially decreasing tails, the heavy- and fat-tailed distributions have a considerable probability mass in their tails. Their tails follow a power-law decay, namely tails of the Pareto–Lévy form:

$$P(X > x) \sim Cx^{-\alpha}, \quad x > 0$$

where $0 < \alpha < 2$ and $C > 0$ are constants. When $1 < \alpha < 2$, the distribution has a finite mean but infinite variance. When $\alpha \leq 1$, the distribution has neither a finite mean nor a finite variance. In practice, even though the search space may be exponentially large, it is in fact finite. The effect of this disparity was studied by Gomes through a comparison to a bounded heavy-tail distribution. Such a distribution exhibited a very similar power-law decay over an extensive portion of the tail, but exhibits a sudden drop-off further out in the tail [65]. They concluded that one must sacrifice some small inaccuracies in calculations, such as in the index of stability. Nevertheless, the fat- and heavy-tailed distributions have proved valuable as models of solver performance in a number of studies. In essence, these statistical models convey the non-negligible risk of extremely long runs for the solver to solve the instance.

A number of techniques have been proposed as counter-measures to avoid such worst-case behaviour. Randomised restarting randomly restarts the search after a specified number of backtracking failures, or nodes, and has been shown to eliminate heavy-tailed behaviour [69]. Restarting is now common-place in modern state-of-the-art solvers [66]. Gomes et al. demonstrated that an exponential distribution for the depth of inconsistent subtrees corresponded to the presence of heavy-tailed behaviour in the overall search method [72].

The Handbook of Satisfiability dedicates a chapter to runtime variations in complete solvers, modelling them as complex physical phenomenon [66]. Solver performance is again described in terms of fat- and heavy-tailed distributions. The chapter focuses on an analysis and prevention of extreme variations using the framework of fat- and heavy-tailed distributions. However, no consideration is given to less extreme runtime distributions, nor to their effect on empirical comparisons.

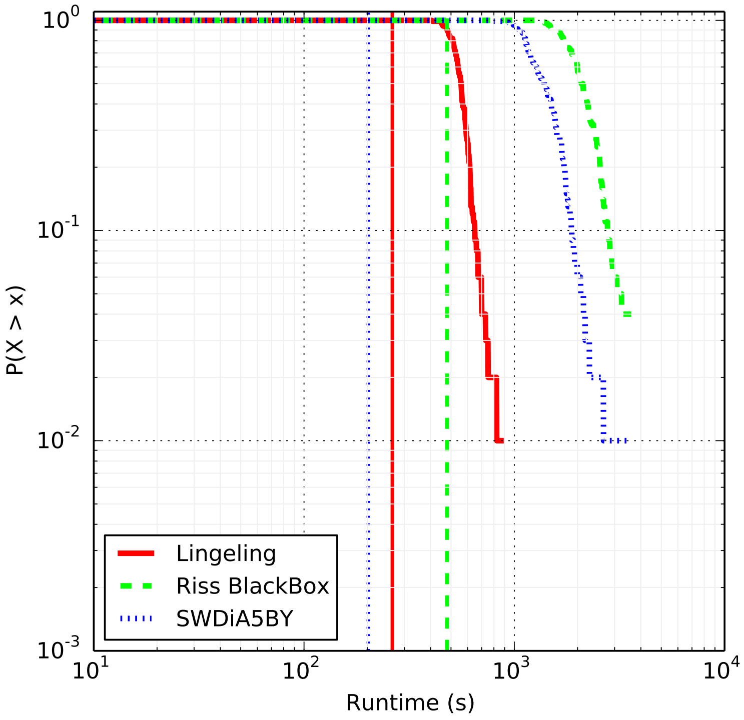
While running one of the SAT solver competitions, Le Berre and Simon accidentally observed the so called *lisa syndrome*, whereby dramatically different performances were observed for many solvers on two randomly shuffled versions of the same instance [110]. Even an early version of the SATzilla portfolio solver was shown to have given two majorly different runtimes on the same shuffled instance [110].

The papers on heavy-tailed runtime distributions referenced previously were written during a time before randomised restarting was common and the fastest complete-search algorithms for SAT were based on the DPLL method. Modern SAT and CP solvers have come a long way since, with runtime distribution studies contributing largely to advances [66]. In most cases, the aforementioned papers also modified the algorithms to add an element of randomisation and a new heuristic equivalence parameter to increase its frequency. The solvers considered in this dissertation have not been modified in any way, current state-of-the-art solvers are used. Nevertheless, it is evident that modern solvers have an element of stochasticity built in, resulting in runtime distributions with significant variance. Undoubtedly, it is time such studies are revisited given the considerable advances in solver technology over the last decade.

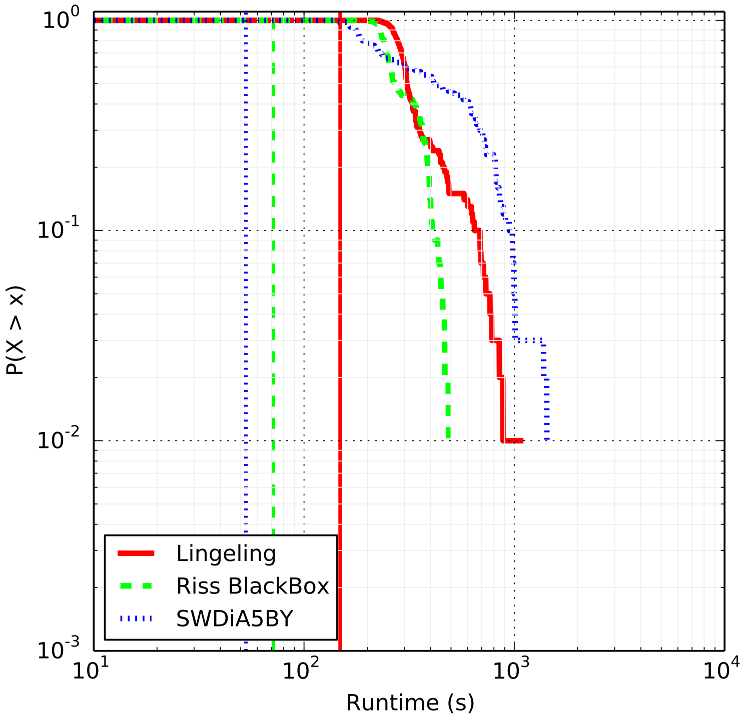
5.3 Runtime Variation in Empirical Comparisons

Much of the work mentioned in Section 5.2 studies the heavy-tailed behaviour of a solver's runtime, with the motivation of improve search algorithms. However, little consideration has been given to the effect of such runtime variations in empirical comparisons of solvers. The typical methodology for evaluating deterministic solvers in empirical comparisons such as the the CSP, SAT, MAX-SAT competitions is to take a single statistic of each solver's runtime on a collection of benchmarks. This may be a single sample of the runtime, or an average over multiple runs. This section illustrates a deficiency in such a methodology, showing that a more holistic view needs to be taken.

Figure 5.1 shows the runtime distribution of the top three solvers [8, 22, 127] on two industrial instances from the 2014 SAT Competition. Vertical lines mark the solver's observed runtime during the competition, and the other lines plot the survival function of the empirical runtime distribution over 100 runs. In Figure 5.1(a), the solver SWDiA5BY solved the instance faster than Lingeling during the competition, however we can clearly deduce from the runtime distribution that over repeated runs there is a larger probability of SWDiA5BY having a



(a) SAT Competition 2014 instance UCG-15-10p0.cnf



(b) SAT Competition 2014 instance q_query_3_L90_coli.sat.cnf

Figure 5.1: Log-log plots showing the empirical runtime distribution of two instances from the application category at the 2014 SAT Competition. The three solvers were the top-ranked from the competition. The vertical lines mark their observed runtime during the competition.

longer runtime than Lingeling. In this instance, the runtime distribution shows a clear ranking of the solvers: Lingeling $\prec$ SWDiA5BY $\prec$ Riss BlackBox. In Figure 5.1(b) the distinction between solvers is not as pronounced in the tail, but we could compare these runtime distributions in several ways. Interestingly, in the competition, SWDiA5BY is ranked as the fastest of the three solvers on this instance, however when you consider its runtime distribution, it is most likely to take the longest.

This exemplifies a critical flaw in this type of empirical evaluation that is often performed when comparing solvers, only a single statistic, normally a single runtime or an average, of the solver's runtime is taken on each benchmark instance. The current evaluation metric used in the SAT Competition purely considers the number of instances solved within a specified timeout, and only uses runtimes in the case of ties. However, the runtime distribution can often span over the timeout value, thus affecting the number of instances solved. Section 5.5 will demonstrate the ramifications of this, showing that these three solvers could have been ranked in any permutation. However, first, Section 5.4 will present a more holistic, statistically-founded methodology which is lacking in current comparisons.

5.4 Expected Performance

The performance of a solver on a set of benchmark instances can be modelled as a random variable, doing so enables statistical bounds to be defined on the possible outcomes. Let p_i be the probability that instance i is solved within a given timeout. p_i will be simply 0.0 or 1.0 if a single sample runtime is taken, or a continuous value if a solver is run multiple times on the same instance. Naturally, the accuracy of p_i is proportional to the number of repeated runs performed.

Next, let X_i be a discrete random variable that takes the value 1 if instance i is solved within the timeout, and 0 otherwise. The expected value of X_i is p_i:

$$E[X_i] = \Pr(X_i = 0) \times 0 + \Pr(X_i = 1) \times 1 \equiv p_i \quad . \tag{5.1}$$

We can define a bound on the variation of X_i by taking the critical value for $p_i = 0.5$; thus the standard deviation of X_i can be bounded by $\frac{1}{2}$:

$$\begin{aligned}
\sigma^2(X_i) = {} & E[X_i^2] & - & \ (E[X_i])^2 \\
= {} & E[X_i] & - & \ p_i^2 \\
= {} & p_i & - & \ p_i^2 \\
= {} & & p_i(1-p_i) & \\
\leq {} & \tfrac{1}{4} & & \text{for any } p_i
\end{aligned} \tag{5.2}$$

$$\sigma(X_i) = \sqrt{p_i(1-p_i)} \leq \frac{1}{2} \tag{5.3}$$

If the benchmark set contains k instances, then the success rate S, or ratio of instances solved within the timeout, is simply the mean of the X_i variables over the k instances. This allows the definition of the expected value of S as being the mean over the p_i values:

$$E[S] = \frac{\sum_{i=1}^{k} E[X_i]}{k} = \frac{\sum_{i=1}^{k} p_i}{k} \tag{5.4}$$

Finally, bounds on the variation and standard deviation of S can be defined:

$$\begin{aligned}
\sigma^2(S) = {} & \sigma^2\left(\frac{\sum_{i=1}^{k} X_i}{k}\right) = \frac{1}{k^2}\left(\sum_{i=1}^{k} \sigma^2(X_i)\right) \\
= {} & \frac{1}{k^2}\left(\sum_{i=1}^{k} p_i(1-p_i)\right) \leq \frac{k}{4k^2} = \frac{1}{4k}
\end{aligned} \tag{5.5}$$

$$\sigma(S) \leq \frac{1}{2\sqrt{k}} \tag{5.6}$$

Importantly, this means that even over repeated runs, the variation in the total number of instances solved is bounded by a function of the number of instances. This fact should be of great consequence to the designers of empirical comparisons.

5.5 SAT Competition 2014

This section will project the analysis of Section 5.4 onto one annual empirical solver comparison. Consider the top three solvers from the application category at 2014 SAT Competition, namely, *Lingeling* [22], *SWDiA5BY5by* [127], and *Riss BlackBox* [8] which solved 231, 228, and 226 of the 300 instances in this

Table 5.1: Number of instances solved out of 300 by the top-3 solvers in the application category of the 2014 SAT Competition along with the upper-bound on the deviation from these values if the competition was rerun.

Solver	#Solved	S	$\overline{\delta(S)}$	95% bounds
Lingeling	231	0.770	0.0289	214..248
SWDiA5BY5by	228	0.760	0.0289	211..245
Riss BlackBox	226	0.753	0.0289	209..243

category, respectively. Note that, like many other empirical competitions, only a single sample of each solver's runtime is taken during the evaluation and the scoring metric is purely based on the number of instances solved within the specified timeout. Using the equations of Section 5.4, we can postulate on possible outcomes and give upper-bounds on the variation that could occur in such empirical evaluations, this is presented in Table 5.1.

Given that the performance difference amongst the top-ranked solvers is so narrow, it is not unreasonable to suspect that the outcome could have been different if the competition was re-run. To assess the potential outcomes, each of the three solvers above were re-run 100 times on each of the 300 instances from the industrial/application category at the SAT Competition 2014. Since Lingeling is the only one of these solvers to support the ability to pass a random seed as input, we emulate the functionality across all solvers simply by shuffling the input instance, a process which is described in detail in Section 5.6.2.1.

Performance data was collected on a cluster of Intel Xeon E5430 2.66Ghz processors running CentOS 6. The solvers were limited to 1 hour and 2GB of RAM per instance. All times are reported in CPU-time. Note that these experiments are run on different hardware to that used in the SAT Competition and used a shorter timeout due to the increased computational complexity. In total, 322 CPU-weeks were used to record the data.

Considering the observed runtime distributions across 100 runs, together with the analysis from Section 5.4, we can quantify more statistically well-founded outcomes to the SAT Competition; these are presented in Table 5.2. Additionally, we can derive 95% confidence bounds on the expected success rate if we assume it to be normally distributed. Significantly, the 95% confidence bounds of the solvers overlap, and as such, any permutation of the ordering could be possible.

Figure 5.2 augments these results by visualising the distribution of the total number of instances solved by each solver across the runs. Ordering by probability

Table 5.2: Expected success rates over multiple runs on 300 application instances from the SAT Competition 2014.

Solver	$E[S]$	$\delta(S)$	95% bounds
Lingeling	0.553	3.6	159..173
Riss BlackBox	0.530	3.3	152..165
SWDiA5BY5by	0.571	2.6	166..176

mass would rank the solvers SWDiA5BY $\prec$ Lingeling $\prec$ Riss BlackBox, consistent with the ranking from Table 5.2. Interestingly, this is a different ranking to that of the official competition. The histograms appear, at least visually, to be normally distributed. Fitting a normal distribution, over 100 samples, gives confidence values of 64%, 47%, and 19% for Lingeling, Riss BlackBox, and SWDiA5BY respectively. The fitted distributions are overlaid in Figure 5.2. Studying these distributions in more detail, possibly over a larger number of samples, would be an interesting avenue for further investigation.

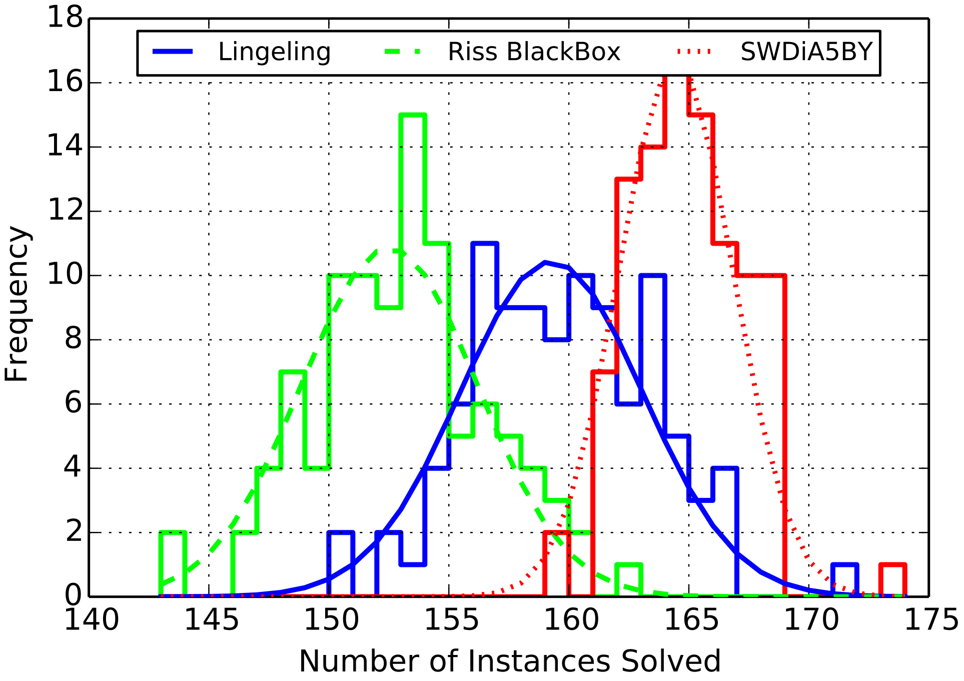


Figure 5.2: A histogram of the number of instances solved across 100 runs on 300 instances from the application category of the SAT Competition 2014. The three solvers are the top-ranked solvers from the competition.

Table 5.3: Summary of the runtime variations of three solvers on a set of sample instances from the 2014 SAT Competition. The 'Comp.' column contains the runtime observed in the competition, AUC is the area under the survival function of the empirical distribution. Bold entries mark instances where there is at least one order of magnitude between the best & worst observed runtime.

	Instance Name	Solver	#runs	Median	Mean	Std.dev	Min	Max	Comp.	AUC
15	**006-23-80.cnf**	**Lingeling**	**100**	**3600.0**	**3509.6**	**438.3**	**63.5**	**3600.0**	**5000.0**	**3509.6**
15	006-23-80.cnf	Riss BlackBox	100	3600.0	3600.0	0.0	3600.0	3600.0	5000.0	3600.0
15	006-23-80.cnf	SWDiA5BY	100	3600.0	3600.0	0.0	3600.0	3600.0	5000.0	3600.0
21	**008-80-4.cnf**	**Lingeling**	**100**	**1274.4**	**1752.1**	**1264.6**	**142.0**	**3600.0**	**621.0**	**1752.5**
21	**008-80-4.cnf**	**Riss BlackBox**	**100**	**808.8**	**818.9**	**382.3**	**85.3**	**3600.0**	**794.0**	**819.4**
21	**008-80-4.cnf**	**SWDiA5BY**	**100**	**209.7**	**202.7**	**78.2**	**10.5**	**359.6**	**85.4**	**203.2**
53	**6s168-opt.cnf**	**Lingeling**	**100**	**260.5**	**473.4**	**622.6**	**26.6**	**3600.0**	**370.8**	**473.9**
53	6s168-opt.cnf	Riss BlackBox	100	13.1	14.2	4.5	7.4	36.8	17.5	14.6
53	6s168-opt.cnf	SWDiA5BY	100	83.9	87.0	21.8	51.9	162.5	56.7	87.4
117	UCG-15-10p0.cnf	Lingeling	100	556.7	562.9	75.0	404.6	863.9	263.2	563.4
117	UCG-15-10p0.cnf	Riss BlackBox	100	2117.1	2165.9	510.9	1290.1	3600.0	478.4	2166.5
117	UCG-15-10p0.cnf	SWDiA5BY	100	1385.2	1445.4	402.9	757.4	3600.0	202.4	1445.9
258	q_query_3_... li.sat.cnf	Lingeling	100	307.8	379.6	165.1	226.7	1071.9	148.6	380.0
258	q_query_3_... li.sat.cnf	Riss BlackBox	100	268.7	308.5	78.5	198.2	515.1	71.5	308.9
258	**q_query_3_... li.sat.cnf**	**SWDiA5BY**	**100**	**422.3**	**507.9**	**320.6**	**141.3**	**1487.7**	**52.9**	**508.4**
265	rpoc_xits_15_SAT.cnf	Lingeling	100	7.3	5.5	2.9	1.6	11.9	1.9	6.0
265	**rpoc_xits_15_SAT.cnf**	**Riss BlackBox**	**100**	**12.1**	**59.7**	**359.1**	**4.9**	**3600.0**	**3.6**	**60.2**
265	rpoc_xits_15_SAT.cnf	SWDiA5BY	100	5.3	5.8	2.0	2.1	11.7	1.1	6.3
271	**stable-400... 140011.cnf**	**Lingeling**	**100**	**3600.0**	**3458.7**	**568.3**	**125.6**	**3600.0**	**5000.0**	**3458.7**
271	**stable-400... 140011.cnf**	**Riss BlackBox**	**100**	**3600.0**	**3482.0**	**532.4**	**133.6**	**3600.0**	**5000.0**	**3482.0**
271	**stable-400... 140011.cnf**	**SWDiA5BY**	**100**	**3600.0**	**2577.2**	**1346.1**	**78.3**	**3600.0**	**5000.0**	**2577.4**
287	**vmpc_32.re... 5-1919.cnf**	**Lingeling**	**100**	**3600.0**	**2929.7**	**1113.6**	**19.3**	**3600.0**	**5000.0**	**2929.8**
287	**vmpc_32.re... 5-1919.cnf**	**Riss BlackBox**	**100**	**1445.9**	**1831.9**	**1365.0**	**17.2**	**3600.0**	**1931.7**	**1832.2**
287	**vmpc_32.re... 5-1919.cnf**	**SWDiA5BY**	**100**	**2533.8**	**2187.1**	**1488.7**	**3.1**	**3600.0**	**149.5**	**2187.4**

Table 5.3 presents some more detailed statistics about the performance of the three solvers, only a sample of the 300 instances is presented due to space constraints. We list the runtime which was observed in the SAT Competition under the 'Comp.' column. Notice that for numerous instances where a solver recorded a timeout in the SAT Competition we also observe many timeouts across the runs, but critically, the solver may also solve the instance very quickly.

We observe dramatic variations in runtime, often more than three orders of magnitude between a solver's best and worse time on an instance. Note that three orders of magnitude covers the entire range of the specified time limit, thus, larger variations might have been possible had a longer timeout been used. In numerous cases, on a single instance, a solver will timeout on a large proportion of the runs but conversely may solve the instance very quickly in other cases. This could be the critical difference between solved or unsolved in the competition setting. Given that the difference amongst the top ranked solvers in the SAT Competitions is typically only a handful of instances, this constitutes a significant observation.

5.6 State-of-the-art Runtime Prediction

The ability to predict the runtime of a solver is interesting from a number of perspectives. Solver designers can make use of it as an informative analytical tool and it frequently serves as a central component in solver portfolios and configuration tools. Often, these tools base their decisions on underlying runtime prediction models. The prediction target is the runtime of the solver on an instance, which has conventionally been collected by recording a single statistic from the runtime distribution [96]. However this approach is flawed, since in reality we are dealing with highly variable runtime distribution, a single value is simply not representative. This section highlights the detrimental effects that this fundamental flaw can have on state-of-the-art runtime prediction methods.

5.6.1 Log Transformation

The typical practice when predicting solver runtime is to log-transform the runtime in seconds [96]. Error calculations on this, like root-mean-squared error and mean absolute error, will over-penalise runtimes of less than one second. Differences between runtimes of 0.01 seconds and 0.1 seconds are valued to the same extent as the difference between 10 and 100 seconds. These are all an order of magnitude different, but we argue that the first order of magnitude for runtimes should range from [0..9) seconds, the second from [9..99), and so on. Thus, the runtimes should be transformed as log(1 + runtime). This additionally eliminates the need to censor runtimes of zero seconds which would have been needed when performing the log transformation. For the predictions in this section we will use the adjusted log-transformed runtime, $\log_{10}(1 + \text{runtime})$.

5.6.2 Experimental Setup

The benchmark set comprises 1676 industrial instances from 9 years of the SAT Competitions, Races, and Challenges between 2002 and 2011. We use MiniSat 2.0 [43] as the solver with a timeout of 1 hour and a limit of 2GB RAM. Performance data was collected on a cluster of Intel Xeon E5430 Processors (2.66GHz) running CentOS 6. A total of 315 weeks of CPU-time was consumed to accumulate this performance data.[1] Table 5.4 presents a sample summary of the runtimes.

5.6.2.1 Instance Shuffling

Two intuitive options that can be used to affect the stochastic decisions made by the solver are to pass a seed for the random number generator, or to shuffle the input instance. Both options will lead to different search trees being explored by the solver. Only a limited number of solvers support the ability to pass a random seed as input. Therefore the experiments in this chapter opt to pass a randomly shuffled version of the instance to all solvers for consistency.

Shuffling a SAT instance requires randomly renaming the variables, shuffling their order in each clause, and reordering the clauses. This preserves the same structure and semantics of the original instance but may result in the algorithm taking different paths in the solving process across runs. This is because solvers often resort to lexicographically ordering variables and clauses. Note that it is common in the SAT Competition to re-use instances from previous years by performing this type of shuffling [110].

[1] The dataset is available at `http://ucc.insight-centre.org/bhurley/`

Table 5.4: Samples of the runtime variations of MiniSat on a set of sample industrial category instances from the SAT Competitions, Races, and Challenges between 2002 and 2011. The AIJ'14 column contains the runtime observed in [96], AUC is the area under the survival function for the empirical distribution.

Instance Name	#runs	Median	Mean	Std.dev	Min	Max	AIJ'14	AUC
dated-10-13-s	100	82.0	337.4	675.3	7.5	3600.0	12.6	337.9
fvp-unsat	100	94.4	1294.5	1642.6	8.0	3600.0	3600.0	1294.8
gri. . . s05-34	100	3600.0	2963.2	1189.9	9.6	3600.0	3600.0	2963.3
gss-23-s100	100	3600.0	3542.7	310.8	1243.9	3600.0	3600.0	3542.7
k2fix_gr. . . w8	100	3600.0	3394.9	818.5	5.6	3600.0	3600.0	3394.9
mizh-md5-47-4	100	200.3	283.3	254.0	18.8	1458.7	595.2	283.8
q_que. . . coli	100	609.4	763.3	522.6	129.6	2236.2	139.0	763.8
rbc. . . SAT	100	3.7	50.6	373.1	1.1	3600.0	1.9	51.1
shuff. . . t04-461	100	6.2	1455.6	1642.4	3.3	3600.0	6.1	1456.0
uts. . . unknown	100	234.7	276.4	173.0	109.0	1546.4	228.9	276.9
vel. . . at-3.0-b18	100	34.9	65.3	69.7	16.4	372.3	6.2	65.8
vel. . . sat-1.0-03	100	127.4	153.6	92.6	52.9	628.4	71.7	154.1
vmpc_31	100	2179.5	2080.3	1395.9	2.9	3600.0	3600.0	2080.7
vmpc_33	100	2591.9	2244.0	1311.2	15.6	3600.0	305.8	2244.3

5.6.3 Prediction Fragility

To perform the runtime prediction we use a random forest with the same parameters as [96], that is using 10 regression trees, using half of the variables as split variables at each node ($perc = 0.5$), and a maximum of 5 data-points in leaf nodes ($n_{min} = 5$). This configuration was shown to be the most accurate for runtime prediction in comparison to ridge regression, neural networks, Gaussian processes, and individual regression trees. The random forest may choose from a set of 138 SAT features [166] which have been used for runtime prediction and many award winning portfolios. As per [96] and other work on runtime prediction, we do not treat runs that timeout with any speciality, they are considered to just have taken the timeout value and not penalised in any particular way. We use a standard randomised 10-fold cross-validation, where the dataset is split in 10 folds. Each of the folds takes a turn as the test set, with a model being built from the remaining 9 folds of training data. We use a standard error metric for regression, namely, the root-mean-squared error (RMSE) from the predicted value to the true test value.

Some simple statistical values are extracted from the runtime distribution of each instance in order to model a range of scenarios that may arise using the current methodology, where it is possible for the models to be trained on completely different runtime samples to that considered in testing. Table 5.5 presents a set of these scenarios, highlighting the fragility in current state-of-the-art runtime

Table 5.5: Mean RMSE on log(1 + runtime) predictions across 10-fold cross-validation, lower is better. Comparison of training and testing on different samples of each instance's observed runtimes. 'sample' is simply a single random sample from the observed runtimes for an instance. For each column, values in bold represent the best mean RMSE, italicised values are statistically significantly different from this best.

		Test				
		sample	min	median	mean	max
Train	sample	0.698	*0.812*	0.683	*0.659*	*0.723*
	min	*0.812*	**0.611**	*0.816*	*0.832*	*0.987*
	median	**0.675**	*0.820*	**0.659**	*0.647*	*0.725*
	mean	*0.681*	*0.848*	*0.667*	**0.637**	*0.700*
	max	*0.782*	*1.010*	*0.755*	*0.702*	**0.663**

prediction methods. Columns vary the target metric and rows vary the training sample. Within each column, bold values highlight the best mean RMSE, values in italics are those which produced predictions statistically significantly different (95% confidence) from this best.

The closest setting to typical methodology is shown in the *sample* entries, this simply uses a random sample of the observed runtimes. The predictions produced by these models are reasonable but are not the most accurate. Unsurprisingly, the most accurate predictions typically arise when we are using the same sample statistic for both training and testing. For example, when aiming to predict the median runtime it is best to have used the median runtime of the training set instances, likewise for predicting the minimum and maximum runtimes.

Conversely, another setting, albeit the most extreme, is if the training data consists of the minimum value from each distribution and we are attempting to predict the maximum runtime. Likewise for predicting the minimum runtime where the maximum is used as training. These settings give the largest error in predictions, with RMSE of 1.0.

It is clear, and should seem natural, that the origin of the training sample has a considerable impact on the outcome of the models, but surprisingly, this is not something that is considered in existing methodology.

5.7 Runtime Variation Analysis

This section presents some further analysis of the runtime variations observed in this chapter. Figure 5.3 summarises the runtime variation across the 1676 benchmark instances catalogued in Section 5.6.2. The figure presents a histogram of the standard deviation of the runtime, on a log scale, for MiniSat across repeated runs.

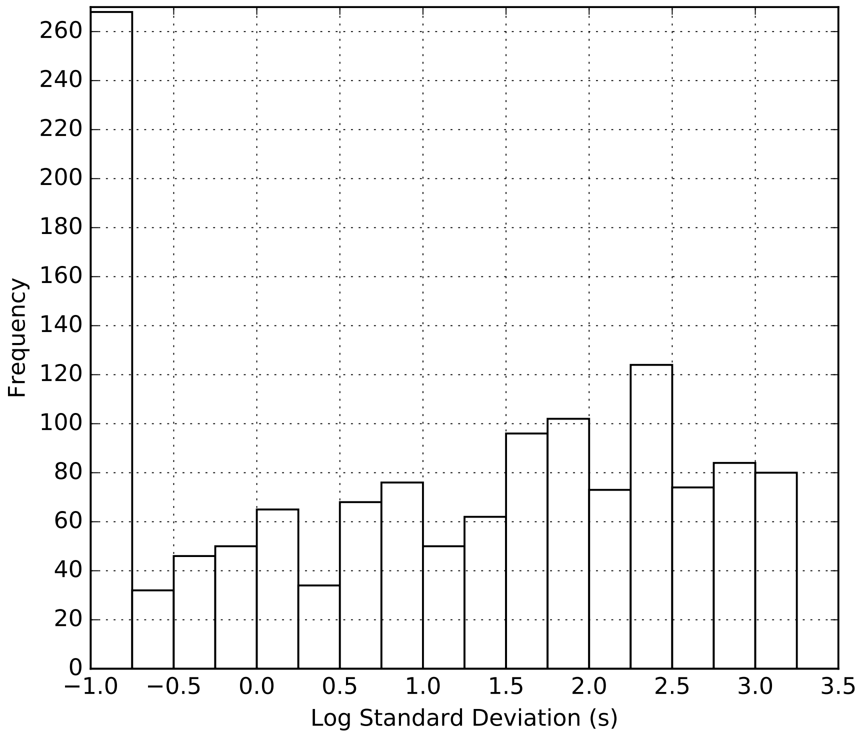
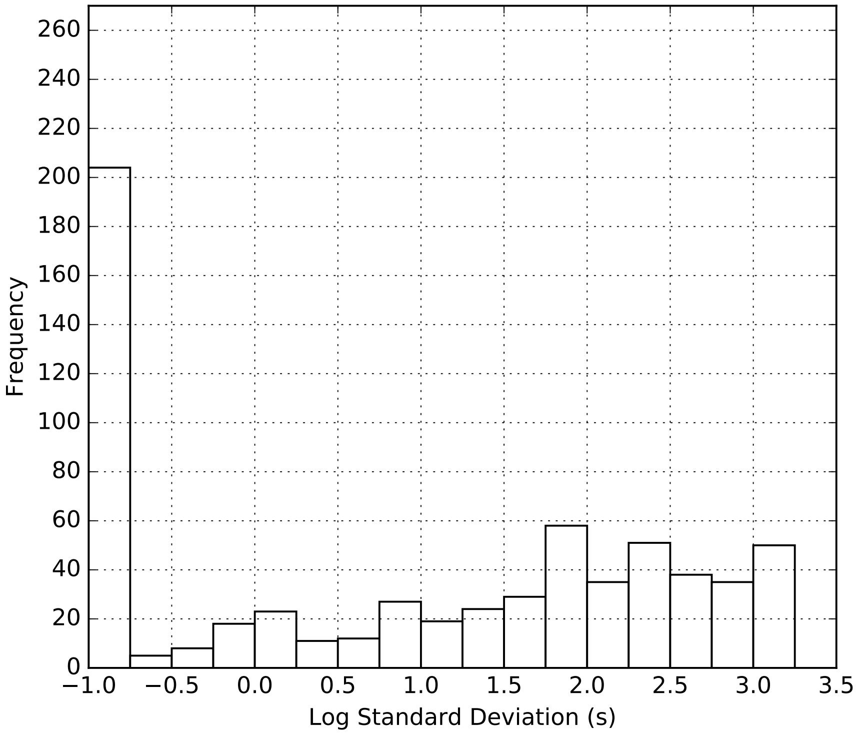


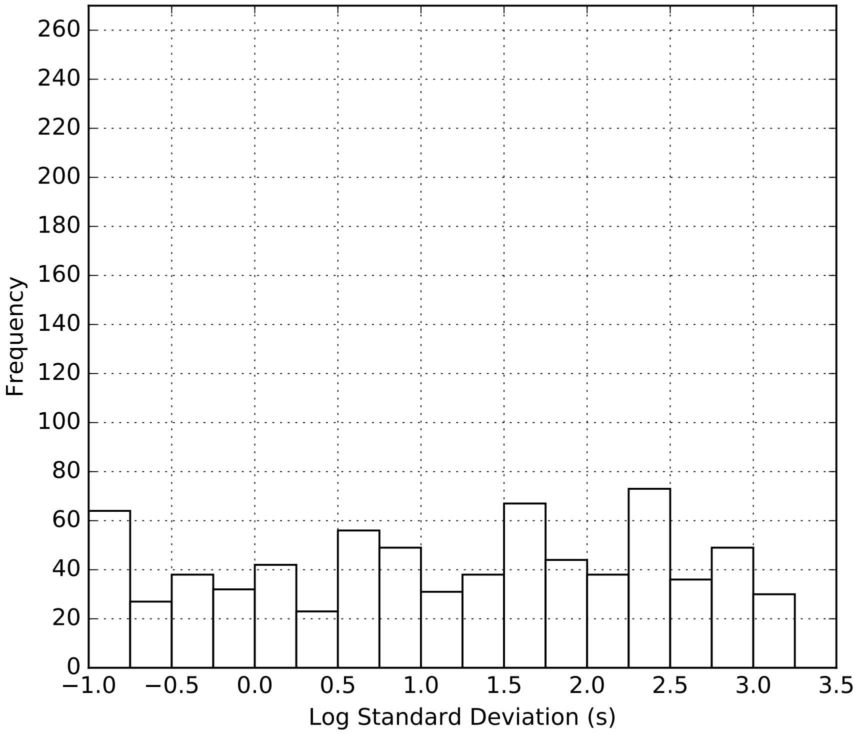
Figure 5.3: Histogram of runtime standard deviations.

Firstly, 268 instances have a standard deviation of less than 0.1 second. These are instances for which the solver produced a highly consistent behaviour across repeated runs. Additionally, 292 instances were not solved within the specified timeout across any run and are excluded from this figure. Nevertheless, throughout the centre of the plot, we can see that many instances have large deviations, often up to 3 orders of magnitude. Furthermore, note that these figure are right-censored due to the imposed timeout of 1 hour. Thus, in certain cases these under-represent the true standard deviation which would be measured if the runs had been run to completion.

Figure 5.4 divides this analysis between satisfiable and unsatisfiable instances. 647 (resp. 737) of the 1676 instances were satisfiable (resp. unsatisfaible), with an additional 292 unsolved. Of the 268 instances with a standard deviation less



(a) Satisfiable instances.



(b) Unsatisfiable instances.

Figure 5.4: Histogram of runtime standard deviations, broken out by satisfiability.

than 0.1 second, 204 are satisfiable compared to just 64 unsatisfiable instances. Additionally, there are few satisfiable instances which elicit a log standard deviation in the range -0.5 to 1.5. The runtime of satisfiable instances tends to either have a very narrow, consistent behaviour, or have a very large standard deviation; whereas unsatisfiable instances tend to have a more even mixture of small to large deviations. This is consistent with the fact that a heavy-tailed distribution is never elicited by a solver from unsatisfiable instances. We note that this analysis may be different if a different solver is used, for example one which is tuned for unsatisfiable instances.

Instance Category Analysis

This section illustrates and compares the runtime variations across multiple solvers by grouping instances into their respective categories. We demonstrate that the existence of large runtime variations between repeated runs is not limited to a small set of instances or specific types of instances, rather it occurs across many categories. Additionally, the behaviour is not limited to a single solver, but is evident in each of the solvers we considered.

Figure 5.5 plots a comparison of the runtimes among the three solvers of the dataset presented in Section 5.5, where each instance is grouped into its respective category. The boxplots illustrate the range of observed runtimes. The red horizontal line marks the median value, the central box ranges from the first to the third quartile, and its whiskers extend to the fifth and ninety-fifth percentiles.

Firstly, note that the majority of the runtimes are towards the higher end of the scale, closer to the timeout. This may be due, in part, to the selection mechanism used by competition organisers to select instances that are sufficiently hard, but not too hard such that no solver is able to solve it. In nearly all categories, every solver encounters timeouts on many of the runs.

For some instance categories the range of runtimes is very narrow, for example *fpga-routing* and *hardware-bmc*, but these categories constitute a small number of instances. In other cases, the typical behaviour is that the majority of a solver's runtimes span one order of magnitude. For the larger categories of instances, the runtimes consistently range across three orders of magnitude for each solver. For example, with the *argumentaion* instances, Lingeling's performance spans a relatively tighter range than both Riss BlackBox and SWDiA5BY. Lingeling's median runtime is a timeout on these instances but it does also solve the instances

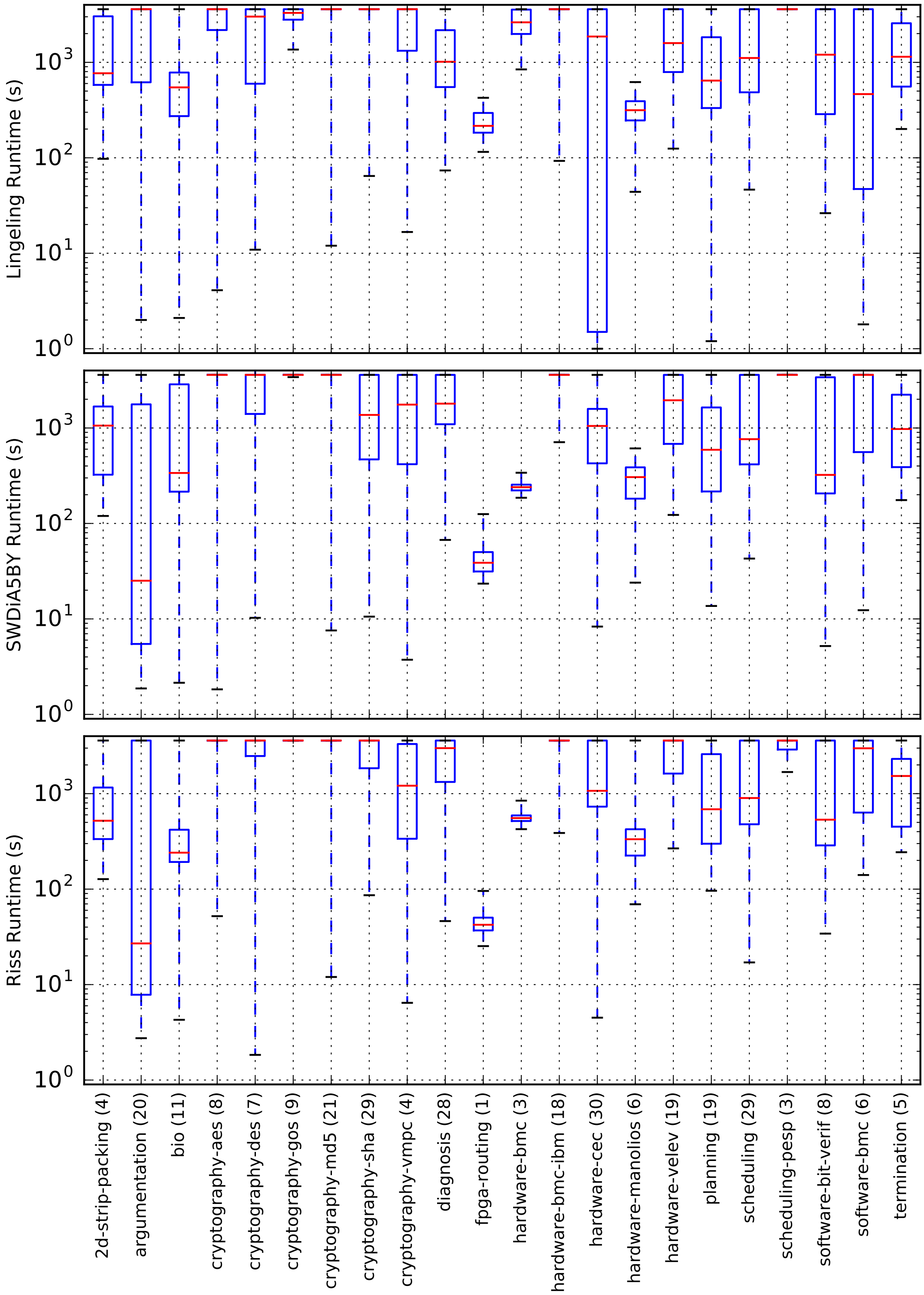


Figure 5.5: Boxplot comparison of the runtime variations between solvers on 300 SAT instances. Instances are grouped by their category, values in brackets correspond to the number of instances in the category, each of which has been run 100 times.

in a matter of seconds on rare occasions. This is not the case for Riss BlackBox nor SWDiA5BY as their median runtime is much lower, with their second and third quartiles spanning a larger range.

In summary, the existence of large runtime variations between repeated runs is not limited to a small set of instances or specific types of instances, rather it occurs across many categories. Additionally, the behaviour is not limited to a single solver, but is evident in each of the solvers we considered. In particular, note that these are all of the industrial instances from the most recent SAT Competition, using three state-of-the-art solvers which have not been modified in any way. Thus, it seems prudent that a more holistic view should be taken in empirical evaluations of solvers, portfolios, configuration, and related areas.

5.8 Related Work

A large body of work in the CSP and SAT communities has studied runtime variations when running deterministic backtracking algorithms on distributions of random instances, but also by repeated runs of randomised backtracking algorithms on individual instances. The Handbook of Satisfiability dedicates a chapter [66] to runtime variations in complete solvers, modelling them as complex physical phenomena. The chapter focuses on an analysis and prevention of extreme variations using the framework of fat- and heavy-tailed distributions.

Gomes et al. first presented the heavy- and fat-tailed phenomenon in solver runtime [71]. That paper was written before randomised restarting was common and the fastest complete-search solvers for SAT were based on the DPLL method; modern SAT solvers have come a long way since. The paper also modified the algorithms to add an element of randomisation and a new heuristic equivalence parameter to increase the frequency of random tie-breaking. Our work has not modified the solvers in any way, we use the current state-of-the-art performers. Nevertheless, it is evident that modern solvers have an element of stochasticity built in, resulting in runtime distributions with significant variance.

Nikolić proposed a methodology for statistically comparing two solvers given observations from their runtime distributions [126]. This chapter provides substantial additional evidence to support the need for such comparisons. Van Gelder proposes a new *careful ranking* method for competitions which addresses a number of issues with existing ranking methods, but the system is still based on individual runtime samples [156].

Hutter et al. presents the state-of-the-art for solver runtime prediction, evaluating the effectiveness of a number of machine learning methods for runtime prediction across a wide range of SAT, MIP, and TSP benchmarks [96]. Our comparison in Section 5.6 does not compare different models on their accuracy, but does question a fundamental assumption being made about the underlying runtime behaviour.

Fawcett et al. tackles runtime predictions for state-of-the-art planners where they observe substantial runtime variations of planners across different domains [46]. The authors propose new feature sets which improve the accuracy of predictions. However the work does not consider runtime variations on individual instances, only a single sample runtime is taken. Leyton-Brown et al. identified features of combinatorial auction problems to predict the runtime of CPLEX on different distributions of instances, but only one sample runtime is taken despite elements of randomness in the solver [112].

5.9 Conclusions and Discussion

This chapter questions a fundamental assumption that is made about the runtime behaviour of complete search solvers. Modern solvers usually have some form of inbuilt randomness. As a consequence their runtime can exhibit significant variation, sometimes by orders of magnitudes on individual instances. We have shown that the outcome of empirical comparisons such as the SAT Competitions can fluctuate simply by re-running the experiments, and we provide statistical bounds on such variations, under some assumptions. We also project the fragility of state-of-the-art runtime prediction methods to these runtime distributions, showing that it is insufficient to take a single sample of the runtime in the current practice. Such observations have broad reaching implications for empirical comparisons in a number of fields, solver analysis, portfolios, automated configuration, and runtime prediction.

There may not be a single solution that should be applied to overcome this phenomenon, but rather a context dependant approach should be taken. For empirical evaluations, solvers should be compared based on their runtime distributions. Statistical tests such as the Kolmogorov-–Smirnov or Chi-squared tests seem applicable. Of course, this increases the computational cost of evaluating solvers but gives the advantage of more authoritative and robust conclusions.

One related question concerns the appropriate number of runtime samples required to obtain representative results. For the fields of runtime prediction, solver portfolios, and automated configurators, there remain a great number of open questions. It is simply insufficient to rely on a single statistic of a solver's runtime as ground-truth. Alternatively we should consider predicting statistics or parameters of the runtime distribution. Depending on the application we may attempt to maximise the probability of solving an instance within a certain time, we may prefer solvers that give more consistent behaviour, or conversely a solver which varies dramatically in the hope that we might get lucky. Additionally, in a parallel setting this would have knock-on effects. There are many avenues for future study in these areas.

Chapter 6

Balancing Solution Time and Energy Consumption

Summary. *With the proliferation of cloud computing, it is natural to think about a solver which can scale up by launching searches in parallel on thousands of machines (or cores). However, this could result in consuming a lot of wasted energy. Moreover, not every instance would require thousands of machines. The challenge is to resolve the tradeoff between solution time and energy consumption optimally for a given problem instance. We analyse the impact of the number of machines (or cores) on not only solution time but also on energy consumption. We highlight that although solution time always drops as the number of machines increases, the relationship between the number of machines and energy consumption is more complicated. In many cases, the optimal energy consumption may be achieved by a middle ground, and we analyse this relationship in detail. The tradeoff between solution time and energy consumption is studied further, showing that the energy consumption of a solver can be reduced drastically if we increase the solution time marginally. We also develop a prediction model using machine learning, demonstrating that such insights can be exploited to achieve faster solutions times in a more energy efficient manner.*

6.1 Motivation

Energy consumption for cloud providers and data centres is a growing concern, being one of the largest consumers of electricity [103]. Recently, practitioners in the areas of Constraint Satisfaction (CSP) [139], Boolean satisfiability (SAT) [23], Integer Programming (IP) [162], and numerous other combinatorial search frameworks have turned to the cloud to solve larger and more challenging combinatorial problems efficiently. Many industrial solvers such as IBM ILOG CPLEX and Gurobi already exploit the elasticity of the cloud. These solvers can run on many machines in parallel to solve difficult combinatorial problems. The traditional view of parallel computing has focused on minimising execution time, in which case one might simply launch the solver on all the available machines. An issue arises in that one does not know a-priori the optimal number of machines to be used in parallel, nor has the energy consumption of such a decision been considered. In the context of solving combinatorial problems in the cloud, solution time alone cannot be viewed as a single objective. Instead, one needs to assess the tradeoff in solution time against energy consumption. In our context, the total energy consumption is approximated by the solution time multiplied by the number of searches done in parallel (number of cores).

In general, solving combinatorial search problems is an $\mathcal{NP}$-complete task, typically solved using a combination of search and inference to prune the search space. Choices for parameters such as the search heuristics, restarting policy, and even random seed can affect the size of the search space and subsequently the time it takes to find a solution [66]. Variable and value selection heuristics have elements that are stochastic in nature, so the slightest difference over repeated runs can magnify performance variations [86]. Thus, as seen in Chapter 5, modern combinatorial search solvers often exhibit a very high variation in solution time. Such variations can be modelled by heavy- or fat-tailed distributions [71]. Intuitively, these model a non-negligible probability of a solver taking exponential time. However, the runtime distributions can be exploited, either by randomised restarting [66], or parallelisation [81]. An instance for which the runtime is variable may be solved more-effectively if several searches are performed in parallel using different seeds, with search terminating as soon one finds a solution.

In this respect, we exploit the runtime distribution through parallel searches and study its impact, not only on solution time but also on energy consumption as the number of CPU-cores (or machines) is increased. We show that the relationship between the number of cores and the total energy consumption is not a simple linear relationship. The natural intuition is to assume that as the number of machines is increased, energy increases correspondingly. In fact, in many cases the minimal energy consumption may be achieved by a using a larger number of machines, with the increased likelihood of finding a solution faster meaning the search can be terminated sooner across all machines, resulting in a reduced energy consumption overall. Secondly, we analyse the trade-off between the solution time and the total energy consumption. We motivate the need for a multi-objective optimisation problem to decide the number of (virtual) machines offered by cloud providers in order to strike a balance between solution time and energy consumption. Finally, we demonstrate that it is possible to use machine learning to predict the number of machines (or number cores) that are required to meet a desired level of balance between energy consumption and solution time.

6.2 Solution Time, Number of Cores, Energy

This section analyses the behaviour of the solution time and the total energy with respect to the number of cores. Without loss of generality we assume each physical machine is associated with one CPU-core.

6.2.1 Empirical Setup

We reuse the benchmark set from Chapter 5, comprised of 1676 industrial instances of combinatorial problems from 9 years of the SAT Competitions, Races, and Challenges between 2002 and 2011 [2]. Each instance was run using MiniSat 2.0 [43] as the solver with 100 different seeds, a timeout of 1 hour CPU-time for each run, and a limit of 2GB RAM. Performance data was collected on a cluster of Intel Xeon E5430 Processors (2.66GHz) running CentOS 6. A total of 315 weeks of CPU-time was consumed to accumulate this performance data. Instances that were not solved within the time limit across any run, or were solved in under 1 second across every run were excluded, leaving a total of 902 challenging industrial instances.

The solution time for a instance, when running on k cores in parallel, is equal to the minimum of k parallel runs, since all other runs will be terminated as soon as one finds a solution. If the precise distribution of the runtime is known, then we may use order statistics to model the expected time of running k parallel searches.

However, since many of the instances considered did not fit any well known distribution with high-confidence, we will sample from the empirical distribution to compute this value. By repeatedly taking a minimum of k sample runtimes we get an approximation of the expected runtime for k parallel searches. By doing this over 100,000 iterations we should achieve an accurate approximation of the true expected runtime.

6.2.2 Solution Time versus Number of Cores

Figure 6.1 illustrates how the expected solution time changes as the number of cores (number of parallel searches) increases. Only a sample of the most challenging instance are presented, but they are representative of the complete data set. Naturally, solving the same instance many times in parallel by using more cores reduces the solution time. It is interesting to see that in some cases multiple orders of magnitude speedup can be achieved by only a handful of additional cores. The solution time for any given problem instance is always non-increasing with respect to the number of cores.

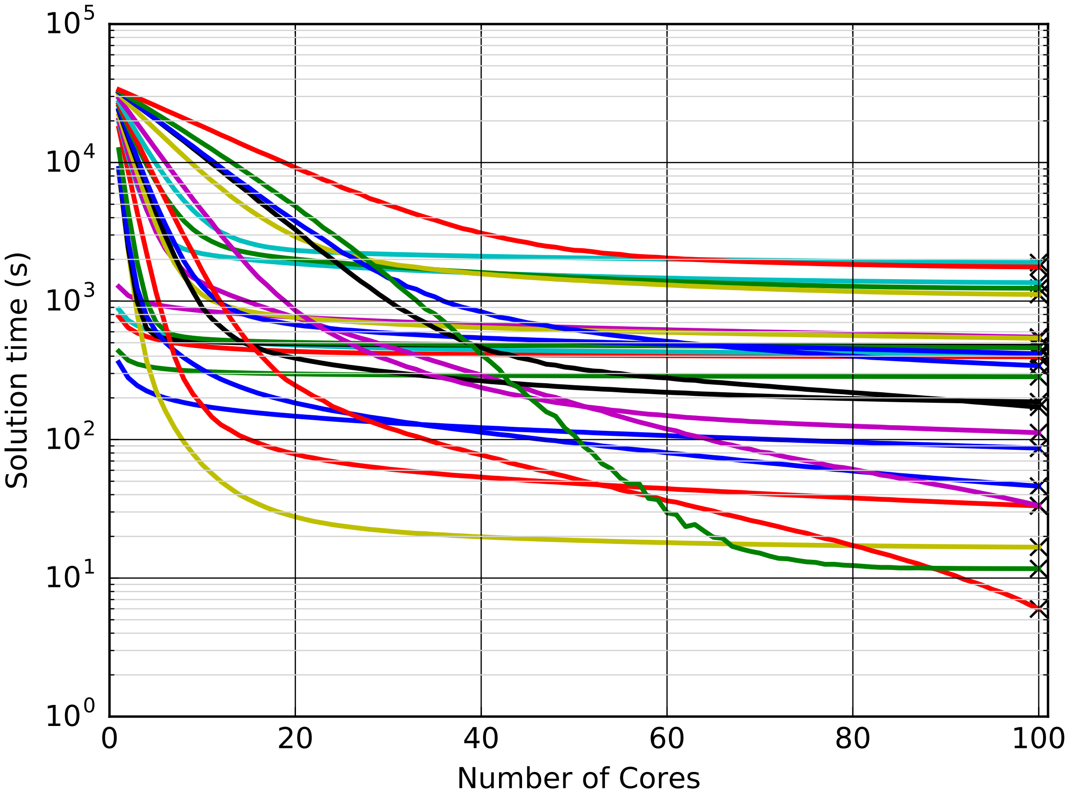


Figure 6.1: Illustration of solution time versus the number of cores for some sample instances. The minimal solution for each instance is marked.

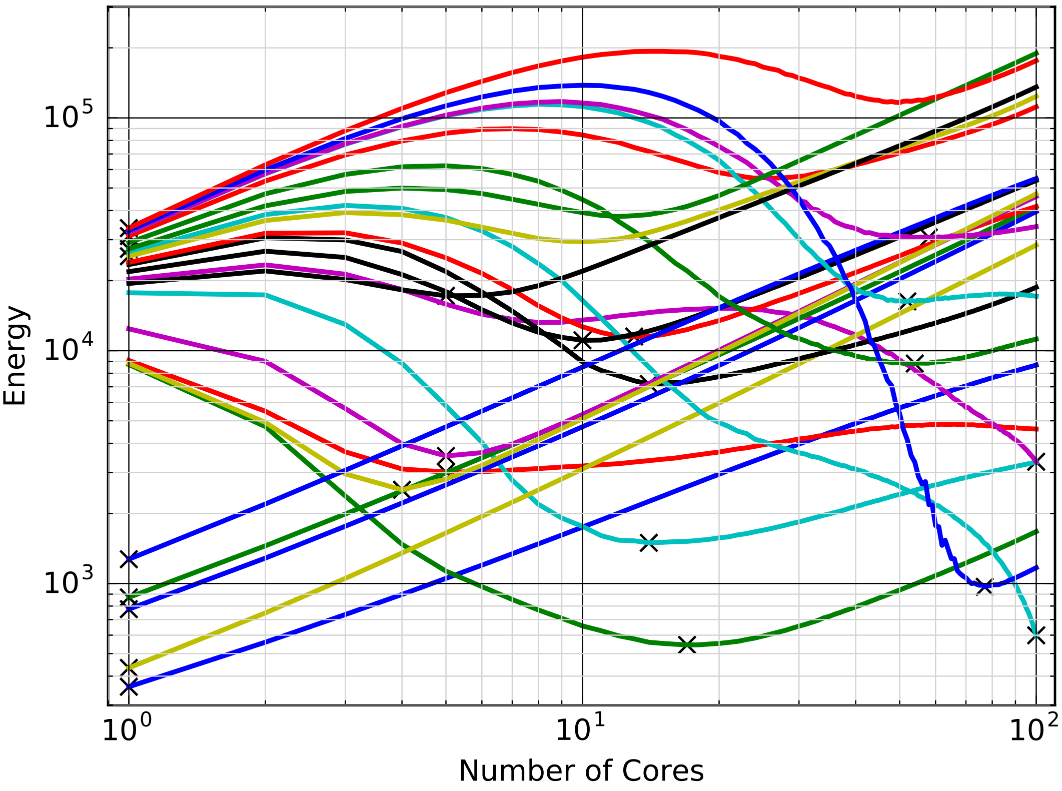


Figure 6.2: Illustration of the total energy versus the number of cores in log-scale for some sample instances. The minimal energy point is marked for each instance.

6.2.3 Energy versus Number of Cores

Figure 6.2 illustrates the energy consumed with respect to the number of cores for the same set of instances as used in Figure 6.1. Let $E[s_k]$ denotes the expected solution time using k cores. The energy consumed using k cores is going to be proportional to the expected solution time with respect to k cores multiplied by k, i.e. $E[s_k] \times k$. Recall that each point in the figure is the mean of 100,000 samples.

Although the expected solution time is non-increasing with respect to number of cores, the product of the number of cores and the expected time results in a number of interesting profiles. Sometimes the energy cost initially decreases as the number of cores increases, reaches a minima, and steadily climbs again. In other cases, the energy cost initially increases with respect to the number of cores and thereafter it declines as the number of cores increases further. Other interesting profiles are also visible in the figure. Evidently, there is no consistent behaviour between instances which achieves the minimal energy cost. The relationship between the total energy consumed with respect to the number of cores is more complicated as evident by a variety of behaviours shown in the figure.

The total energy consumed for solving an instance depends on the runtime distribution. For example, if in certain cases the runtime is uniform, showing no variation in runtime between different runs, then the minimal energy cost is achieved by sticking to a single core, adding any more only serves to increase the energy cost. In contrast, if the distribution is heavy-tailed and if the expected solution time using 100 cores is 100 times less than the running time using 1 core for a given instance then the most energy efficient manner is by running it on 100 cores. Additionally, a middle grounds also exist, where the most energy efficient solution is somewhere between 1 and 100 cores. Thus, the energy consumed is minimal using k cores if $E[s_k] \times k$ is less than $E[s'_k] \times k'$ for any $k' \geq 1$.

6.2.4 Expected Solution Time versus Energy

Figure 6.3 plots the trade-off between the expected solution time and the energy consumption for a sample of instances. It shows that the energy consumption curve with respect to solution time can be significantly different for different problem

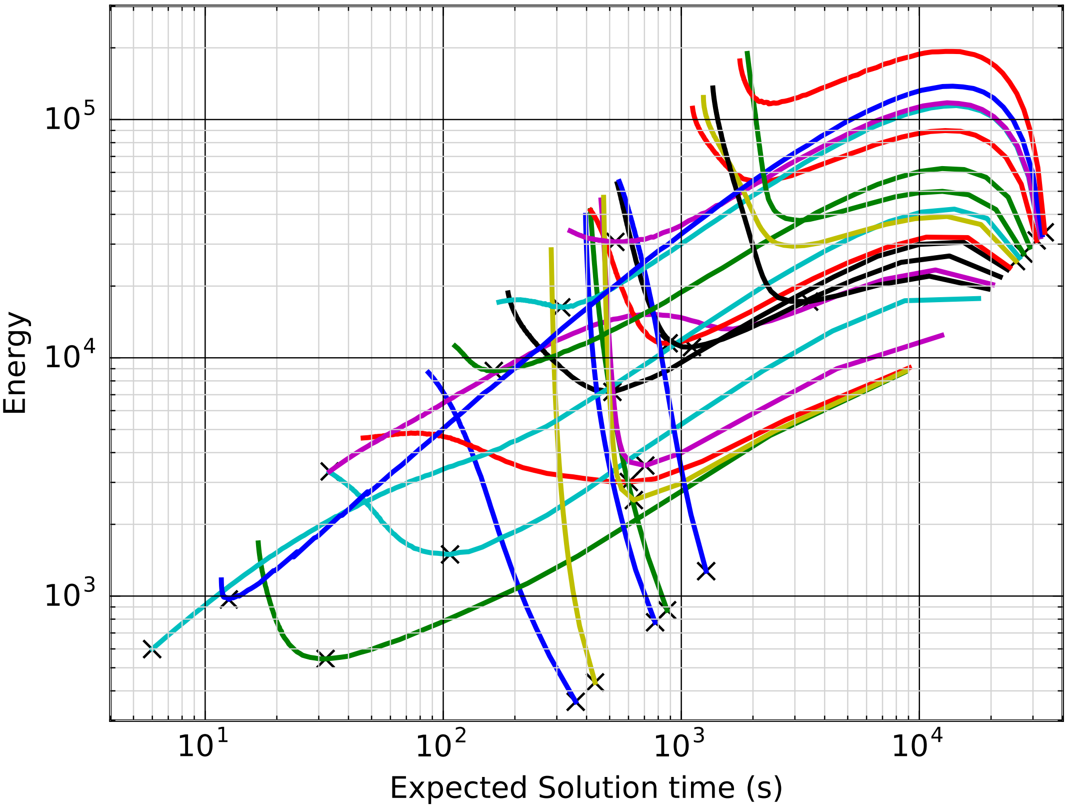


Figure 6.3: Illustration of solution time versus total energy in log-scale for some sample instances. The plus symbol marks the minimal energy for each instance.

instances. The benefit in terms of energy consumption from k independent parallel searches is determined by the nature of the full distribution of runtimes. We remark that if the expected solution time is minimum using k cores then the total energy curve would be linear with respect to the number of cores beyond the point k. In other words, the total energy required by k' where $k' > k$ would always be more than that required by k cores. Thus, if the expected time stops improving beyond a given number of cores k, then any solution obtained by using k' cores where $k' > k$ would not be part of the pareto-frontier. Consequently it will be dominated by at least one solution obtained using k'' cores where $k'' \leq k$.

6.2.5 Solution Time versus Energy Tradeoff

Figure 6.4 presents the trade-off between the expected solution time and energy consumption, aggregated over all instances. The figure depicts that on average by increasing the solution time by just 10%, the energy consumption can be reduced by 20%, and by increasing the solution time by 20%, the total energy can be reduced by 40%. Thus, depending on the preferred bound on the expected solution time, it might be possible to select a number of cores that minimises the energy

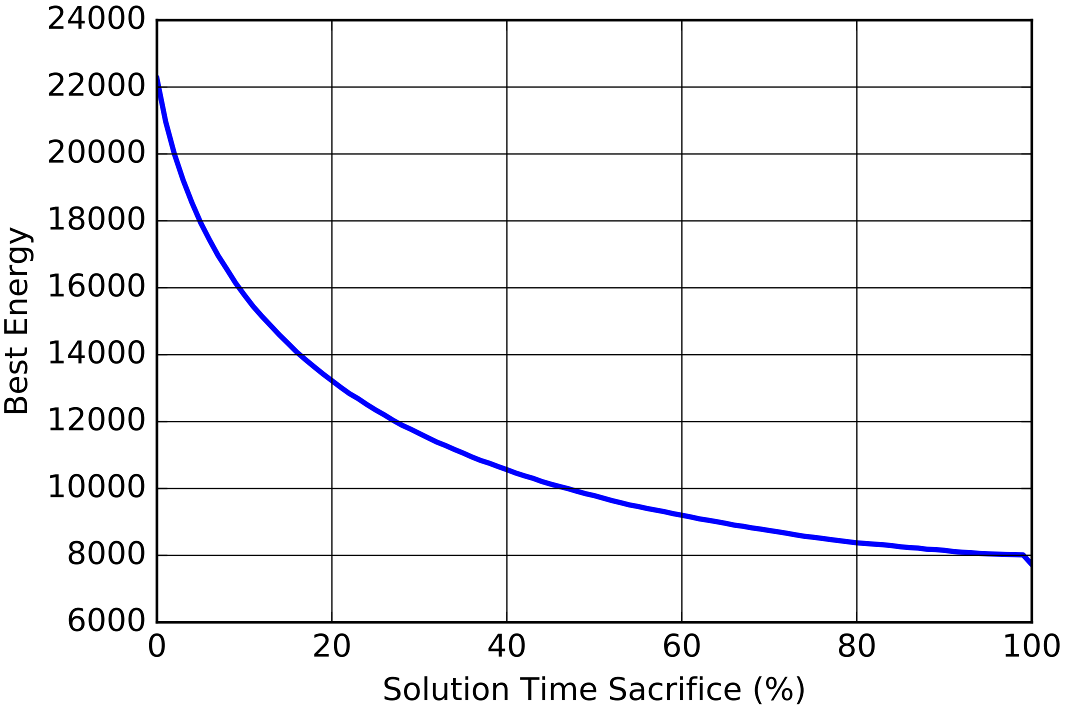


Figure 6.4: Illustration of trade off between solution time and the best energy achievable over all instances.

consumption on a per-instance level. More precisely, the objective would be to predict a number of cores that can minimise the energy consumption and solve a given problem instance within a given target solution time.

6.3 Predicting the Optimal Number of Cores

The previous sections have presented evidence that there is no consistent number of cores to run in order to achieve the desired level of balance between energy consumption and solution time. This section demonstrates that a machine learning algorithm can be built to exploit this knowledge and make intelligent decisions on an instance specific basis. In particular, we will develop a model for predicting the number of cores for minimising energy consumption.

To develop a prediction model, we employ the same state of the art collection of 138 features [166], that were used in Chapters 4 and 5. These have proved highly-effective in the areas of runtime prediction [86, 96] and solver portfolios [99, 165]. Random forest regression is used as the machine learning model, with default parameters except for setting the number of estimator trees to 100. This model has been shown to be highly effective, robust, and is capable of modelling highly non-linear relations. The model is built using stratified 10-fold randomised cross-validation. This splits the dataset into 10 equally sized folds with an even distribution of the label in each. One fold is set aside for testing with the remaining folds used to train the model. This is repeated with each fold taking a turn as the test set.

Each instance is labelled with the number of cores that minimised the overall energy consumption, i.e. the minima values marked in Figure 6.3. The goal of the machine learning algorithm is to predict this value. For the evaluation, we take the predicted number of cores and compute its success rate, total energy, and expected solution time.

Table 6.1 summarises the comparison between the intelligent machine learning model to various baseline policies. Results are sorted by success rate first and then by solution time. The success rate shows the expected percentage of the jobs to produce a valid result within the specified time limit of 1 hour. The solution time shows the expect time in which a solution would be found and returned to the user. As a proxy for the total energy consumed, we use the cumulative CPU-time across all cores. More sophisticated energy functions may also be employed, but

Table 6.1: Evaluation of various parallel policies.

	Policy	Success%	Total Energy	Solution Time (s)
1	Fixed 100 cores	100.0%	23227	232.3
2	VB Solved	100.0%	7727	362.4
3	ML Prediction	98.7%	5494	366.8
4	Fixed 8 cores	96.4%	3228	403.5
5	Fixed 4 cores	94.9%	1883	470.9
6	Fixed 2 cores	93.0%	1107	553.3
7	VB Energy	92.6%	583	556.9
8	Fixed 1 core	90.5%	654	654.2

the one used here serves to be intuitive.

The first set of baseline policies consider a static approach where the instance is always run on a fixed number of k cores. Two other baselines correspond to the *virtual best* (VB) energy policy, and the *virtual best solved* policy. These respectively correspond to an oracle choosing, for each instance, the number of cores leading to i) the overall minimal energy cost, and ii) the highest expected success rate with minimal energy cost.

Firstly, as would be expected, the fixed policy of 1 core is the worst in terms of both success rate and solution time. Interestingly, the virtual best energy policy, as well as having a lower energy consumption has a slightly better success rate and lower solution time than the single core policy. As the number of cores in the fixed policies increases, both the success rate and solution time improve, but the overall total energy increases. Naturally, the largest policy, where all 100 cores are used in parallel provides the highest success rate and best solution time but its energy cost is wasteful.

Most importantly, the machine learning model that predicts the number of cores to be run for each instance can provide a success rate of almost 99% and a solution time very close to the VB Solved policy. Interestingly, its energy consumption is much better than that of the VB Solved; by sacrificing a success rate of 1%, it reduces the total energy usage by 29%.

It was surprising that the random forrest performed so well given that the problem was modelled as a classification task. The target predictions are essentially discrete values representing the number of cores, but they do in fact have an ordinal relationship. The benefit of using a random forrest initially was that little normalisation or scaling of the features is needed. The model presented above was intended as a prototype, we expected to have to refine it in some form, such as bucketing the number of cores, or by switching to a regression model.

In the end, the random forrest ended up making predictions that were relatively close to the optimal value. Where the number of cores it predicted was not optimal, it was often close and the resulting energy cost for that is not far from the optimal. This is surprising, and impressive, as the final evaluation metric differs to that which the random forrest tries to minimise during the training phase. When training the random forrest, it tries to minimise the misclassification penalty, treating the corresponding labels as simply right or wrong. So, in some sense, the decision trees of the random forrest are picking up some latent information about the relationship between the instance features and its runtime distribution, warranting some investigation in future work.

6.4 Chapter Summary

In this chapter we have proposed an elastic solver that can balance the solution time and energy consumption. The solver can scale up in the cloud setting by predicting the number of cores required to strike the balance between the two criterion. We have studied the behaviour of the energy consumed by the solver for many real-world industrial instances when different number of cores are used and provided some insight into their intricate relationship. Despite the non-trival relationship between solution time and energy, the prediction model is highly effective at predicting the optimal number of cores which will minimise the overall energy consumption.

Chapter 7

Conclusions and Future Work

Summary. *To conclude this dissertation, this chapter first summarises the theses defence, and subsequently outlines some possible avenues for future work.*

7.1 Thesis Defence

We conclude this dissertation by recapitulating the two sub-thesis which centred around the exploitation of machine learning for solving combinatorial decision and optimisation problems, and summarising their defence.

Sub-thesis 1. *To-date the application of machine learning to improving the efficiency with which combinatorial problems can be solved has focused on either selecting a solver from a portfolio of possibilities, or on tuning how a specific solver should be used. We claim that a machine learning approach can provide even greater improvement in problem solving efficiency if it can select amongst a set of alternative problem representations in addition to solver choice.*

Defence. In Chapter 3, we studied a variety of adaptation schemes for a family of portfolios using case-based reasoning. We can conclude that adaptation schemes that consider runtime rather than relative ranking give superior performance. We also proposed a constraint programming model for the NP-hard task of computing the Kemeny optimal ranking, and describe an encoding to which linearises to a mixed integer programming model. Given a set of voter rankings, we can effectively compute the Kemeny optimal aggregate ranking.

In Chapter 4 we presented a hierarchical portfolio, named Proteus. The portfolio was first applied to the constraint satisfaction problem, considering a portfolio of CSP solvers, as well as a number of encodings to SAT and subsequently a portfolio of SAT solvers. Detailed empirical evidence across the phase-transition established that it is not sufficient to consider the decisions of which representation or solver to use in isolation, but that they must be considered in tandem.

The hierarchical nature of the portfolio makes it highly extensible with additional encodings and solvers. Moreover, different models and features may be used to make different decisions in the hierarchy, providing greater flexibility to exploit additional knowledge such as features of the chosen representation. We empirically demonstrated the complementary nature of such a portfolio and ultimately its superior performance to that of a portfolio based on a single representation.

The relationship between CSP and SAT, in terms of empirical performance, was studied in detail. Specifically, the faster paradigm is not necessarily distinguishable by the instance category. We also provided some insight to highlight certain characteristics of the instance which hint at the preferred paradigm.

Proteus was also applied to the domain of graphical models. A number of languages were considered, employing several translations between them, and subsequently a set of solvers. We demonstrated the complementary nature between languages in this domain, and that it can be successfully exploited by machine learning models in a portfolio to achieve significant empirical gains.

Sub-thesis 2. *The complex runtime distributions exhibited by combinatorial solvers on a range of interesting problem instances pose a challenge to the standard methodology in algorithm selection and configuration which does not take a holistic view of such distributions. Considering the runtime distribution in a more holistic fashion provides greater insight into solver performance, but also presents a range of challenges that the research community should focus more purposefully upon.*

Defence. Chapter 5 questioned a fundamental assumption being made about the runtime behaviour of complete search solvers. Modern solvers usually have some form of in-built randomness. As a consequence their runtime exhibit significant variation, sometimes by orders of magnitudes on individual instances. We have shown that the outcome of empirical comparisons such as the SAT Competitions can fluctuate simply by re-running the experiments, and we provide statistical bounds on such variations. We also projected the fragility of state-of-the-art runtime prediction methods to these runtime distributions, showing that it is insufficient to take a single sample of the runtime in the current practice. Such observations have broad reaching implications for empirical comparisons in a number of fields, solver analysis, portfolios, automated configuration, and runtime prediction.

There may not be a single solution that should be applied to overcome this phenomenon, but rather a context dependant approach should be taken. For empirical evaluations, solvers should be compared based on their runtime distributions. Statistical tests such as the Kolmogorov-–Smirnov or chi-squared tests would be applicable.

For the fields of runtime prediction, solver portfolios, and automated configurators, there remains a great number of open questions. It is simply insufficient to take a single sample of a solver's runtime as ground-truths. Alternatively we should consider predicting statistics or parameters of the runtime distribution. Depending on the application we may attempt to maximise the probability of solving an instance within a certain time, we may prefer solvers which give more consistent behaviour, or conversely a solver which varies dramatically in the hope that we may get lucky. Additionally, in a parallel setting this would have knock-on effects.

The runtime distribution of a solver can be exploited in a cloud computing setting. We proposed an elastic solver, in Chapter 6, that considers the overall energy consumption when running many searches in parallel. We studied the impact of the number of machines on not only the solution time, but also on energy consumption, providing some insights into their intricate relationship. We demonstrated that the overall energy consumption can be reduced by running search in parallel, exploiting the lower end of the runtime distribution to find a solution quicker and terminate the remaining search. In sacrificing some solution quality, we can also achieve significant reductions in energy consumption. A machine learning model can be built to effectively exploit this knowledge by predicting the optimal number of parallel searches to be run, in order to achieve a low energy consumption.

7.2 Future Work

7.2.1 SAT Encodings

Pre-processing. There is much scope for improvements and optimisation around the process of encoding a CSP instance to SAT. Pre-processing the original CSP or employing some transformations before encoding could be a promising direction which could reduce the size or complexity of the encoded instance. Common subexpression elimination has recently been used to improve the encoded SAT instance, resulting in dramatically reduced search effort by the SAT solver [125]. Additional pre-processing techniques such as performing a single round of CSP propagation, possibly a higher-level of consistency such as singleton arc-consistency, could lead be further gains. Naturally, there is an additional cost associated with performing this reasoning, but this may elicit a reduced time to encode the instance and/or reduce the time required to solve the instance.

Mixed Encodings. Throughout this dissertation, when encoding the instance to SAT, the complete instance has been encoded using the same encoding. A natural extension would be to consider mixed-encodings whereby the encoding is varied depending on characteristics of the problem. The regular encoding which we employ does this to some extent whereby the domains are encoded in two representations and the encoding for particular types of constraints is determined by hand-crafted rules. `CSP2SAT4J` [109] employed a similar methodology, varying between the direct or support encoding according to hand-crafted rules.

The creation of such rules could be automated by learning from past performance data. Alternatively, machine learning models could be employed to learn and predict what encoding to be used on a per-constraint or variable basis. However, it is not obvious how such a model could be trained. It is not as simple as varying the encoding on a single constraint to learn from its performance, rather the performance on the solver is tied to a number of other combinatorial factors.

7.2.2 Multi-language

Chapter 4 demonstrated the significant performance gains achievable by employing alternate representations, showing that SAT solvers can compliment CSP solvers nicely, and that the same benefits are attainable for graphical model languages. It would be natural to extend the Proteus approaches with additional languages such as satisfiability modulo theories, answer set programming, mixed integer programming, and so on. Furthermore, we conjecture that such utility can be obtained in other related languages by employing translations such as done in Proteus.

7.2.3 Runtime Distributions

Much progress has been made on understanding problem hardness, the behaviour of backtracking search, and their runtime distributions. Many seminal papers modelling solver performance in the context of heavy- and fat-tailed distributions were written over a decade and a half ago, during a time before randomised restarting was common and the fastest complete-search algorithms for SAT were based on the DPLL method. Modern SAT solvers have come a long way since, albeit with much of the evolution stemming from such runtime distribution studies. Nevertheless, it is evident that modern solvers have an element of stochasticity built in, resulting in runtime distributions with significant variance. Undoubtedly, it is time such studies are revisited given the considerable advances in solver technology since. We may ask questions such as whether the heavy-tailed phenomenon has been reduced or even completely eliminated by recent advances. Does modern solver performance fit into different classes of distributions, are the variations and probability of exponential worst-case time as extreme. Such a study would be complicated by a multitude of techniques which form part of modern solvers, such as randomised restarting, learning between restarts, explanations, etc. We are almost at a level where solvers need to be treated as *black-boxes*.

7.2.4 Portfolios

Chapter 5 raises some questions regarding the confidence in empirical comparisons of solvers that exhibit a highly variable distribution of their runtimes, including those presented in Chapter 4. It is insufficient to take a single sample as ground-truth, as is the current practice. The findings in Chapter 5 do not invalidate such results completely but rather argue for a more statistically founded method in which we can have more confidence. We could gain more empirical confidence in Chapter 4's findings by rerunning the experiments, but given the large computation expense incurred already, 100 weeks CPU-time, it was not reasonable to re-run the entire dataset again. Besides, we do not propose any absolute solution in Chapter 5, but there are clearly a great number of open questions that should be answered first, and many possible avenues to consider for fields such as solver portfolios, automated configuration, and runtime prediction. Firstly, much of this work could be guided by a better understanding of modern solver runtime distributions which have evolved dramatically since the majority of these studies.

Instead of trying to predict the exact runtime, maybe we should predict parameters of the runtime distribution. This would give a much more holistic approach, enable greater flexibility, and be much more reliable all round. In the context of local search SAT solvers, where it is natural to consider dramatic variation between runs, predicting the runtime distribution is an active area of research [13].

Existing portfolio approaches are generally designed for maximising an objective such as the number of instances solved. If we are able to consider the runtime distribution of a solver, it opens up new possibilities in terms of objectives and enables more flexible, practical applications. For example, we may want to maximise the probability of solving an instance within a certain time limit, or ensure that the median solution time is below a certain threshold. For portfolios which run solvers in parallel [82, 113, 140], it should be necessary to take the runtime distribution into account.

Many opportunities emerge with the proliferation of cloud computing, such as guaranteeing a certain response time for computing solutions, or considering the energy cost [92]. There are many avenues of potential study here when considering solvers with highly variable runtime distributions. In particular, the effectiveness of an *elastic solver* such as that presented in Chapter 6 could be improved by a more in-depth understanding of the runtime distribution across different instance categories, as well as incorporating a portfolio of solvers that exhibit contrasting runtime distributions.

Bibliography

[1] CSP Solver Competition Benchmarks. `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`, 2009.

[2] SAT Solver Compeition, November 2014. URL `http://www.satcompetition.org`.

[3] Y. Tourbier A. Oplobedu, J. Marcovitch. CHARME: Un langage industriel de programmation par contraintes, illustré par une application chez Renault. In *Proceedings of the Ninth International Workshop on Expert Systems and their Applications*, pages 55–70, 1989.

[4] Ignasi Abío and Peter J. Stuckey. Encoding linear constraints into SAT. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming, CP 2014*, volume 8656 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2014. doi: 10.1007/978-3-319-10428-7_9.

[5] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A parametric approach for smaller and better encodings of cardinality constraints. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, CP 2013*, volume 8124 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2013. doi: 10.1007/978-3-642-40627-0_9.

[6] Dimitris Achlioptas, Michael S. O. Molloy, Lefteris M. Kirousis, Yannis C. Stamatiou, Evangelos Kranakis, and Danny Krizanc. Random Constraint Satisfaction: A More Accurate Picture. *Constraints*, 6(4):329–344, 2001. doi: 10.1023/A:1011402324562.

[7] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In *Proceedings of*

the 1$^{\text{ères}}$ Journées Francophones de Programmation Logique, JFPL 1992, page 51, 1992.

[8] Enrique Matos Alfonso and Norbert Manthey. Riss 4.27 BlackBox. SAT Competition, 2014.

[9] David Allouche, Simon de Givry, George Katsirelos, Thomas Schiex, and Matthias Zytnicki. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming, CP 2015*, pages 12–28, 2015.

[10] Carlos Ansótegui and Felip Manyà. Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, pages 1–15, 2004.

[11] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A Gender-based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, CP 2009*, pages 142–157, 2009.

[12] Carlos Ansótegui, Yuri Malitsky, and Meinolf Sellmann. MaxSAT by Improved Instance-Specific Algorithm Configuration. In *Proceedings of the 28th National Conference on Artificial Intelligence, AAAI 2014*, pages 2594–2600. AAAI Press, 2014.

[13] Alejandro Arbelaez, Charlotte Truchet, and Philippe Codognet. Using sequential runtime distributions for the parallel speedup prediction of SAT local search. *Theory and Practice of Logic Programming*, 13(4-5):625–639, 2013. doi: 10.1017/S1471068413000392.

[14] Josep Argelich, Alba Cabiscol, Inês Lynce, and Felip Manyà. Encoding Max-CSP into partial Max-SAT. In *Proceedings of the 28th International Symposium on Multiple-Valued Logic, ISMVL 2008*, pages 106–111, 2008.

[15] Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 Competition. *Proceedings of SAT Competition 2013 – Solver and Benchmark Descriptions*, page 42, 2013.

[16] László Babai. Monte-carlo Algorithms in Graph Isomorphism Testing. Technical Report DMS 79-10, Université de Montréal, 1979.

[17] Fahiem Bacchus. GAC via Unit Propagation. In *Proceedings of the 13th*

International Conference on Principles and Practice of Constraint Programming, CP 2007, pages 133–147. Springer, 2007.

[18] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical Report SICS-T–2005/08-SE, 2005.

[19] Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, pages 29–83. Elsevier, 2006. doi: 10.1016/S1574-6526(06) 80007-6.

[20] Nadja Betzler, Michael R Fellows, Jiong Guo, Rolf Niedermeier, and Frances A Rosamond. Computing Kemeny rankings, parameterized by the average KT-distance. In *Proceedings of the 2nd International Workshop on Computational Social Choice, COMSOC 2008*, pages 85–96, 2008.

[21] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. *Proceedings of SAT Competition 2013 – Solver and Benchmark Descriptions*, 2013.

[22] Armin Biere. Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014. SAT Competition, 2014.

[23] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, February 2009. ISBN 978-1-58603-929-5.

[24] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Thomas Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger H. Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A Benchmark Library for Algorithm Selection. *CoRR*, abs/1506.02465, 2015.

[25] Duncan Black. The Theory of Committees and Elections. 1958.

[26] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI 2004*, pages 146–150, 2004.

[27] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees.* CRC press, 1984.

[28] Klaus Brinker and Eyke Hüllermeier. Case-based Multilabel Ranking. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007.*

[29] Klaus Brinker and Eyke Hüllermeier. Case-based Label Ranking. In *Proceedings of 17th European Conference on Machine Learning, ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 566–573. Springer, 2006. doi: 10.1007/11871842_53.

[30] Klaus Brinker, Johannes Fürnkranz, and Eyke Hüllermeier. A Unified Model for Multilabel Classification and Ranking. In *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, pages 489–493. IOS Press, 2006. ISBN 1-58603-642-4.

[31] Jingchao Chen. A New SAT Encoding of the At-Most-One Constraint. In *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation, ModRef 2010*, 2010.

[32] Geoffrey Chu and Peter J. Stuckey. Learning value heuristics for constraint programming. In Laurent Michel, editor, *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2015*, pages 108–123. Springer, 2015. ISBN 978-3-319-18008-3. doi: 10.1007/978-3-319-18008-3_8.

[33] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971. doi: 10.1145/800157.805047.

[34] Martin C. Cooper, Simon de Givry, M Sanchez, Thomas Schiex, Matthias Zytnicki, and T Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010. doi: 10.1016/j.artint.2010.02.001.

[35] John P. Costas. A study of a class of detection waveforms having nearly ideal range – doppler ambiguity properties. *Proceedings of the IEEE*, 72(8): 996–1009, Aug 1984. ISSN 0018-9219. doi: 10.1109/PROC.1984.12967.

[36] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming, CP 2011*, pages 225–239. Springer, 2011.

[37] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MaxSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013*, pages 166–181, 2013.

[38] Martin Davis, George Logemann, and Donald W. Loveland. A Machine

Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962. doi: 10.1145/368273.368557.

[39] Simon de Givry, Matthias Zytnicki, Federico Heras, and Javier Larrosa. Existential Arc Consistency: Getting Closer to Full Arc Consistency in Weighted CSPs. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005*, pages 84–89. Morgan Kaufmann, 2005.

[40] Simon de Givry, Steve Prestwich, and Barry O'Sullivan. Dead-end elimination for weighted CSP. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, CP 2013*, pages 263–272. Springer, 2013.

[41] Gregoire Dooms. *The CP(Graph) Computation Domain in Constraint Programming.* PhD thesis, Université catholique de Louvain, Faculté des sciences appliquées, 2006.

[42] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank Aggregation Methods for the Web. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 613–622, New York, NY, USA, 2001. ACM. ISBN 1-58113-348-0. doi: 10.1145/371920.372165.

[43] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, pages 502–518. Springer-Verlag, 2004.

[44] Jean-Guillaume Fages. *Exploitation de structures de graphe en programmation par contraintes. (On the use of graphs within constraint-programming).* PhD thesis, École des mines de Nantes, France, 2014. URL https://tel.archives-ouvertes.fr/tel-01085253.

[45] Aurélie Favier, Simon de Givry, Andrés Legarra, and Thomas Schiex. Pairwise decomposition for combinatorial optim. in graphical models. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, pages 2126–2132, 2011.

[46] Chris Fawcett, Mauro Vallati, Frank Hutter, Jörg Hoffmann, Holger H Hoos, and Kevin Leyton-Brown. Improved Features for Runtime Prediction of Domain-Independent Planners. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling, ICAPS 2014*, 2014.

[47] Tadhg Fitzgerald, Yuri Malitsky, Barry O'Sullivan, and Kevin Tierney.

ReACT: Real-Time Algorithm Configuration through Tournaments. In *Proceedings of the 7th Annual Symposium on Combinatorial Search, SOCS 2014*. AAAI Press, 2014.

[48] Eugene C. Freuder. In Pursuit of the Holy Grail. *Constraints*, 2(1):57–61, 1997. doi: 10.1023/A:1009749006768.

[49] Eugene C. Freuder and Barry O'Sullivan. Grand challenges for constraint programming. *Constraints*, 19(2):150–162, 2014. doi: 10.1007/s10601-013-9155-1.

[50] Alan M. Frisch and Paul A. Giannaros. SAT Encodings of the At-Most-k Constraint. In *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation, ModRef 2010*, 2010.

[51] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[52] Johannes Fürnkranz and Eyke Hüllermeier. Pairwise Preference Learning and Ranking. In *Proceedings of 14th European Conference on Machine Learning, ECML 2003*, volume 2837 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2003. doi: 10.1007/978-3-540-39857-8_15.

[53] Marco Gavanelli. The Log-Support Encoding of CSP into SAT. In Christian Bessière, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP 2007*, pages 815–822. Springer, 2007. ISBN 978-3-540-74970-7. doi: 10.1007/978-3-540-74970-7_59.

[54] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning 2007*, pages 260–265. Springer, 2007.

[55] Gecode Team. Gecode: Generic Constraint Development Environment, 2006. URL `http://www.gecode.org`.

[56] Pieter Andreas Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI 1992*, pages 31–35. John Wiley & Sons, Inc., 1992. ISBN 0-471-93608-1.

[57] Allen Van Gelder. Another look at graph coloring via propositional satisfiabil-

ity. *Discrete Applied Mathematics*, 156(2):230–243, 2008. doi: 10.1016/j.dam.2006.07.016. URL `http://dx.doi.org/10.1016/j.dam.2006.07.016`.

[58] Ian P. Gent. Arc Consistency in SAT. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI 2002*, pages 121–125. IOS Press, 2002.

[59] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random Constraint Satisfaction: Flaws and Structure. *Constraints*, 6(4):345–372, 2001. doi: 10.1023/A:1011454308633.

[60] Ian P. Gent, Patrick Prosser, and Barbara M. Smith. A 0/1 encoding of the GACLex constraint for pairs of vectors. In *Workshop on Modelling and Solving Problems with Constraints*, 2002.

[61] Ian P. Gent, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Learning when to use lazy learning in constraint solving. In *Proceedings of the 19th European Conference on Artificial Intelligence, ECAI 2010*, pages 873–878, 2010. doi: 10.3233/978-1-60750-606-5-873.

[62] 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.

[63] Carmen Gervet. *Set intervals in constraint-logic programming: definition and implementation of a language.* PhD thesis, Université de France-Compté, 1995.

[64] Amir Globerson and Tommi S. Jaakkola. Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *Proceedings of the 21st Annual Conference on Neural Information Processing Systems, NIPS 2007*, pages 553–560. Curran Associates, Inc., 2007.

[65] Carla P. Gomes. Randomized Backtrack Search. In Michela Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*, chapter 8, pages 233–291. Kluwer Academic Publishers, 2003.

[66] Carla P. Gomes and Ashish Sabharwal. Exploiting Runtime Variation in Complete Solvers. In *Handbook of Satisfiability*, pages 271–288. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-271.

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

[68] Carla P. Gomes and Toby Walsh. Randomness and structure. In *Handbook of Constraint Programming*, pages 639–664. Elsevier, 2006.

[69] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-Tailed Distributions in Combinatorial Search. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming, CP 1997*, pages 121–135. Springer, 1997.

[70] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence, AAAI 1998*, pages 431–437. AAAI Press, 1998.

[71] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.

[72] Carla P. Gomes, Cèsar Fernández, Bart Selman, and Christian Bessière. Statistical Regimes Across Constrainedness Regions. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP 2004*, pages 32–46, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[73] Shai Haim and Toby Walsh. Restart strategy selection using machine learning techniques. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT 2009*, pages 312–325, Berlin, Heidelberg, 2009. Springer-Verlag.

[74] 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.

[75] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[76] Emmanuel Hebrard. Mistral, a Constraint Satisfaction Library. In *Proceedings of the Third International CSP Solver Competition*, 2008.

[77] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Super solutions in

constraint programming. In *Proceedings of the 1st International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004*, pages 157–172. Springer, 2004. doi: 10.1007/978-3-540-24664-0_11.

[78] Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint Programming and Combinatorial Optimisation in Numberjack. In *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2010*, volume 6140 of *Lecture Notes in Computer Science*, pages 181–185. Springer, 2010. doi: 10.1007/978-3-642-13520-0_22.

[79] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence, AAAI 1988*, pages 660–664. AAAI Press, 1988.

[80] Brahim Hnich. CSPLib problem 034: Warehouse location problem. http://www.csplib.org/Problems/prob034.

[81] Tad Hogg and Colin P Williams. Expected Gains from Parallelizing Constraint Solving for Hard Problems. *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI 1994*, pages 331–336, 1994.

[82] Holger Hoos, Roland Kaminski, Marius Lindauer, and Torsten Schaub. aspeed: Solver Scheduling via Answer Set Programming. *Theory and Practice of Logic Programming*, 15:117–142, 1 2015. ISSN 1475-3081. doi: 10.1017/S1471068414000015.

[83] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[84] Tudor Hulubei and Barry O'Sullivan. The Impact of Search Heuristics on Heavy-Tailed Behaviour. *Constraints*, 11(2-3):159–178, 2006. doi: 10.1007/s10601-006-8061-1.

[85] Barry Hurley and Barry O'Sullivan. Adaptation in a CBR-Based Solver Portfolio for the Satisfiability Problem. In *Proceedings of the 20th International Conference on Case-Based Reasoning Research and Development, ICCBR 2012*, volume 7466 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2012. doi: 10.1007/978-3-642-32986-9_13.

[86] Barry Hurley and Barry O'Sullivan. Statistical regimes and runtime predic-

tion. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 318–324, 2015.

[87] Barry Hurley and Barry O'Sullivan. Introduction to Combinatorial Optimisation in Numberjack. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 3–24. 2016. doi: 10.1007/978-3-319-50137-6_1.

[88] Barry Hurley, Serdar Kadioglu, Yuri Malitsky, and Barry O'Sullivan. Transformation based Feature Computation for Algorithm Portfolios. Technical Report abs/1401.2474, January 2014. URL `http://arxiv.org/abs/1401.2474`.

[89] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2014*, volume 8451 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2014. ISBN 978-3-319-07045-2.

[90] Barry Hurley, Lars Kotthoff, Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Advanced Portfolio Techniques. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 191–225. 2016. doi: 10.1007/978-3-319-50137-6_8.

[91] Barry Hurley, Lars Kotthoff, Barry O'Sullivan, and Helmut Simonis. ICON Loop Health Show Case. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 325–333. 2016. doi: 10.1007/978-3-319-50137-6_14.

[92] Barry Hurley, Deepak Mehta, and Barry O'Sullivan. Elastic Solver: Balancing Solution Time and Energy Consumption. Technical Report arXiv:1605.06940, May 2016. URL `http://arxiv.org/abs/1605.06940`.

[93] Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki, and Simon de Givry. Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints*, 21 (3):413–434, 2016. ISSN 1572-9354. doi: 10.1007/s10601-016-9245-y.

[94] Barry Hurley, Barry O'Sullivan, and Helmut Simonis. ICON Loop Energy Show Case. In *Data Mining and Constraint Programming - Foundations of*

a Cross-Disciplinary Approach, pages 334–347. 2016. doi: 10.1007/978-3-319-50137-6_15.

[95] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization, LION 2011*, pages 507–523, 2011. doi: 10.1007/978-3-642-25566-3_40.

[96] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014. doi: 10.1016/j.artint.2013.10.003.

[97] Kazuo Iwama and Shuichi Miyazaki. Sat-varible complexity of hard combinatorial problems. In *Proceedings of the IFIP 13th World Computer Congress*, pages 253–258, 1994.

[98] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - Instance-Specific Algorithm Configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence, ECAI 2010*, pages 751–756, 2010.

[99] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC – Instance-Specific Algorithm Configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence, ECAI 2010*, pages 751–756, 2010.

[100] Michal Karpinski and Marek Piotrów. Smaller selection networks for cardinality constraints encoding. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming, CP 2015*, volume 9255 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2015. doi: 10.1007/978-3-319-23219-5_16.

[101] Simon Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, October 1990. ISSN 0004-3702. doi: 10.1016/0004-3702(90)90009-O.

[102] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[103] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3):034008, 2008. URL `http://stacks.iop.org/1748-9326/3/i=3/a=034008`.

[104] Arie Koster. *Frequency Assignment: Models and Algorithms*. PhD thesis, 1999.

[105] Lars Kotthoff. LLAMA: leveraging learning to automatically manage algorithms. Technical Report arXiv:1306.1031, arXiv, June 2013. URL `http://arxiv.org/abs/1306.1031`.

[106] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35(3):48–60, 2014.

[107] Lars Kotthoff, Ian P. Gent, and Ian Miguel. A Preliminary Evaluation of Machine Learning in Algorithm Selection for Search Problems. In *Proceedings of the 4th Annual Symposium on Combinatorial Search, SOCS 2011*, 2011.

[108] Jean-Louis Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978. doi: 10.1016/0004-3702(78)90029-2.

[109] Daniel Le Berre and Inês Lynce. CSP2SAT4J: A Simple CSP to SAT Translator. In *Proceedings of the 2nd International CSP Solver Competition*, 2008.

[110] Daniel Le Berre and Laurent Simon. The Essentials of the SAT 2003 Competition. In Enrico Giunchiglia and Armando Tacchella, editors, *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, volume 2919 of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2004. ISBN 978-3-540-20851-8. doi: 10.1007/978-3-540-24605-3_34.

[111] Christophe Lecoutre and Sebastien Tabary. Abscon 112, Toward more Robustness. In *Proceedings of the Third International CSP Solver Competition*, 2008.

[112] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the Empirical Hardness of Optimization Problems: The Case of Combinatorial Auctions. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, CP 2002*, pages 556–572. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2002.

[113] Marius Lindauer, Holger Hoos, and Frank Hutter. From sequential algorithm selection to parallel portfolio selection. In *Proceedings of the 9th International Conference on Learning and Intelligent Optimization, LION 2015*, pages 1–16. Springer, 2015. ISBN 978-3-319-19084-6. doi: 10.1007/978-3-319-19084-6_1.

[114] Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. AutoFolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015. doi: 10.1613/jair.4726.

[115] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47(4):173–180, 1993. doi: 10.1016/0020-0190(93)90029-9.

[116] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977. doi: 10.1016/0004-3702(77)90007-8.

[117] Norbert Manthey. The SAT Solver RISS3G at SC 2013. *Proceedings of SAT Competition 2013 – Solver and Benchmark Descriptions*, page 72, 2013.

[118] Deepak Mehta, Barry O'Sullivan, Luis Quesada, Marco Ruffini, David B. Payne, and Linda Doyle. Designing resilient long-reach passive optical networks. In *Proceedings of the 23rd Conference on Innovative Applications of Artificial Intelligence, IAAI 2011.* AAAI Press, 2011.

[119] Deepak Mehta, Barry O'Sullivan, Lars Kotthoff, and Yuri Malitsky. Lazy Branching for Constraint Satisfaction. In *Proceedings of the 25th International Conference on Tools with Artificial Intelligence, ICTAI 2013*, pages 1012–1019, 2013. doi: 10.1109/ICTAI.2013.152.

[120] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. Soft Constraints Processing . In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 9. Elsevier, 2006.

[121] Tom M. Mitchell. *Machine Learning.* McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

[122] Katta G. Murty. *Linear Programming.* Wiley, 1983. ISBN 978-0-471-09725-9.

[123] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP 2007*, pages 529–543. Springer, 2007.

[124] Bertrand Neveu, Gilles Trombettoni, and Fred Glover. ID Walk: A Candidate List Strategy with a Simple Diversification Device. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP 2004*, pages 423–437. Springer, 2004.

[125] Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically Improving SAT Encoding of Constraint Problems Through Common Subexpression Elimination in Savile Row. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming, CP 2015*, pages 330–340, 2015. doi: 10.1007/978-3-319-23219-5_23.

[126] Mladen Nikolić. Statistical methodology for comparison of SAT solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing, SAT 2010*, pages 209–222. Springer-Verlag, 2010.

[127] Chanseok Oh. MiniSat_HACK_999ED, MiniSat_HACK_1430ED and SWDiA5BY. SAT Competition, 2014.

[128] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. *Proceeding of the 19th Irish Conference on Artificial Intelligence and Cognitive Science, AICS 2008*, 2008.

[129] Lars Otten, Er Ihler, Kalev Kask, and Rina Dechter. Winning the PASCAL 2011 MAP challenge with enhanced AND/OR branch-and-bound. In *Proceedings of NIPS Workshop on Discrete and Combinatorial Problems in Machine Learning, DISCML 2014*, 2012.

[130] Eric Pacuit. Voting Methods. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.

[131] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[132] Justyna Petke. *On the bridge between Constraint Satisfaction and Boolean Satisfiability*. PhD thesis, St John's College, University of Oxford, 2012.

[133] Justyna Petke and Peter Jeavons. The Order Encoding: From Tractable CSP to Tractable SAT. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing, SAT 2011*, 2011.

[134] Steven David Prestwich. CNF Encodings. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages

75–97. IOS Press, 2009. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-75.

[135] Philippe Refalo. Impact-based search strategies for constraint programming. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP 2004*, pages 557–571, 2004.

[136] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI 1994*, pages 362–367. AAAI Press, 1994.

[137] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence, AAAI 1996*, pages 209–215. AAAI Press, 1996.

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

[139] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming.* Foundations of Artificial Intelligence. Elsevier, New York, NY, USA, 2006. ISBN 0444527265.

[140] Olivier Roussel. Description of ppfolio. Technical report, 2011.

[141] Olivier Roussel and Christophe Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[142] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence, ECAI 1994*, pages 125–129. Springer, 1994.

[143] M.I. Schlesinger. Syntactic analysis of two-dimensional visual signals in noisy conditions. *Kibernetika*, 4:113–130, 1976.

[144] Jendrik Seipp, Silvan Sievers, Malte Helmert, and Frank Hutter. Automatic Configuration of Sequential Planning Portfolios. In *Proceedings of the 29th National Conference on Artificial Intelligence, AAAI 2015.* AAAI Press, January 2015.

[145] Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling Scientific Experiments on the Rosetta/Philae Mission. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP 2012*, pages 23–37, 2012.

[146] Helmut Simonis. Constraint applications in networks. In *Handbook of Constraint Programming*, pages 875–903. Elsevier, 2006. doi: 10.1016/S1574-6526(06)80029-5.

[147] Helmut Simonis. Models for Global Constraint Applications. *Constraints*, 12(1):63–92, 2007. doi: 10.1007/s10601-006-9011-7.

[148] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, CP 2005*, pages 827–831. Springer, 2005.

[149] David Sontag, Talya Meltzer, Amir Globerson, Tommi S. Jaakkola, and Yair Weiss. Tightening LP relaxations for MAP using message-passing. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, UAI 2008*, pages 503–510, 2008.

[150] David Sontag, Do Kook Choe, and Yitao Li. Efficiently searching for frustrated cycles in MAP inference. In *Proceedings of the 28th Conference in Uncertainty in Artificial Intelligence, UAI 2012*, pages 795–804, 2012.

[151] Mate Soos. Cryptominisat 2.9.0, 2011.

[152] Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara. System Description of a SAT-based CSP Solver Sugar. In *Proceedings of the 3rd International CSP Solver Competition*, pages 71–75, 2009.

[153] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. Azucar: A SAT-Based CSP Solver Using Compact Order Encoding — (Tool Presentation). In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 456–462. Springer, 2012.

[154] Choco Team. Choco: an Open Source Java Constraint Programming Library, 2008.

[155] Peter van Beek. CSPLib problem 006: Golomb rulers. http://www.csplib.org/Problems/prob006.

[156] Alan Van Gelder. Careful Ranking of Multiple Solvers with Timeouts and Ties. *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing, SAT 2011*, 2011.

[157] Willem-Jan van Hoeve and Irit Katriel. Global Constraints. In *Handbook of*

Constraint Programming, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi: 10.1016/S1574-6526(06)80010-6.

[158] Shankar Vembu and Thomas Gärtner. *Label Ranking Algorithms: A Survey*, pages 45–64. Springer, 2011. ISBN 978-3-642-14125-6. doi: 10.1007/978-3-642-14125-6_3.

[159] Mark Wallace. Practical Applications of Constraint Programming. *Constraints*, 1(1/2):139–168, 1996. doi: 10.1007/BF00143881.

[160] Toby Walsh. Search in a Small World. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*, pages 1172–1177, 1999.

[161] Toby Walsh. SAT v CSP. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000. doi: 10.1007/3-540-45349-0_32.

[162] Laurence A. Wolsey. *Integer programming*. Wiley-Interscience, 1998. ISBN 978-0-471-28366-9.

[163] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hierarchical hardness models for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP 2007*, pages 696–711, 2007.

[164] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: The Design and Analysis of an Algorithm Portfolio for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP 2007*, pages 712–727, 2007. doi: 10.1007/978-3-540-74970-7_50.

[165] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. In *Journal Of Artificial Intelligence Research*, pages 565–606, 2008.

[166] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Features for SAT. Technical report, 2012. URL `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/Report_SAT_features.pdf`.

[167] Yue Kwen Justin Yip. *The Length-Lex Representation for Constraint Programming over Sets*. PhD thesis, Brown University, 2011.