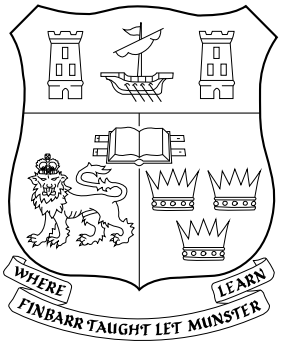
Low Knowledge Algorithm Control for Constraint-Based Scheduling

Tom Carchrae



A Thesis Submitted to the National University of Ireland

in Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in the Faculty of Science.

September, 2009

Research Supervisor: Prof J. Christopher Beck
Research Supervisor: Prof Eugene C. Freuder
Head of Department: Prof James Bowen

Department of Computer Science,
National University of Ireland, Cork.

Contents

Abstract **xi**

1 Introduction **1**
1.1 Motivation 2
1.2 Overview of Dissertation 4
1.3 Summary of Contributions 6

2 Constraint-Based Scheduling **8**
2.1 Constraint Programming 8
2.1.1 Applications 10
2.1.2 Constraint Satisfaction Problems 11
2.1.3 Constraint Optimization 14
2.1.4 Search 15
2.2 Job Shop Scheduling 16
2.2.1 Constraint Model 17
2.3 Constructive Search 18
2.3.1 Propagation 19
2.4 Local Search 23
2.5 Large Neighbourhood Search 24
2.5.1 Search Methods 26
2.5.2 Neighbourhood Heuristics for LNS 26
2.6 Summary 27

3 Algorithm Control Review **28**
3.1 Introduction 28

3.2 The Algorithm Selection Problem 28
3.3 Framework 29
3.3.1 Gathering Knowledge 32
3.3.2 Control Decisions 34
3.4 Off-line Knowledge Control Systems 35
3.4.1 Best Overall Algorithm 37
3.4.2 Algorithm Portfolios 38
3.4.3 Configuring Algorithms 41
3.4.4 Conclusion 42
3.5 Off-line and On-line Knowledge Control Systems 43
3.5.1 Empirical Performance Models 44
3.5.2 Classifying Problem Instances 46
3.5.3 Monitoring Search 48
3.5.4 Control Policies 50
3.5.5 Conclusion 51
3.6 On-line Knowledge Control Systems 52
3.6.1 On-line Reinforcement Learning 53
3.7 Summary 55

4 The Low Knowledge Approach to Algorithm Control 56
4.1 Introduction 56
4.2 The Knowledge Requirement 57
4.2.1 High Knowledge 58
4.2.2 Low Knowledge 59
4.2.3 Comparison 59
4.2.4 Related Work 61
4.3 Knowledge in Algorithm Control 61
4.3.1 Knowledge Engineering Effort 61
4.3.2 High Knowledge Approaches 62
4.3.3 Low Knowledge Approaches 63
4.3.4 Summary 64
4.4 Conclusion 65

5 Low Knowledge Control Applied to Scheduling Algorithms 66
5.1 Introduction . . . 66
5.1.1 Scenario . . . 68
5.1.2 Outline . . . 68
5.2 Pure Algorithms . . . 69
5.2.1 SetTimes . . . 69
5.2.2 Texture . . . 71
5.2.3 Tabu-TSAB . . . 72
5.3 Experimental Details . . . 76
5.3.1 Problem Instances . . . 76
5.3.2 Software and Hardware . . . 77
5.3.3 Evaluation Criteria . . . 77
5.4 Pure Algorithm Performance . . . 78
5.5 Prediction . . . 80
5.5.1 Control Rule . . . 80
5.5.2 Bayesian Classifier . . . 81
5.5.3 Perfect Knowledge Classifiers . . . 82
5.5.4 Experiments: Prediction Control . . . 83
5.5.5 Summary: Prediction Control . . . 85
5.6 Continuous Control . . . 87
5.6.1 Switching . . . 87
5.6.2 Experiments: Algorithm Switching . . . 89
5.6.3 Algorithm Switching Summary . . . 93
5.7 Conclusion . . . 94

6 Low Knowledge Control for Large Neighbourhood Search 95
6.1 Introduction . . . 95
6.1.1 Scenario . . . 97
6.1.2 Outline . . . 97
6.2 Previous Work . . . 98
6.3 LNS for Scheduling . . . 99
6.3.1 Solution Representation . . . 99
6.3.2 Search Algorithm . . . 100

6.3.3 Neighbourhood Heuristics . . . 101
6.3.4 Computational Limits . . . 107
6.4 LNS Control Problem . . . 109
6.5 Experimental Details . . . 110
6.5.1 Software and Hardware . . . 110
6.5.2 Problem Instances . . . 111
6.5.3 Evaluation Criteria . . . 111
6.5.4 Best Algorithm . . . 112
6.5.5 Statistical Analysis Method . . . 113
6.6 Parameter Tuning . . . 114
6.6.1 Tuning Procedure . . . 115
6.6.2 Parameter Configurations . . . 116
6.6.3 Analysis of Results . . . 117
6.6.4 Best Parameters . . . 117
6.7 Pure Neighbourhood LNS . . . 120
6.7.1 Experiments . . . 120
6.7.2 Discussion . . . 130
6.8 Combined Neighbourhood LNS . . . 131
6.8.1 Control Methods . . . 132
6.8.2 Experiments . . . 136
6.8.3 Discussion . . . 148
6.9 Evaluation on Taillard Benchmarks . . . 150
6.9.1 Experiments . . . 151
6.9.2 Discussion . . . 162
6.10 Adaptive Large Neighbourhood Search . . . 163
6.11 Conclusion . . . 165

7 Conclusions and Future Work **167**
7.1 Contributions . . . 167
7.1.1 Low Knowledge Algorithm Control . . . 168
7.1.2 Analysis of Control Algorithms . . . 169
7.1.3 Algorithm Control Problem . . . 169
7.1.4 Control Applied to Large Neighbourhood Search . . . 170

7.2 Future Work 171
7.2.1 Application Areas 172
7.2.2 Control Mechanisms 173
7.2.3 Configuring Control Systems 175
7.2.4 Developing Algorithms for Algorithm Control 177
7.3 Conclusion 178

A Detailed Results on Taillard's JSP Instances **180**

Bibliography **184**

List of Figures

2.1 Map of Munster and a corresponding graph representing a map colouring CSP. 12
2.2 Example solution to the map colouring CSP. 13
2.3 Two different solutions to the map colouring CSP. 15
2.4 Solution to a job shop scheduling problem shown as a Gantt chart. 17
2.5 Example of constructive search exploring a search space. . . . 19
2.6 Example of edge-finding propagation. 22
2.7 Example of local search exploring a search space. 23

3.1 Information and control flow of an algorithm control system. 30
3.2 Off-line knowledge control system. 36
3.3 Example of algorithm performance varying with problem size. 44
3.4 On-line knowledge control system. 52

5.1 Predictive and switching algorithm control paradigms. 67
5.2 Graph representing a solution to a JSP. 73
5.3 Gantt chart of the example JSP showing the critical path. . . 73
5.4 Bayesian classifiers. 81
5.5 Algorithm switching. 88

6.1 Time window neighbourhood heuristic. 101
6.2 Resource load neighbourhood heuristic. 103
6.3 Pure neighbourhood heuristics run independently on the 20x20 problem set. 121

6.4 Pure neighbourhood heuristics run independently on the 40x40 problem set. 122
6.5 Best solutions found by pure neighbourhood heuristics. 123
6.6 Weight decay of *AdaptP* and *AdaptR* mechanisms. 135
6.7 Combined neighbourhood heuristics run on the 20x20 problem set. 137
6.8 Combined neighbourhood heuristics run on the 40x40 problem set. 138
6.9 Best solutions found by combined neighbourhood heuristics. . 139
6.10 Pure neighbourhood heuristics run on Taillard's problem set. . 152
6.11 Best solutions found on Taillard's problem set. 153
6.12 Combined neighbourhood heuristics run on Taillard's problem set. 154
6.13 Comparison of SA-LNS to our methods. 164

List of Tables

2.1 Example CSP. 13
2.2 Example of a job shop scheduling problem with 3 jobs and 3 machines. 16

5.1 Mean fraction of problems in each learning problem set for which the best solution was found by each pure algorithm. . . 79
5.2 Mean relative error of solutions found by each pure algorithm on the learning set. 79
5.3 Mean fraction of best solutions found by each prediction algorithm on the test set. 84
5.4 Mean relative error of solutions found by each prediction algorithm on the test set. 84
5.5 Mean relative error of each prediction algorithm for different time limits over all problem sets. 86
5.6 Mean fraction of best solutions found by switching techniques on the test set. 90
5.7 Mean relative error performance of switching techniques on the test set. 90
5.8 The mean relative error of variations of the switching algorithm at different time limits. 91
5.9 Mean relative error of the best switching algorithm against perfect knowledge prediction for different time limits. 93

6.1 Example of neighbourhood heuristic time slicing and neighbourhood search limit. 108
6.2 Tuning performance results for 20x20 problems. 118

6.3 Tuning performance results for 40x40 problems. 119
6.4 Count of significant differences over *30 time points* for pure neighborhood heuristics and *BestMRE* and *BestInstance* of pure neighborhood heuristics. 126
6.5 Mean difference over *all time points* for pure neighborhood heuristics. 128
6.6 Mean difference over *all time points* for pure neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics. 129
6.7 Count of significant differences over *30 time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics. 142
6.8 Mean difference over *all time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics. 144
6.9 Count of significant differences over *30 time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of combined neighbourhood heuristics. 145
6.10 Mean difference over *all time points* for combined neighborhood heuristics. 147
6.11 Mean difference over *all time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of combined neighbourhood heuristics. 149
6.12 Count of significant differences over *30 time points* for pure neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics. 156
6.13 Mean difference over *all time points* for pure neighborhood heuristics on Taillard's benchmarks. 157
6.14 Count of significant differences over *30 time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics. 158
6.15 Mean difference over *all time points* for combined neighborhood heuristics. on the Taillard problem set. 159

6.16 Count of significant differences over *30 time points* for combined neighborhood heuristicsagainst *BestMRE* and *BestInstance* of combined neighbourhood heuristics. 160
6.17 Mean difference over *all time points* for combined neighborhood heuristicson Taillard's benchmarks. 161

A.1 Legend for Results in Appendix A. 181
A.2 Taillard 20x15 181
A.3 Taillard 20x20 181
A.4 Taillard 30x15 182
A.5 Taillard 30x20 182
A.6 Taillard 50x15 182
A.7 Taillard 50x20 183
A.8 Taillard 100x20 183

Abstract

The central thesis of this dissertation is that *low knowledge* control methods allow non-experts to achieve high quality results from optimization technology. This is achieved through the use of algorithm control methods that automatically determine the best performing algorithms for a particular problem instance. We create and investigate mechanisms for algorithm control that do not require detailed knowledge of the problem domain or algorithm behaviour. These low knowledge control methods make decisions based only on observations of algorithm performance. The strong performance observed is not the primary result; it is that these results are possible using simple general control methods that do not require significant effort and expertise to implement.

We develop a framework for algorithm control methods and use it to present an analysis of existing work and highlight some concerns regarding the practical use of these control methods. In particular, we critique the use of high knowledge models of algorithm behaviour for shifting the expertise requirement from algorithm development to building models of algorithm behaviour. Furthermore, such models are often brittle, so cannot cope with change or be generalized to other problem domains or algorithms.

The investigation of our thesis applies low knowledge control methods to scheduling algorithms, both search algorithms and large neighbourhood search heuristics, and compares performance against optimal high knowledge algorithm selection methods. We also present a tuning procedure for configuring control systems that combine multiple algorithms. The best performing low knowledge control methods perform well across all problem sets and time limits and do not require significant effort or expertise to implement.

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 been 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.

1. Carchrae, T. & Beck, J.C., Principles for the Design of Large Neighborhood Search. *Journal of Mathematical Modelling and Algorithms*, 8(3), 245-270, 2009

2. Carchrae, T. & Beck, J.C., Applying Machine Learning to Low Knowledge Control of Optimization Algorithms. *Computational Intelligence*, 21(4), 372-387, 2005.

3. Carchrae, T.& Beck, J.C., Cost-based Large Neighborhood Search. *Workshop on the Combination of Metaheuristic and Local Search with Constraint Programming Techniques*, 2005.

4. Carchrae, T. & Beck, J.C. Low Knowledge Algorithm Control. *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, 2004.

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

Dedication

This dissertation is dedicated to Duncan.

Acknowledgements

I am extremely grateful to Chris Beck for his guidance and support. I was very fortunate to have such a dedicated and thorough supervisor. I would like to thank Gene Freuder for providing the amazing environment of 4C. The quantity and quality of researchers I was exposed to was overwhelming.

I would also like to thank Jim Bowen for making this come to be. I arrived at the university in the same year (1993) as Jim and we met through his first year computing course, where he was teaching first order predicate calculus to bewildered undergraduates. He invited 'bored' students to contact him to work on more interesting stuff; I was the only one. Together we dissected Dynamic Backtracking (a not-so-gentle introduction to constraints) and Jim taught me the fundamentals of CSPs. In the following years, Jim formed the Constraint Processing Group, won a significant European research grant to apply constraints to design, and planted the seed of what would become 4C. Some years later, Barry O'Sullivan contacted me with the news of the inception of 4C. I left industry and came back to do my PhD in what had become a world-class lab in constraints.

There are many people at 4C who helped either directly or indirectly. Eleanor O'Riordan, thank you so much for helping me to submit my dissertation. James Little, thank you for your help through the ACOMME project. Ken Brown, thank you for being my internal examiner and your assistance through the examination process. For the talks, discussions, and good times, I thank the many others.

I am grateful for the financial support of the government of Ireland for my research, directly by IRCSET to the ACOMME project, and indirectly to SFI for providing the facility and researchers of 4C.

Last and not least, thanks to my family for their love and support, and most importantly, patience.

Chapter 1

Introduction

The central thesis of this dissertation is that *low knowledge* control methods for optimization algorithms allow non-experts to achieve high quality results from optimization technology. The primary motivation for our research is to extend the reach of optimization technology by making it more accessible. To this end, we are interested in methods that not only provide high quality solutions, but do so without the requirement for significant effort and expertise on the part of a practitioner who wants to use off-the-shelf methods to solve a problem.

Traditional *high knowledge* approaches [21, 42, 68, 81, 83, 86, 94, 117] to the algorithm selection and control problem have focused on building models of problem structure and algorithm performance. While high knowledge approaches have had success on specific classes of problems, they have not succeeded in making optimization technology easier to use in general. The use of such models has shifted the expertise requirement from algorithm engineering to feature engineering and predictive model building. In contrast, our methods make control decisions based on very general features that are independent of the type of problem being solved and the algorithms that are being employed. The *low knowledge* feature explored in this dissertation is the improvement of solution quality over time, which we believe is useful in many applications. Our empirical studies show that these control methods achieve strong performance on a range of problems, including problems that

are far larger than typical academic benchmarks and more like those seen in industry. The strong performance is not the primary result; it is that these results are possible using simple general control methods that do not require significant effort or expertise.

In particular, in this dissertation:

- We create and investigate mechanisms for algorithm control that do not require detailed knowledge of the problem domain or of algorithm behaviour. These low knowledge control methods make decisions based only on observations of algorithm performance. We claim these approaches reduce the engineering effort and required expertise to effectively apply optimization algorithms.

- We compare a low knowledge control approach to idealized high knowledge approaches in the domain of scheduling algorithms. We present an analysis against the best possible high knowledge approaches and observe competitive performance. Although such methods have been shown to provide good performance, we claim they are impractical on the basis that they shift expertise from analysis of algorithm performance to an analysis of high knowledge models.

- We apply low knowledge control methods to a large neighbourhood search configuration for solving industrial-sized scheduling problems. The control mechanisms are applied to the selection of neighbourhood heuristics during search. In addition, a tuning procedure for combining neighbourhood heuristics is presented. The best performing control methods perform well across all problem sets and time limits.

1.1 Motivation

This thesis is motivated by the observation that, often, modelling is easy but solving is hard. In practice, modelling becomes hard because we want solving to be easy and the two aspects of problem solving are certainly dependant on each other [101]. Commercial libraries of problem solvers are sold under the

premise that they will solve your model using state-of-the-art algorithms. In some cases these work well, but in many cases they produce mediocre results, and in other cases they fail to produce solutions at all. So, in practice, the application of optimization technology requires the expertise to model the problem *and* to configure the algorithms to produce high quality solutions. The quality of the system is determined by the capability and experience of these experts, as shown in the applied study of Le Pape et al. [65].

In this dissertation we do not address the modelling issue. Instead, we address the issue of algorithm configuration, with the goal of producing a system that comes closer to the promise of declarative programming that 'the user states the problem, the computer solves it' [37]. The future, as we see it, is one where toolkits of algorithms are available off-the-shelf. Algorithm experts are used to build toolkits rather than configure them. Such toolkits provide the raw components for an automated system to configure and apply. What remains to be seen is if this configuration exercise can be performed while reducing the reliance on an algorithm expert to effectively use this technology.

The motivation for the work in this dissertation is as follows:

1. **The success of optimization technology is hampered by lack of experts -** The primary driver for our research is that optimization technology suffers from a bottleneck of available experts to implement solutions [37, 38, 93]. Experts are required to gather requirements, create a computational model of the problem, and then develop a system to produce high quality solutions to the computational model. In this dissertation, we address the challenge of reducing the expertise required to develop a system that produces high quality solutions for a given computational model.

2. **Algorithm performance is often brittle -** In many problem domains, the world is constantly changing. Because of change, optimization systems that perform well at first, may start to fail as the problems they are solving change in subtle, or not so subtle ways. We are interested in producing an optimization system that is able to adapt to

changes in the world. Such behaviour represents a significant practical cost savings, since it is not only expensive to repair a failing system; costs are incurred, before (or if) the failure is noticed, when the decisions produced by such a system are used in operations.

3. **There is no dominating algorithm -** There is a tendency in the field of optimization to focus on average performance and claim that the algorithm that is best on average is superior to all algorithms. This has led to many papers claiming that, for a particular time limit, method X is better than method Y on problem sets A, B and C. While such results are useful as a measure of incremental improvement, there is an implicit assumption that, eventually, a single algorithm will be developed that dominates all others on all problem sets. Perhaps this is the case on particular problem instances, or even entire problem sets, and maybe across time limits as well. However, as the range of problems to be solved increases, the chance that a single method will dominate decreases [115]. This is especially true when optimization methods are applied to real world problems rather than academic benchmarks. Most methods will exhibit strong performance on some problems at some time limits. Therefore, in a fashion similar to boosting in machine learning [39, 96], we are interested in control mechanisms to combine ensembles of algorithms to produce a system that is ultimately more robust, and performs better, than a single method that performs best on average. This observation, in relation to optimization algorithms, was first made in Leyton-Brown et al. [66] and we continue this direction of research.

1.2 Overview of Dissertation

This dissertation is structured as follows. First, we present a brief review of the concepts of constraint programming and their application to job shop scheduling. This gives the context for the algorithm control problem by introducing classes of algorithms that require significant expertise to use effec-

tively. In the next two chapters, we discuss the algorithm control problem and point to failings of the existing approaches, with respect to the requirement of reducing expertise. We suggest a low knowledge approach to overcome this problem. In the remaining chapters, we explore the application of our approach and evaluate performance against idealized high knowledge approaches.

Chapter 2 reviews constraint programming and how it can be applied to solve optimization problems and, in particular, scheduling problems. We define the core concepts in constraint programming and a constraint model for the job shop scheduling problem. We then discuss three general classes of algorithms to solve constraint programming problems, which lays the groundwork for later chapters where we present concrete examples of these algorithms.

Chapter 3 presents a detailed review of algorithm control methods. First, we present a framework to classify control mechanisms based on when they make decisions and when they capture knowledge. Then we review control methods from the literature and place them in the context of this framework. We present a discussion on the benefits and limitations of each instantiation of the framework.

Chapter 4 states our thesis: that a low knowledge approach is a practical way to achieve the goal of expertise reduction. We present a discussion on the benefits of high versus low knowledge control, and show that although theoretically, a high knowledge approach is superior, in practical terms, it is inferior since it is more prone to error, in addition to requiring significantly more expertise and effort to implement.

Chapter 5 investigates the use of low knowledge control methods applied to scheduling algorithms. We present three state-of-the-art scheduling algorithms and evaluate the performance across time limits and problem sets. We show that, without the effort or expertise of knowledge engineering, the low knowledge control methods perform strongly when compared to optimal high knowledge selection methods. An ablation study is performed to determine the impact of the components of the best control method.

Chapter 6 investigates the use of low knowledge control methods ap-

plied to large neighbourhood search. We present a large neighbourhood search algorithm for scheduling and four neighbourhood heuristics and discuss the challenge of effectively configuring such a system. We introduce a low knowledge method for tuning combinations of neighbourhood heuristics. The neighbourhood heuristics and control methods are evaluated on medium (400 activities) and large (1600 activities) scheduling problems, and then on a set of standard benchmarks.

Chapter 7 concludes with a review of the contributions of the dissertation and a discussion of areas for future work.

Appendix A presents detailed results on Taillard's benchmark problems.

1.3 Summary of Contributions

The contributions of this dissertation are:

1. A framework that defines the structure of control methods through knowledge capture and control decisions. This framework provides a tool to categorize and understand the literature, resulting in an analysis of the benefits and shortcomings of on-line and off-line control approaches.

2. The introduction of the idea of low knowledge algorithm control. We propose the use of low knowledge control methods to directly address the issue of the expertise required for the effective use of optimization technology. We challenge the belief that more information leads to a better reasoning system and propose that low knowledge control systems are both easier to implement and more robust to change.

3. The demonstration that low knowledge control methods can reduce required expertise while achieving good performance. Several control methods are presented and evaluated against optimal high knowledge selection methods. Empirical results suggest that low knowledge approaches can perform as well or better than high knowledge approaches.

4. An extension of the algorithm selection problem [94] to the more general problem of algorithm control, where algorithm selection is repeated during search. The algorithm control approach allows the interleaving of knowledge capture and decision making. We show that low knowledge control methods are able to perform as well as high knowledge selection methods when applied to scheduling algorithms.

5. An analysis of the impact of components of a low knowledge control strategy through an ablation study. This analysis is executed by producing variants of the control method that lack one of the components. We show empirically that all of the components benefit performance.

6. The evaluation of low knowledge control applied to a large neighbourhood search to solve industrial sized scheduling problems. We present a tuning method for optimizing the parameters of combined neighbourhood heuristics. Two novel neighbourhood heuristics are presented: a general purpose large neighbourhood search heuristic based on solution cost impact and a heuristic that focuses scheduling effort on the resources with the highest load.

Chapter 2

Constraint-Based Scheduling

In this chapter, we present a review of constraint programming with an emphasis on scheduling. In the first section, we describe the core concepts of constraint programming. In the second section, we introduce the job shop scheduling problem (JSP), which is the problem we will focus experiments on in this dissertation. In the remaining sections of the chapter, we describe three classes of general algorithms for constraint programming. Concrete examples of these algorithms applied to the JSP appear in later chapters.

In the context of this dissertation, constraint programming is interesting due to the fact that many different algorithms can be used to solve constraint programming problems. The wide range of available algorithms leads to an algorithm selection problem to determine the best algorithm, or combination of algorithms, to solve a particular problem instance. The algorithm selection problem is considered in Chapter 3.

2.1 Constraint Programming

Constraint programming [3, 30, 95, 106–108] is a problem solving framework that separates the problem specification from the method of solving.

$$\text{Constraint Program} = \text{Model} + \text{Search}$$

The first component of a constraint program is the model, which specifies the problem using variables and constraints. Each variable has a domain of

possible values that can be assigned to the variable. Each constraint describes a relationship that must hold among some subset of variables.

Constraint programs can be used to model satisfaction problems or optimization problems. A solution to a satisfaction problem is an assignment to each of the decision variables such that all constraints are satisfied. Solutions to optimization problems must satisfy all constraints in addition to minimizing or maximizing an objective. An optimal solution will have the minimal (or maximal) value possible for the objective.

The second part of a constraint program is the search component. The search component specifies how to explore the space of possible solutions. Many search algorithms have been proposed, however most fall into two categories, constructive and local search.

Constructive search algorithms [31, 57] build a solution by making a series of decisions, such as variable v_i is assigned the value 1. If these decisions do not violate any constraints and all variables are assigned a value then a solution has been found. Typically, each decision is recorded as a choice point. If no further decisions can be made without causing a constraint violation, then a previous choice must be changed either by backtracking and changing a previous decision or by restarting search and making different decisions. Constructive search algorithms are discussed in detail in Section 2.3.

Local search algorithms [48, 108] operate by modifying a complete assignment to all variables. An initial complete assignment is produced by a fast heuristic or from a previous solution to a similar problem. It is unlikely that this initial solution will satisfy all the constraints so a process of iterative improvement commences. During each iteration, a change is made to the complete solution. This process is repeated until a termination condition is met, such as all constraints are satisfied or a computational limit has been reached. Local search algorithms are discussed in more detail in Section 2.4. An interesting hybrid of constructive and local search, called large neighbourhood search, is discussed in Section 2.5.

One of the appealing aspects of constraint programming is that the search component can be specified independently of the constraint model. For the purpose of this dissertation, we are interested in exploring the automated

selection of algorithms to solve a constraint model. In this respect, the separation of model and search is ideal since several algorithms can employed to solve the same constraint model.

2.1.1 Applications

Constraint programming approaches have had success in many application areas. The flexible modelling ability that is available with the constraint programming approach produces optimization applications with more realistic problem models than are available with, for example, standard mathematical programming approaches. There are reports in the literature of success in areas such as manufacturing scheduling, logistics, circuit design, telecommunications network design, bio-informatics and product configuration [95].

What makes constraint programming a good fit for many applications is the ability to preserve the logical structure of the problem being modelled. The decisions to be made are typically discrete such as: what sequence should these items be ordered in, what resource should be used, should a connection between two nodes exist, what options are compatible with the current configuration, and so on. The structural relationships are modelled as constraints. For example, workers are allowed a lunch break 4 to 6 hours into their shift, the circuit power capacity cannot exceed 5mW if component X is used, the telecommunication link must have a bandwidth that exceeds a minimum threshold, and so on. These features allow constraint programming to represent many real-world problems in a natural way and make the creation and maintenance of constraint models easier.

Because constraint models are declarative in nature, it is possible to combine constraints to produce a model for a new application. This design philosophy is the basis of commercial constraint programming tools that provide a toolkit of ready-to-use constraints. It is also often relatively straightforward to add new types of constraints to a model, however skill may be required to implement them efficiently.

While constraint programming has a good ability to model many of the constraints in a real-world problem, finding a solution once the problem has

been modelled may not be trivial. For this reason, the number of successful applications appears to be limited by the availability of constraint programming experts to configure efficient constraint models and solvers. An open challenge to the constraint community is to develop constraint programming systems that are easier to use [93]. The central thesis explored in this dissertation is that algorithm control methods will help to make these systems easier to use.

2.1.2 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is a model that expresses a problem using variables and constraints. Variables are used to model decisions and constraints are used to specify the relationship between variables through allowed combinations of values. Every variable has a domain of values that can be assigned to the variable. An assignment that does not violate any constraints is said to satisfy the constraints. A solution to a CSP is found when every variable is assigned and all the constraints are satisfied.

2.1.2.1 Definitions

A finite-domain constraint satisfaction problem (CSP) [3, 30, 95, 106–108] is defined by a triple $\langle V, D, C \rangle$. V is a set of variables and D is a set of domains, where each domain, D_{v_i}, is a set of possible values for each variable $v_i \in V$ and C is a set of constraints. An assignment is a tuple (v_i, d_j) that indicates that variable $v_i \in V$ takes the value $d_j \in D_{v_i}$. A compound assignment A_W is a set of assignments for the variables in set W.

Each constraint $C_W \in C$ specifies the allowed compound assignments of a subset $W \subseteq V$ of variables. If a compound assignment $A_{W'}$ contains assignments for all variables W in a constraint C_W, this assignment satisfies the constraint if the projection $P(W, A_{W'}) \in C_W$, where $P(W, A_{W'}) = \{(v_i, d_j) \mid (v_i, d_j) \in A_{W'}, v_i \in W\}$. A solution is a compound assignment A_V that contains a labelling for all variables $v_i \in V$ such that all of the constraints are satisfied by A_V.

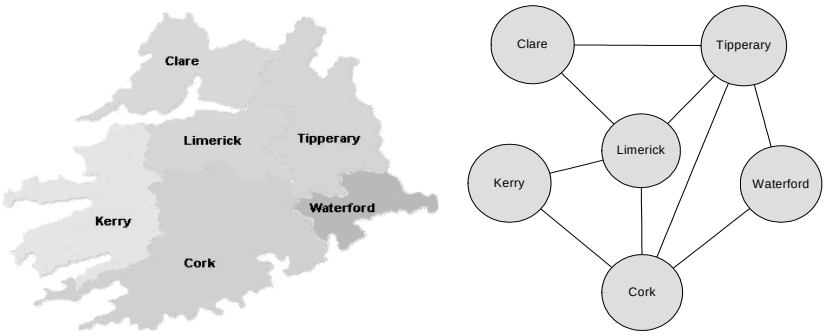


Figure 2.1: Map of Munster and a corresponding graph representing a map colouring CSP.

2.1.2.2 Example

In Figure 2.1, we show an example of a CSP in the form of a map colouring problem for the province of Munster in the south of Ireland. The map colouring problem consists of finding a colour for each county so that no two counties that share borders have the same colour. A graph representing the counties is shown on the right of Figure 2.1, where counties that share borders are connected by an edge.

From this graph, we construct a CSP that is shown in Table 2.1. The variables, V, are the nodes, representing the colour of each county. The domain of allowed values for each variable is the set of possible colours. The constraints for the problem are defined by the edges in the graph, where any nodes that are connected cannot take the same colour. To represent the constraints in this problem we state that any two connected nodes cannot share a colour, e.g. $C_{Cork,Kerry}$ where $Cork \neq Kerry$. For compactness, we represent this relationship using a not-equals constraint rather than listing out the allowed pairs of value assignments.

For example, the compound assignments $(Cork, Red)$ and $(Kerry, Red)$ violates the constraint $C_{Cork,Kerry}$. An assignment that satisfies that constraint is $(Cork, Red)$ and $(Kerry, Yellow)$. A solution to this problem is an assignment for each variable that satisfies all of the constraints simultaneously. An example of a solution is shown in Figure 2.2.

V	Cork, Claire, Kerry, Limerick, Tipperary, Waterford
D_{Cork}	Red, Blue, Yellow, White, Green
D_{Claire}	Red, Blue, Yellow, White, Green
D_{Kerry}	Red, Blue, Yellow, White, Green
$D_{Limerick}$	Red, Blue, Yellow, White, Green
$D_{Tipperary}$	Red, Blue, Yellow, White, Green
$D_{Waterford}$	Red, Blue, Yellow, White, Green
$C_{Cork,Kerry}$	$Cork \neq Kerry$
$C_{Cork,Limerick}$	$Cork \neq Limerick$
$C_{Cork,Tipperary}$	$Cork \neq Tipperary$
$C_{Cork,Waterford}$	$Cork \neq Waterford$
$C_{Claire,Limerick}$	$Claire \neq Limerick$
$C_{Claire,Tipperary}$	$Claire \neq Tipperary$
$C_{Kerry,Limerick}$	$Kerry \neq Limerick$
$C_{Limerick,Tipperary}$	$Limerick \neq Tipprerary$
$C_{Tipperary,Waterford}$	$Tipperary \neq Waterford$

Table 2.1: Example CSP.

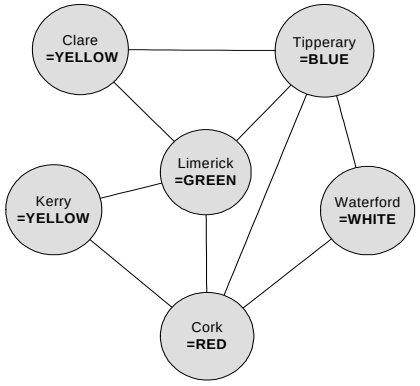


Figure 2.2: Example solution to the map colouring CSP.

2.1.3 Constraint Optimization

In many problems we wish to find a solution that not only satisfies all the constraints, but is the best solution possible (or at least a very good one) according to some objective. For example, creating a schedule that minimizes delay, a product configuration that minimizes hardware costs, a vehicle route that minimizes driving times and so on. A solution to a constraint optimization problem (COP) must satisfy all constraints and minimize an objective.[1]

2.1.3.1 Definitions

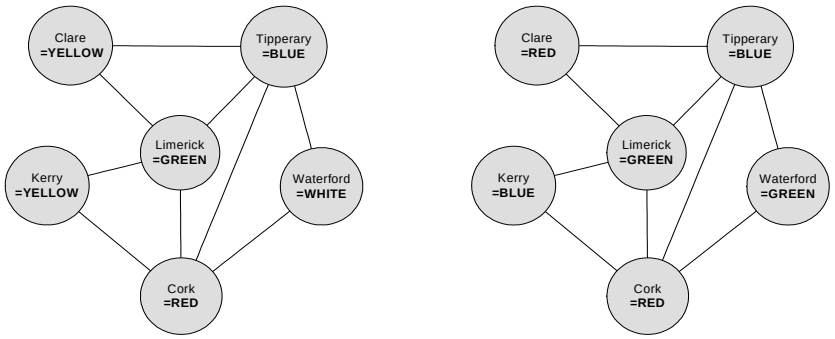
A constraint optimization problem (COP) is defined by $\langle V, D, C, f \rangle$, where V, D, and C are defined as in a constraint satisfaction problem and $f(s) \rightarrow c$ is a function that maps any solution s to a value representing the quality c of the solution. Let S be the set of all solutions to a CSP. The aim of a COP is to find the solution $s \in S$ with the minimum value $c^* = min_{s \in S}(f(s))$.

In practical terms, it is often impossible to evaluate every solution $s \in S$ and instead we return $c' = min_{s \in S'}(f(s))$ where S' is the set of solutions that have been explored. In some cases it is possible to find the optimal c^* without evaluating all $s \in S$, but in general this is not the case and c' may not equal c^* but will be the best solution found during search.

2.1.3.2 Example

Consider the coloring example from Section 2.1.2.2 where instead of simply finding a colouring that satisfies the constraints, we also wish to minimize the number of colours used. The function $f(s)$ simply counts the number of unique colours that have been assigned in a solution. We show two solutions in Figure 2.3 with an optimal solution shown on the right.

[1] In this dissertation we will always speak of minimizing the objective. It is trivial to convert an optimization problem into one that maximizes the objective. For example by stating that the objective is $max = -min$, where min is the value of the objective as a minimization problem.



The solution on the left has an objective value of $f(s) \rightarrow 5$ while the solution on the right has a value of $f(s) \rightarrow 3$.

Figure 2.3: Two different solutions to the map colouring CSP.

2.1.4 Search

The second component of a constraint program is search. Search defines the strategy used to find solutions to a constraint model. Recall that a solution is an assignment of values to all variables so that no constraint is violated.

A search space defines the possible combination of assignments, and therefore potential solutions, that a search procedure can reach. Given a CSP, the search space is the Cartesian product of the variables' domains, $D_{v_1} \times ... \times D_{v_n}$. This represents every possible assignment of values to variables. The search space therefore has a size of $O(|D|^n)$ where $|D|$ is the size of the maximum domain size and n is the number of variables.

A naive search procedure could simply generate all possible combinations and evaluate them, stopping when a combination satisfies the constraints. However, this clearly suffers as the search space is exponential with respect to problem size. For example, a problem with only 10 variables, each with 10 domain values, could require at worst 10^{10} such evaluations. Presuming that each evaluation takes 1 nanosecond, this is only 10 seconds. However, for a problem with just twice as many variables, the number of evaluations jumps to a staggering 10^{20} evaluations requiring approximately 3,000 years.

The development of search algorithms is therefore concerned with the efficient exploration of very large search spaces. The search procedures presented in this dissertation make use of techniques such as heuristics and

Job	Activity 1	Activity 2	Activity 3
A	A1 on R2 (87)	A2 on R1 (125)	A3 on R3 (100)
B	B1 on R2 (100)	B2 on R1 (62)	B3 on R3 (112)
C	C1 on R1 (75)	C2 on R2 (75)	C3 on R3 (62)

Each job has three activities that require different resources. Each activity requires a resource for the duration as specified in parentheses.

Table 2.2: Example of a job shop scheduling problem with 3 jobs and 3 machines.

propagation to direct search towards promising areas of the search space. In Sections 2.3, 2.4, and 2.5 three types of search methods are described that find solutions to constraint models: constructive search, local search and large neighbourhood search.

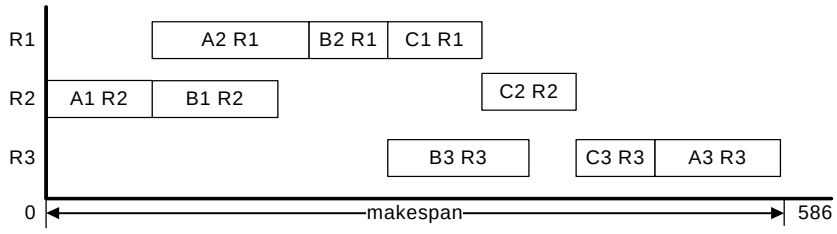
2.2 Job Shop Scheduling

We now introduce the job shop scheduling problem (JSP) which will be the central focus of experiments in this dissertation. We show how the JSP can be formulated as a CSP and a COP. In later chapters, we present constraint programming algorithms that are designed to solve the JSP.

The JSP involves scheduling n jobs across m machines. Each job j consists of m ordered activities, such that each activity is scheduled one after another. For each job, every activity requires a unique machine for a specified duration, and the sequence of these requirements differs among jobs. There are two types of constraints we must satisfy: precedence constraints between activities in a job, and resource constraints which specify that two activities may not overlap if they share the same resource.

We look at the objective of minimizing makespan: the total time required from the start of the earliest scheduled activity to the finish of the latest scheduled activity. Minimizing makespan in the JSP is NP-hard [41]. See Blazewicz et al. [17] for a survey of the work on the JSP over the past 40 years.

An example of a JSP is shown in Table 2.2 and a solution is shown as



The makespan of this schedule is 586, using the durations from Table 2.2.

Figure 2.4: Solution to a job shop scheduling problem shown as a Gantt chart.

a Gantt chart in Figure 2.4. The table specifies the order of activities in each job, the resource required by each activity and the duration of each activity. For example, A1 on R2 (87) states that the first activity of job A is on resource R2 with a duration of 87. Figure 2.4 shows a sample solution to this JSP as a Gantt chart. Each activity is scheduled so that resource constraints and job precedences are satisfied. The makespan of this solution is 586, which is not an optimal solution.

2.2.1 Constraint Model

The variables and constraints required to model the JSP are start time variables, precedence constraints and unary resource constraints. Start time variables represent the time at which an activity takes place. A precedence constraint enforces the order of activities in each job. A unary resource constraint requires that each resource can only execute a single activity at a time.

Each activity $a_{j,i}$ (the i^{th} activity in job j) has an associated start time variable, $start_{j,i}$. For convenience we define the end time $end_{j,i}$ to be equal to $start_{j,i} + duration_{j,i}$ where $duration_{j,i}$ is a constant value indicating the duration of activity $a_{j,i}$. The domain of the start time variables is the interval $[origin..horizon]$ where $origin$ is the earliest time an activity can be started and $horizon$ is the latest time an activity can be started. The range, or domain size, of the start time variables is typically large.

Precedence constraints define a precedence order of the form, $x \leq y$ where x and y are two start time variables. In the context of the JSP, precedence constraints are used to model the order of activities in a job. For a job j consisting of m activities, there are $m-1$ precedence constraints of the form $end_{j,i} \leq start_{j,i+1}$ for $i = 1..m-1$.

Unary resource constraints specify that only a single activity can be processed on a resource at a time. Logically, this can be seen as a set of disjunctive precedence constraints $(end_{j,i} \leq start_{k,l})$ *or* $(end_{k,l} \leq start_{j,i})$ for every pair of activities that require the same resource. In practice, resource constraints are often implemented using specialized propagators, such as edge-finding which is described in Section 2.3.1.1.

2.3 Constructive Search

Constructive search procedures build up a partial solution by assigning each variable, one after another, until all variables are assigned. The procedure is as follows. Given an empty set of initial assignments, $A = \{\}$, we select a variable $v_i \in V$ and a value $d_j \in D_{v_i}$. If the combination of the assignment of (v_i, d_j) and the assignments already in A satisfy all of the constraints, then we add the new assignment to A. We repeat this procedure until A contains assignments for all $v_i \in V$. If, given a set of assignments A, all of the values in the domain of an unassigned variable cause a constraint violation, then no value can be assigned. This condition is called a *dead-end* and requires that a previous assignment must be changed, a process referred to as backtracking. In the case that there are no previous assignments left to be changed, we can infer that the problem has no solution.

An example of constructive search is shown in Figure 2.5. In this example, search starts by assigning value 1 to variable X. Next, variable Y is assigned the value 2, and finally variable Z is assigned the value 1. These three steps lead to a solution, denoted by the square leaf node at the bottom of the tree. In this example, the assignment order of variables is fixed.

The SetTimes [64, 97] and Texture [8, 9] algorithms that are presented in Chapter 5 are examples of constructive search as applied to the JSP. These

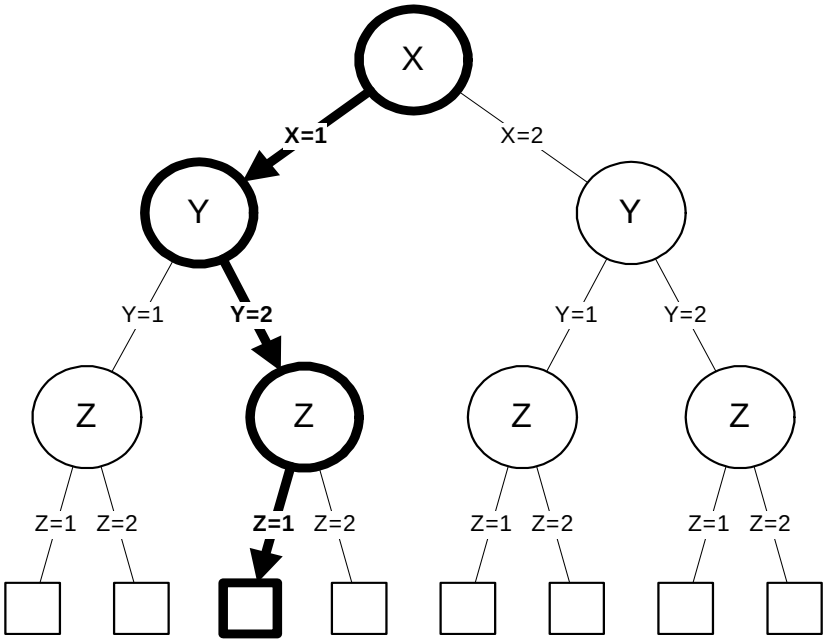


Figure 2.5: Example of constructive search exploring a search space.

algorithms represent the state-of-the-art for constructive search heuristics to scheduling.

2.3.1 Propagation

Constraint propagation is a procedure that adds constraints based on logical inference. The basic idea is that further information can be inferred from an analysis of the current assignment and the constraints in a problem. This information is used to reduce the area of search space that is explored. The most common form of constraint propagation is arc-consistency [70, 110] which can be seen as processing a single variable at a time against each constraint it is involved in. Stronger forms of consistency processing, such as path [77] or k-consistency [33, 36], have been defined however they are seldom used in practice due to efficiency reasons.

A constraint C_W is arc-consistent if, for every variable $v_i \in W$ and value $d_j \in D_{v_i}$ there exists an assignment to the remaining variables in W which satisfies C_W. That is, there exists at least one compound assignment $A_W \in$

C_W, where $(v_i, d_j) \in A_W$. If no such assignment exists, then the value d_j can be removed from D_{v_i}. A CSP is said to be arc-consistent if every constraint is arc-consistent.

The following shows an example of arc-consistency on a binary constraint. Consider two variables v_1 and v_2, each with the domain of $\{1, 2, 3\}$ and the constraint $v_1 > v_2$. We can immediately deduce that v_1 can never take the value 1, since there is no value smaller than 1 for v_2 to take. We can also deduce that v_2 can never take the value 3, since there is no value greater than 3 for v_1. So, before search has even started we can reduce the variables' domains to $D_{v_1} = \{2, 3\}$ and $D_{v_2} = \{1, 2\}$.

While polynomial algorithms exist for enforcing arc-consistency on binary constraints [70], enforcing generalized arc-consistency [75] in non-binary (or so-called global) constraints has been shown to be intractable in general [14]. However, many specialized algorithms with polynomial complexity have been developed for specific non-binary constraints [12]. Determining the appropriate effort to spend on consistency during search is an open research area in constraint programming.

Constraint propagation can be performed both before and during search. When constraint propagation is performed during search, the current assignments are propagated. In the example above, when v_1 is assigned the value 2 (which can be seen as adding a constraint that reduces the domain of v_1 to $D_{v_1} = \{2\}$), propagation can determine that v_2 must take the value 1. However, this requires maintaining the domains at each search point, since we need to restore the domains when backtracking.

2.3.1.1 Propagation of Scheduling Constraints

Two forms of propagation that useful for scheduling problems involve temporal constraints and resource constraints. Temporal constraint propagation is used to maintain the domains of the start time of variables based on precedence constraints and resource constraint propagation deduces when activities can start based on the fact that they share a common resource.

Efficient polynomial algorithms exist for the propagation of start times

using bound propagation. Precedence constraints $x \leq y$ can be represented[2] using the general form $d_{min} \leq x - y \leq d_{max}$. Such constraints can be maintained by polynomial propagation algorithms using shortest or longest path algorithms [32]. Incremental versions of these algorithms have been proposed [23, 56].

Many types of propagation algorithms have been introduced for activities that compete for resources in scheduling problems [7, 58]. The edge-finding algorithm determines if an activity A must start or end before a set S of other activities on a particular resource. If there is insufficient time available on the resource, then the constraint can reduce the domain of the start time variable.

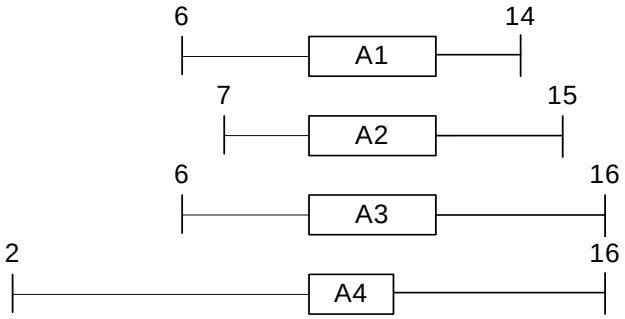
We give an example of the condition for edge-finding algorithms below. Given a set of activities S, let $dur_{sum}(S)$ represent the sum of minimal durations of S, let $end_{max}(S)$ represent the largest of the latest start times of S, let $start_{min}(S)$ represent the smallest of the earliest start times in S.

$$\begin{aligned} end_{max}(S \cup \{A\}) - start_{min}(S) < dur_{sum}(S \cup \{A\}) \\ \Rightarrow \\ end(A) \leq min_{S' \subseteq S}(end_{max}(S') - dur_{sum}(S')) \end{aligned} \tag{2.1}$$

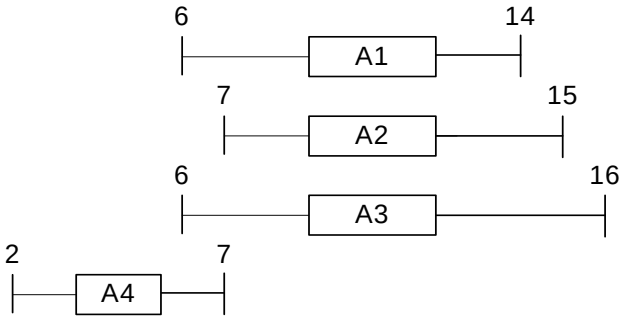
If the condition in Equation 2.1 holds then the end time of activity A can be reduced. An example where this condition holds is shown in Figure 2.6. Let $A = A4$, with a duration of 2, and $S = \{A1, A2, A3\}$, each with a duration of 3. The conditions of the Equation 2.1 are satisfied as $end_{max}(S \cup \{A\}) = 16$, $start_{min}(S) = 6$ and $dur_{sum}(S \cup \{A\}) = 11$. The edge-finding algorithm uses Equation 2.1 to compute a new upper bound on the end time of $A4$ equal to $16 - 9 = 7$ with $S' = \{A1, A2, A3\}$.

Similar conditions allow the detection and propagation of the fact that a given activity must end after all activities in S (Last), cannot start before all activities in S (Not First) or cannot end after all activities in S (Not

[2]A precedence constraint $x \leq y$ is a special case of the general form where $d_{min} = 0$ and $d_{max} = \infty$.



(a) Before applying propagation to $A4$



(b) After applying propagation to $A4$

Figure 2.6: Example of edge-finding propagation.

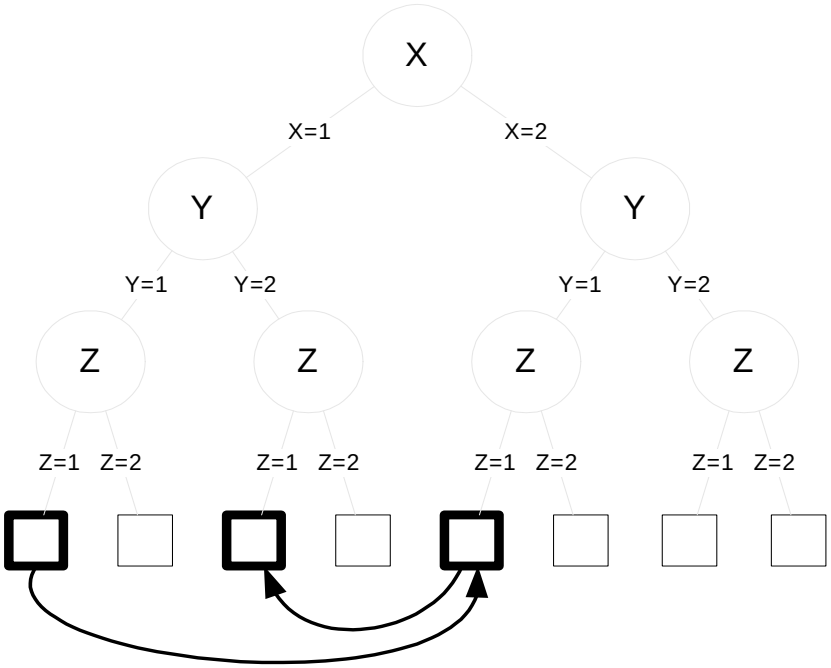


Figure 2.7: Example of local search exploring a search space.

Last). See Baptiste et al. [7] for more details. Edge-finding algorithms can be implemented to propagate on unary resource constraints involving n activities with a total complexity in $O(n \log n)$ [109]. Edge-finding methods also exist for cumulative resource constraints [72, 85] where resources may process more than a single activity at a time.

2.4 Local Search

Local search [48, 108] is a class of search procedures that make changes to a complete assignment to all variables. When search operates with a complete assignment, more information is available regarding the overall quality of an assignment, in particular if the assignment produces a feasible solution and, in the case of optimization, the value of the objective function. This is an advantage over constructive search where the outcome is not certain until all variables are assigned.

In Figure 2.7 we show an example of local search. In this example,

search starts on the leftmost leaf node, which corresponds to the assignment $\{(X,1),(Y,1),(Z,1)\}$. Next, the assignment to X is modified, moving search to the assignment $\{(X,2),(Y,1),(Z,1)\}$. Finally, search changes the assignment of X and Y at the same time and settles on the assignment $\{(X,1),(Y,2),(Z,1)\}$. The key thing to note is that search is always moving from one complete assignment to another; the search tree is not used in local search and is only shown in Figure 2.7 to make the connection with the constructive search example.

A local search procedure takes a complete assignment to all variables and iteratively changes it until a termination criteria is met. Unlike constructive search, local search maintains an assignment for every variable during search. A local search procedure starts with an initial solution s_0, that is often produced by a fast construction heuristic or a random assignment. The initial solution may violate constraints or have a poor objective value. Search then starts an iterative procedure making change to the solution using a neighbourhood heuristic that is guided by a metaheuristic. The neighbourhood heuristic is a function that, given a solution, s_i, produces a set of *neighbouring* solutions S. The metaheuristic then selects a single solution $s_{i+1} \in S$. The purpose of the metaheuristic is to guide changes in solutions toward a globally optimal solution.

Local search algorithms are very competitive methods for solving combinatorial optimization problems. They are among the top performers in a wide variety of problems [48]. A challenge in the application of local search is the design of an efficient neighbourhood heuristic and an appropriate metaheuristic. In Chapter 5 we present the Tabu-TSAB algorithm for the JSP, where more detail is provided on the mechanisms of a local search algorithm.

2.5 Large Neighbourhood Search

Large neighbourhood search (LNS) [99] is a framework that combines the search power of constructive search and propagation with the scaling performance of local search. As in local search, we modify an existing solution to the problem. However, instead of making small changes to a solution, as

Algorithm 1: Large Neighborhood Search.

```
1: N := variables in problem
2: Solution := Initial solution
3: while Termination Condition = FALSE do
4:    Using neighbourhood heuristic choose S ⊆ N
5:    Unassign S in Solution
6:    Lock the assignment in Solution for all r ∈ N\S
7:    if Search finds improvement then
8:       Solution := Update solution
9:    else
10:      Solution := Restore solution
11:   end if
12: end while
```

is typical with local search move operators, we select a subset of variables from the problem. Once variables are selected, we unassign them, lock the remaining variables to take the values assigned in the current solution, and then search for a better solution by reassigning only the selected variables.

For example, a typical local search move for a scheduling problem is to swap the order of two activities, e.g. $A_i \rightarrow A_j$ becomes $A_j \rightarrow A_i$. In the LNS framework, we remove the assignment to a subset of variables and then search for an improved assignment. Since constructive search can efficiently deal with more than two variables, many variables are typically selected when searching for improvements. In local search, a neighbourhood is explicitly defined by a set of possible moves while in LNS the neighbourhood is implicitly defined by the possible reassignments to the selected variables. Methods that select sets of variables to search are called neighbourhood heuristics.

The central idea of LNS is very simple and presented in Algorithm 1. Starting with a suboptimal solution to the entire problem, we select a subset S of variables (the neighbourhood) and perform a search which only changes the values of those variables; all other variables ($N\backslash S$) keep their assignments. By focusing efforts on improving only a part of the solution, we restrict the size of the search space and intensify search to improve the current solution. The key benefits are that we can exploit the strong propagation techniques of constructive search while avoiding the weaknesses inherent in

exploring a single search tree in a large problem space [45].

2.5.1 Search Methods

Once a part of the current solution has been selected by a neighbourhood for improvement, a search method is used to explore that space. In the case of LNS, any method that rebuilds from a partial solution is possible. In this dissertation we use a constraint programming search method Texture that is described in detail in Section 5.2.2. We note that for search to be effective in LNS, it should find solutions quickly or give up. Although search can be used to prove that no improving solution exists in a neighbourhood, such a proof is expensive and the computational effort may be better used instead to explore many more neighbourhoods.

2.5.2 Neighbourhood Heuristics for LNS

The choice of variables to search, the neighbourhood, is crucial to the performance of LNS. We wish to select variables which are likely to reduce the cost of the current solution but we are also concerned with the number of variables in each search: selecting fewer variables results in a quicker search but also reduces the likelihood that an improved solution exists within the sub-space that is explored.

Formally, a neighbourhood heuristic is a function that returns an ordered list of sets. Each set corresponds to a neighbourhood: a selection of variables to explore. The order of the list is typically the order in which neighbourhoods are explored. Many neighbourhood heuristics make use of the current solution and are reset when a new solution is found, thereby producing a new list of neighbourhoods to explore based on the new solution. Other heuristics, such as random selection, do not require any knowledge of the current solution.

A purely random neighbourhood [43, 84] can be effective on some problems. However, random selection may not choose activities that are related and instead relies on chance to select a subset of activities. It is possible that

the selection will include extra activities, which can slow down search. Despite this disadvantage, the random approach has the benefit of being general since it does not exploit any knowledge of the problem.

Scheduling specific neighbourhoods exploit the structure of the problem to select activities. Examples of these neighbourhoods are the time window neighbourhood that selects all activities whose current start times are in a particular interval and the resource neighbourhood that selects all activities on a set of resources. Further details of LNS neighbourhoods are given in Chapter 6.

2.6 Summary

In this chapter we introduced the constraint programming approach to problem solving. We have defined a specific instance of a constraint problem, the job shop scheduling problem, and presented a constraint model to represent it. In the later chapters of this dissertation, we introduce specific algorithms to find solutions to the job shop scheduling problem. As we will see, there are many such methods available, and it is not always clear which algorithm should be applied. In the next chapter, we present a deep review of control methods that have been developed for the purpose of choosing the best algorithm or set of algorithms. This review sets the context for the primary contribution of this dissertation.

Chapter 3

Algorithm Control Review

In situations where algorithm performance varies depending on the problem instance being solved, it is unclear when to use a particular algorithm. In this chapter, we discuss the idea of algorithm control, where we consider the task of improving performance when solving a problem instance by choosing the most effective algorithm(s).

3.1 Introduction

The purpose of an algorithm control system is to improve performance by choosing the most effective algorithm(s). An algorithm control system is comprised of a set of algorithms, methods to gather knowledge of algorithm behaviour, and a control system that selects an algorithm and executes it. In this chapter we introduce a framework to represent algorithm control systems where each system is classified by how it gathers knowledge of algorithm behaviours. After describing our framework, we present a review of existing work and place it in our framework.

3.2 The Algorithm Selection Problem

The algorithm selection problem [94] is to determine the best performing algorithm for a problem instance. Given a set of algorithms A capable of

solving problems in the class K, let $p(a, k)$ be the performance of algorithm $a \in A$ on a problem instance $k \in K$ where higher values of p indicate better performance. The goal is to find a mapping $S(k) \rightarrow a^*$ such that the performance of a^* is the best that can be achieved, $p(a^*, k) \geq p(a, k) | \forall a \in A$.

There are no restrictions on what constitutes an algorithm. The only important characteristic of an algorithm is the availability of an accurate performance measure for the algorithm when solving a particular problem instance. Likewise, no restrictions are placed on the problem instances in the class K.

The user of the system chooses the measure $p(a, k)$ of performance. For example, in constraint satisfaction the user may choose to minimize the computation time in order to find a solution. In optimization, the performance criteria could be the quality of the solution or the speed at which the solution improves. In a system with limited memory, the performance criteria could be the amount of memory used.

Given unlimited computation time, it is trivial to solve the algorithm selection problem: the optimal selection can be found by simply executing each algorithm and then returning the best performer. However, in most cases, computation time is one of the performance criteria to reduce. In many cases, the purpose of algorithm selection is to simply maximize algorithm performance rather than choose the best algorithm. The goal is to boost solving performance when compared to using the same algorithm for all problem instances.

3.3 Framework

In this section, we present a framework for analyzing algorithm control systems. While many algorithm control systems have been proposed, each utilizes different algorithms, knowledge and control decisions. The purpose of this framework is to enable a comparison among these systems, focusing on the differences in how such systems gather knowledge and make control decisions.

In our framework, an algorithm control system is characterized by two

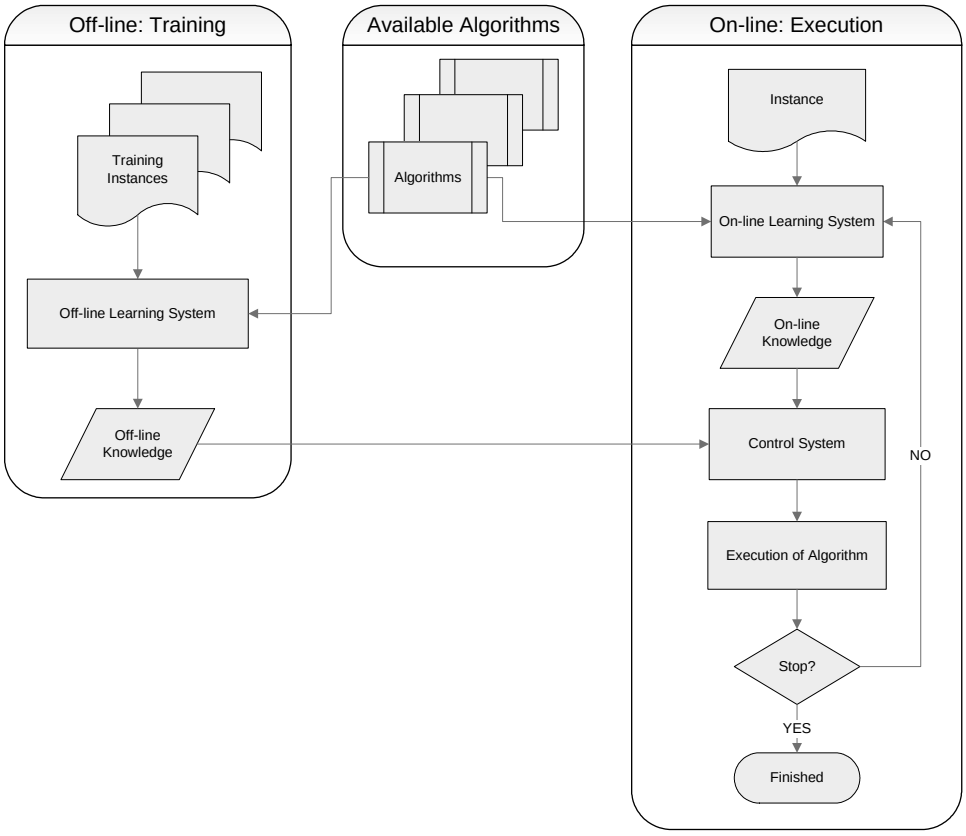


Figure 3.1: Information and control flow of an algorithm control system.

aspects: when knowledge is gathered and how control decisions are made. Gathering of knowledge relates to how the system learns information about algorithm performance on problem instances. Knowledge is used to make algorithm control decisions. Knowledge gathering may be done off-line during a training phase, on-line while solving a problem, or a combination of both. A control decision determines how to select and apply the available algorithms. Control decisions can be made once or multiple times while solving a problem.

In Figure 3.1 we show the central components of our algorithm control framework. An algorithm control system is an instantiation of some, or all, of these components. On the left, we show the off-line components which allow knowledge to be gathered during a training phase. In the off-line training phase, the system learns how algorithms perform on the training set of problem instances. We refer to information gained during the off-line training

phase as off-line knowledge. On the right, we show the on-line components responsible for selecting an algorithm. On-line knowledge can be gathered at any point in the on-line process. What makes on-line knowledge unique is that it is specific to the the problem instance being solved. The on-line control system considers all of the available knowledge and decides which algorithm to execute.

Below we list some instantiations of this framework. We start with the minimal instantiation, no knowledge, followed by several examples which rely on different components.

No Knowledge The system is presented with a set of algorithms and randomly chooses one to apply to a given problem instance. No off-line training is required and no on-line knowledge is used. The control system makes a blind guess regarding which algorithm to use and executes it once.

Off-line Knowledge Only A training set of problem instances is provided. The system learns which algorithm performs best on the training set. The off-line knowledge, in this case, is the best performing algorithm on the training set. When the system is applied on-line, it makes the control decision using the off-line knowledge only; that is, it uses the same choice of algorithm for all problem instances. Since the same off-line knowledge is used for each decision, this decision has been determined off-line.

On-line Knowledge Only No training phase is required in this case. Initially, the system chooses an algorithm randomly. After the algorithm has been executed for a short time, the system measures the performance of the selected algorithm. If the algorithm has performed well, the system applies the algorithm again, otherwise the system randomly selects a different algorithm to apply and repeats this process.

Off-line and On-line Knowledge In the training phase, the system measures each training problem instance to extract some feature values. It then executes all available algorithms on each training instance. A

prediction model is built which correlates feature values with the best performing algorithm. This model is the output of the training phase, and embodies the off-line knowledge in this example. When the system is used on-line, it extracts feature values from the problem instance. On the basis of these feature values, the model predicts which algorithm will perform best and the system chooses that algorithm.

In Section 3.3.1, we discuss the nuances of each type of knowledge and the implications for algorithm control systems. In Section 3.3.2 we discuss control decisions in more detail.

3.3.1 Gathering Knowledge

We characterize the gathering of knowledge based on off-line versus on-line experience. Knowledge gathered off-line is collected during a pre-processing phase while on-line knowledge is gathered only when the system is presented with a particular problem instance. Since off-line knowledge is collected before problem solving, significant processing time may be available to collect it. The opposite is true for on-line knowledge. On-line knowledge is collected only once the system has been presented with a problem instance, so the effort taken to gather on-line knowledge must be considered as part of the overall effort to solve the problem instance.

3.3.1.1 Off-Line Knowledge

Off-line knowledge is gathered by observing algorithm performance on a training set of problem instances. The purpose of this phase is to learn typical algorithm behaviour. For example, an analysis can be performed to gather knowledge of mean algorithm performance. Each algorithm is executed on each problem instance in the training set, and the mean performance of each algorithm is computed.

There is an implicit assumption of the generality of off-line knowledge; that performance on the training set will be similar to that on other problem instances. Thus care must be taken to ensure that off-line knowledge is

relevant to problems that are to be presented to the system later on. If algorithm performance on new problem instances differs from that seen on the training set problem instances, then off-line knowledge is unlikely to be useful.

Significant processing time may be available to collect off-line knowledge, hence accurate knowledge of algorithm performance can be gathered from experience on the training set. This form of analysis is very common in practice and can be particularly useful if one algorithm is found to dominate all others. We will describe several systems which use off-line knowledge in Section 3.4.

3.3.1.2 On-Line Knowledge

On-line knowledge is gathered only when a problem instance is presented to the system. While off-line knowledge is general, on-line knowledge is highly specific. By measuring the problem instance, knowledge is gathered that is particular to the problem instance being solved.

An example of on-line knowledge is measuring the constraint density of a particular constraint network. To measure constraint density, the actual number of constraints in a network are counted and divided by the number of possible constraints. Another example are texture measurements of Beck & Fox [9], which are used to guide heuristic search. Indeed, much work in heuristic search has focused on exploiting the structural knowledge of the problem instance.

Since on-line knowledge is gathered only after a problem instance has been presented for solving, the amount of time taken to perform measurements should be taken into account when assessing the overall problem solving performance. For this reason, most on-line measurements are relatively fast to compute. A special case of gathering on-line knowledge is to run one of the algorithms for a short time. The resulting performance can then be used to make subsequent control decisions. We discuss this case in more detail in Section 3.5.2.

3.3.1.3 Combining Off-Line and On-Line Knowledge

Many algorithm control systems make use of both off-line and on-line knowledge. The goal of combining the two types of knowledge is to create more specific knowledge of the problem instance with a greater degree of accuracy.

A common example of this approach is when off-line knowledge is correlated with problem instance features. When the system is presented with a problem instance on-line, it extracts the instance features. These features can then be used to provide an estimate of algorithm performance using the off-line knowledge.

For example, consider a simple feature such as the number of a variables in a constraint satisfaction problem. Given two algorithms that are observed during an off-line training phase, experience on the training set shows that if the number of variables is small then the first algorithm is superior, otherwise the second algorithm is superior. In the on-line phase, the control decision is based on the off-line knowledge (which algorithm is superior for a given number of variables) and the on-line knowledge (the number of variables in the current instance).

A challenge in combining on-line and off-line knowledge is how to determine relevant features. There are at least two stages to this process. First, possible features must be identified. Then the usefulness of features needs to be evaluated. While the evaluation step may be automated to some degree, it often requires significant time and expertise. However, the automatic identification of features remains a more challenging problem and often requires manual effort and domain expertise. We will discuss these issues further in Section 3.5.

3.3.2 Control Decisions

An algorithm control system is comprised of a decision making component and an algorithm execution component, as shown in Figure 3.1. The algorithm control process is executed as follows. The decision component considers the available knowledge and determines which algorithm to apply. In addition to selecting the algorithm, the decision includes a termination

condition which specifies when to stop executing the algorithm. The execution component applies the algorithm to the problem instance and monitors progress, halting when the termination condition becomes true. The execution component can gather additional on-line knowledge from the results of the algorithm execution. The system can then either stop or make another decision and repeat this process.

For example, execute *backtracking*[3] until it *finds the first solution* is a control decision. In this example the algorithm is *backtracking* and the termination condition is *find the first solution*. The execution component then applies backtracking to the problem instance until a solution is found. At this point the system could make another control decision or terminate. Another example of a control decision is execute *branch and bound* on an optimization problem until *10 minutes have passed or the optimal solution is found*. In this case, the termination condition is a time limit or a proof of optimality. The system can then make another control decision or terminate.

The details of how the decision component selects an algorithm and the choice of termination condition are specific to the particular instantiation of the framework. We give specific examples of control decision systems in Sections 3.4, 3.5 and 3.5.

3.4 Off-line Knowledge Control Systems

In a purely off-line knowledge control system, control decisions are made based only on off-line knowledge. In this section we describe three types of off-line knowledge control systems: best overall algorithm, algorithm portfolios and algorithm configurations. In all cases, these approaches determine the single algorithm, the single portfolio, or the single algorithm configuration that performs best on the set of training instances. Once the best approach has been determined, it is then applied to all problem instances that are encountered on-line.

[3]In this example, *backtracking* refers to a complete search configuration, e.g. backtracking with maintained arc consistency, the minimum domain variable heuristic and a lexographical value heuristic

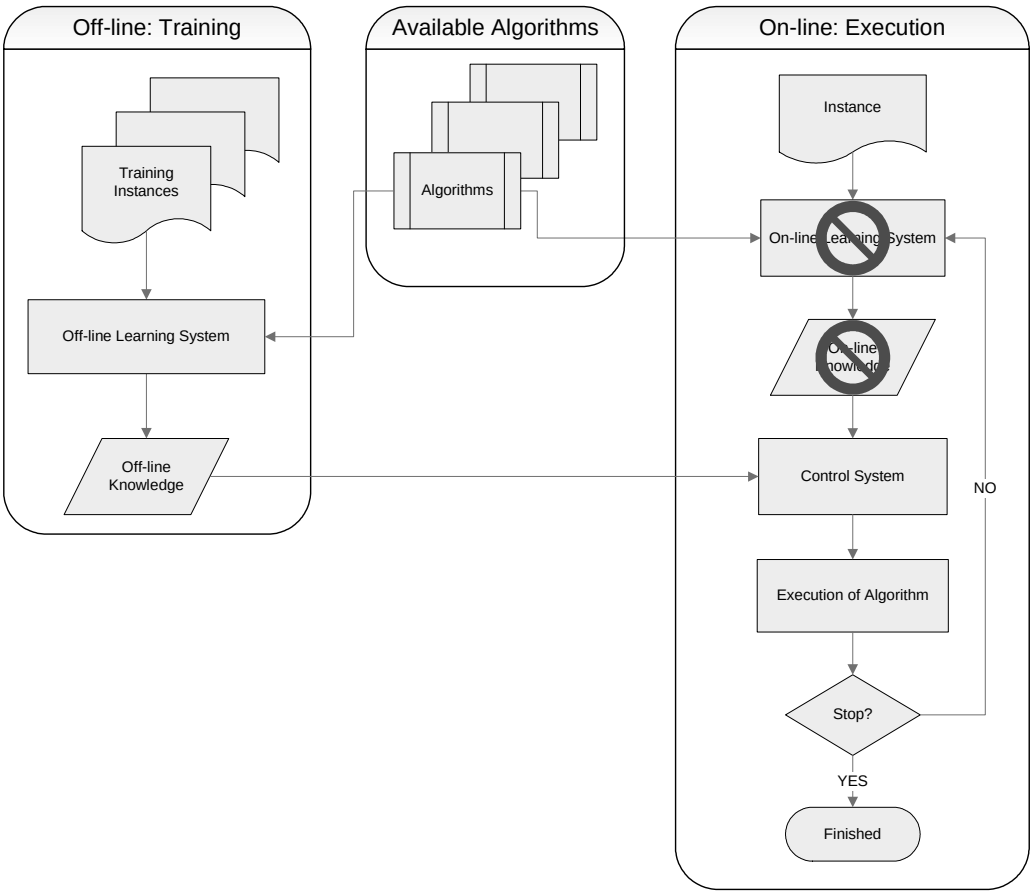


Figure 3.2: Off-line knowledge control system.

In Figure 3.2 we show the components of our framework that are instantiated to support an off-line knowledge control system. The off-line training phase gathers knowledge of algorithm behaviour based on experiences on a training set of problem instances. Once the training phase is finished, the system stops gathering knowledge. In the on-line execution phase, the system bases all control decisions on the off-line knowledge. No knowledge is acquired in the on-line phase.

One of the strengths of an off-line knowledge system is that all computation regarding selection has been performed prior to execution. This allows the maximum time for the selected algorithm to be executed since the choice of algorithm is predetermined. However, a fixed choice of algorithm can have a negative effect if algorithm performance varies significantly among problem instances, as is common for hard combinatorial problems.

3.4.1 Best Overall Algorithm

A common instantiation of the off-line algorithm control system is to choose the algorithm which has the best overall performance on a training set. The implementation of systems in practice, and indeed much experimental analysis of algorithms, has been dominated by this approach. The process of determining the best performing algorithm typically proceeds as follows. Each algorithm is executed on each training problem instance and the results of all algorithms are compared using a measurement of performance. These performance results constitute the off-line knowledge. The control decision is based on these performance results. So, in an on-line context, the system chooses the algorithm that had the best performance on the training set.

There are many ways to measure algorithm performance. While mean performance is one of the most common measures used to determine algorithm performance, median performance can be a useful measure in the case that mean performance is skewed by a only few problem instances. Standard statistical tests like analysis of variance can be used to determine the reliability of an algorithm and the range of expected behaviour. An algorithm that performs with low variance will be predictable, which is often of

great practical value to users. Typically, users expect low variance and high performance, which we refer to as a *robust* algorithm.

Implicit in the approach of choosing the best overall algorithm is the perhaps unrealistic search for a single algorithm that outperforms all others on all instances. If such an algorithm exists, then a single choice of algorithm is a sensible approach. However, for hard computational problems, it is seldom the case that a single algorithm dominates all others. The choice of best overall algorithm serves as a baseline for comparisons with more advanced algorithm control systems. As we will see, it is often better to adopt a more flexible approach.

3.4.2 Algorithm Portfolios

The use of the term *portfolio* comes from economics and relates to reducing the risk of solving combinatorial problems by diversifying computational 'investment' among several algorithms [50]. In a similar manner to investing in the stock market, the idea is to invest computation time in a set of algorithms where at least one algorithm is likely to perform well. Rather than investing all of the computation time in a single algorithm, a proportion of the available time is invested in a set of algorithms. The set of algorithms should have the property that the performance across problem instances is uncorrelated, increasing the likelihood that at least one algorithm performs well on each instance. The motivation is to decrease mean processing time while minimizing variance.

Consider, for example, the scenario where one algorithm finishes in 1 second on half of a set of problem instances and takes 100 seconds on the remaining half. Another algorithm has exactly the opposite behaviour, taking 100 seconds on the first half and 1 second on the remaining half. In this case, a single choice of algorithm will yield an average performance of 50.5 seconds and a standard deviation of approximately 70 seconds. If instead, the available processing time is split evenly between the two algorithms and both algorithms are run in parallel, the combined solving performance is only 2 seconds with a standard deviation of 0 seconds.

In this section, we consider algorithm portfolios of the following form. A portfolio refers to a set of algorithms while a *portfolio algorithm* is a proportional allocation of computation time among the algorithms in a portfolio. The portfolio algorithm is determined off-line based on the computational cost profiles gathered by running each algorithm on a training set. When a problem instance is encountered on-line, the portfolio algorithm executes the algorithms, with some algorithms getting more time than others. The optimal portfolio algorithm is one which maximizes average performance while minimizing variance on the training set. We note that when applied on-line, a portfolio algorithm is only optimal if the performance of each algorithm is similar to performance on the training set. [4]

Huberman et al. [50] were the first to use the algorithm portfolio approach. Their method of building a portfolio algorithm proceeds by computing cost profiles for each algorithm, where a cost profile is a probability distribution (of solution times) for an individual algorithm over the training instances. Once these distributions are available, an aggregate function computes the probability distribution of any portfolio algorithm which uses these algorithms. Through enumeration of possible portfolio algorithms, the one that has the maximum average performance with minimum variance can be determined.

For example, let t_1 and t_2 be random variables representing the solution time of algorithms a_1 and a_2 respectively. Each algorithm has a probability distribution, represented by $p_1(t)$ and $p_2(t)$, that they finish at a particular time t. To construct a portfolio algorithm, both algorithms are executed independently on the same processor. Let f_1 be the fraction of time allocated to algorithm a_1 and $f_2 = 1 - f_1$ be the proportion of time allocated algorithm a_2. Therefore, the solution time of the portfolio algorithm is a random variable t_p defined as:

[4] There are numerous examples in the literature [46, 67, 81] where authors refer to algorithm portfolios where only a single algorithm is chosen from a collection of algorithms. The choice of algorithm is made on-line, based on some observations of the problem instance. We review some of this work in Section 3.5.

$$t_p = \min\left(\frac{t_1}{f_1}, \frac{t_2}{f_2}\right) \tag{3.1}$$

The portfolio algorithm has a probability distribution $p(t)$ that it finishes at a particular time t. This probability is given by the probability that both algorithms a_1 and a_2 finish in time greater or equal to t minus the probability that both finish in time greater than t.

$$p(t) = \left[\sum_{t' \geq f_1 t} p_1(t')\right]\left[\sum_{t' \geq f_2 t} p_2(t')\right] - \left[\sum_{t' > f_1 t} p_1(t')\right]\left[\sum_{t' > f_2 t} p_2(t')\right] \tag{3.2}$$

The probability distribution $p(t)$ can be used to compute the mean and standard deviation for any values of f_1 and f_2. The mean represents the expected running time of the portfolio algorithm, while the standard deviation represents the risk. To determine the optimal portfolio, values of f_1 and f_2 are enumerated. There will exist at least one preferred combination which has better performance and/or lower variance than other combinations. This is an optimal portfolio and corresponds to the notion of an efficient frontier in economics. It is relatively straightforward to generalize Equation 3.2 to the case of N algorithms.

Gomes et al. [44] apply the portfolio algorithm approach to quasi-groups with holes problem and logistics planning problems. Gomes generalizes the work of Huberman to the parallel processor case. We note that the execution of algorithms on parallel processors fits easily into the framework described in Section 3.3. Petrik [91] provides a rigorous study of the theoretical aspects of algorithm portfolios and some promising experimental results in applications to SAT. Petrik provides two algorithms to compute portfolio algorithms, a heuristic method that produces good portfolios and an exact (but worst-case exponential) algorithm to find optimal portfolios.

The core idea of a portfolio algorithm is to run several algorithms concurrently. If each algorithm has a different probability and variance in finding a solution at a given time, a portfolio that combines algorithms may be able to increase algorithm performance while reducing variance. However, since

probability distributions are based on a training set of problem instances, portfolios will suffer if on-line problem instances differ from the training set.

3.4.3 Configuring Algorithms

To instantiate a particular algorithm so that it can be used, one must often specify many parameters and components. For example, to build a constraint programming search procedure, several components must be chosen such as the search heuristics, propagators, and tree traversal algorithms. In addition, each of these components may take parameters, such as the number of backtracks before a randomized restart is applied. Every combination of components produces a unique algorithm, thereby resulting in an algorithm selection problem with a very large number of possible algorithms. With so many choices, it is typically impractical to evaluate all of them and a more focused low-level approach is used instead.

The following systems have been presented to configure algorithms:

- The Multi-Tactic Analytic Compiler (MULTI-TAC) [74] searches through combinations of algorithm schemas, heuristics and propagation levels.
- The Adaptive Constraint Engine (ACE) [34] learns weights for 'advisors' which represent different heuristics and then makes decisions using a voting scheme in the spirit of combining machine learning classifiers.
- Bain et al. [5] use genetic programming to develop novel heuristics by combining primitive measures of problem instances with function trees of numerical operators.
- The F-Race algorithm [15] evaluates possible algorithm configurations and provides statistical guarantees about poorly performing configurations and when to stop evaluating them. In this system, all possible algorithm configurations are provided to the system for evaluation.
- The ParamILS procedure [53] uses iterated local search to explore the space of parameter settings for any type of algorithm.

All of these approaches take a similar form. A training set of problem instances is presented, and an incomplete search of possible configurations is performed, evaluating configurations by executing them on the training set. Since there are too many configurations to perform a complete search, an incomplete search is used. For example, MULTI-TAC uses a beam search to extend the best known configurations. Eventually, the system should converge on a configuration that performs well on the training set. It is hoped, then, that this same configuration will perform well on new problem instances of the same class.

Since many algorithm configurations must be evaluated, and each configuration is evaluated by execution on the training set, these systems often take a very long time to converge on a good configuration. Indeed, authors will often make the argument that configuration can be a lengthy process and, since it is off-line, it is reasonable for it to take a very long time. In many cases, the work can be performed in parallel.

These algorithm configuration approaches are in fact an instance of finding the best overall algorithm, as previously described in Section 3.4.1. They key difference is that these approaches are searching a much larger space of possible algorithms. Instead of testing every possible combination, some form of intelligent guiding is applied so that only promising algorithms are evaluated.

3.4.4 Conclusion

Control systems which use only off-line knowledge are very common in practice. We have shown three types of off-line knowledge control systems: best overall algorithm, algorithm portfolios and algorithm configurations. In all cases, these approaches choose the single algorithm, the single portfolio, or the single algorithm configuration that performs best on the set of training instances. This choice is then applied to all instances encountered on-line.

A feature of off-line algorithm control is that algorithm performance is treated as a black box. That is, a problem instance is presented to an algorithm and the only information that is collected is the algorithm's perfor-

mance on that instance. This is a strength of off-line algorithm control since it does not need to be concerned with determining useful problem instance features, and indeed, the black box nature of the approach makes an off-line algorithm selection method more generally applicable.

However, this feature is also a shortcoming of off-line algorithm control. During on-line execution, the same control decisions must be used on every problem instance presented since no information is gathered about the problem instance being solved. In many cases it may be possible to do better on a problem instance by using a control policy that considers the current problem instance being solved.

Finally, we reiterate a major assumption of purely off-line knowledge approaches: that the training problem instances are similar to those encountered on-line. A system that performs very well on the training set instances may perform poorly as the problem instances change. The best choice of algorithm, portfolio, or configuration may change over time. While it is the case that these new problem instances can be used to develop a new training set and the off-line knowledge gathering phase can be restarted with the new training set, it is not clear when this process should be repeated and there may be a significant expense incurred while the system performs poorly.

3.5 Off-line and On-line Knowledge Control Systems

In this section, we look at systems that combine off-line and on-line knowledge. Recall that off-line knowledge can be very accurate, with respect to the training instances, since significant time can be spent performing analysis during an off-line training phase. However, when a new problem instance is presented to a control system, it may not be clear what off-line knowledge, if any, is relevant. On-line knowledge, extracted from the problem instance that is being solved, provides a way of specializing control to make decisions that are more specific to the problem instance.

Figure 3.1 from Section 3.3 shows the components required by this type

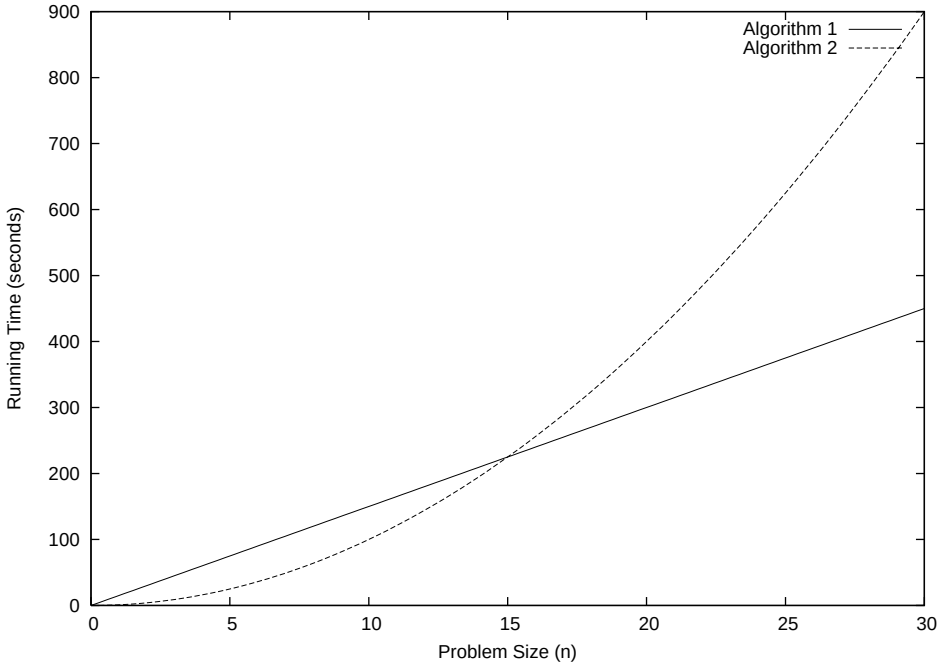


Figure 3.3: Example of algorithm performance varying with problem size.

of system. An off-line training phases gathers knowledge based on a training set. On-line knowledge is gathered from the problem instance that is being solved. In the following sections, we review four research themes that combine off-line and on-line knowledge.

3.5.1 Empirical Performance Models

Empirical performance models [66, 80, 81] approximate the performance of an algorithm using features extracted from a problem instance. To build a model, relevant features are identified and correlated with algorithm performance. Empirical performance models are specific to the knowledge gathered by executing a particular algorithm on a training set of problem instances. Given a performance model for each algorithm, on-line control decisions are made by extracting features from the problem instance and predicting the performance of each algorithm . The algorithm that is predicted to have the best performance is then selected.

Consider the scenario where two algorithms have the following behaviour. *Algorithm 1* solves problems in $t_1 = n \times 15$ seconds, where n is the size of the problem.[5] *Algorithm 2* solves problems in $t_2 = n^2$ seconds. The results of running these algorithms are shown in Figure 3.3. For any problems with n less than 15, *Algorithm 2* is the best performer, otherwise *Algorithm 1* is best. In this example, the problem instance feature is size and the predicted value is running time.

Complexity results for many algorithms are available but they are typically a worst-case analysis. While there has been some progress on the analytical study of average-case performance [35], few results are available for state of the art algorithms for hard combinatorial problems. A promising practical direction for predicting the performance of an algorithm given a certain problem instance, is the work on empirical performance models.

Nudelman et al. [66, 80–82] use linear regression to approximate running times by developing empirical performance models for several computationally challenging problems such as SAT and the winner determination problem for combinatorial auctions. To enable linear regression to be more effective, significant effort is required to determine useful features. For example, the authors suggest adding extra features that are derived from other features, such as new features that are the pairwise products of other features. However, as the number of features grows, the performance of linear regression deteriorates, so care must be taken to select useful features.

Once a set of useful features have been identified, an approximation function can predict an algorithm's runtime on a new problem instance. When a new problem instance is presented, the approximation function for each algorithm is applied to the new instance to predict each algorithm's running time. The algorithm with the shortest predicted running time is then chosen. The authors also apply this approach to the case of optimization with a fixed running time. In this case, the prediction is the overall quality of the solution, and they choose the algorithm with the best predicted solution quality for the given time limit.

[5] For example, size might be the number of variables in a constraint satisfaction problem, or the number of elements in a list to be sorted.

The idea of creating an approximation function based on problem feature values is the basis of much work in machine learning. Leyton-Brown et al. [68] evaluated other machine learning approaches aside from linear regression, yet none yielded superior results and some methods took considerably more computation time. Mitchell[6] noted that while the choice of learning algorithm is important, the choice of model (i.e., the features and value chosen for approximation) is more important and will dictate the performance of any machine learning method. In other words, the problem of feature engineering is what makes empirical performance models challenging to apply in practice.

3.5.2 Classifying Problem Instances

Rather than trying to compute an approximation function to predict runtime, a classification approach tries to directly predict the algorithm that will perform best on a problem instance based on previous experience on similar problem instances. In this context, a class represents the set of problem instances that are best solved by a particular algorithm. Hence, there exists a class for each algorithm. In this section, we discuss methods that classify a problem instance: two case-based reasoning approaches, a decision tree approach, and a simple classifier based on short (on-line) runs of each algorithm.

Gebruers et al. [42] developed a case-based reasoning system to select search strategies for constraint programming applied to the social golfers problem, which is a special case of a timetabling problem. Problem instances are classified using domain specific instance features. Further features are generated by creating weighted ratios of instance features. The case-based reasoning approach is compared against a decision tree built by the C4.5 algorithm with results showing that the case-based reasoning outperforms the decision tree. Their results indicate that the case-based reasoning approach outperforms the policy of using the single algorithm that was best on average

[6]Tom Mitchell, IJCAI-05 Tutorial on Representation And Learning In Robots And Animals.

over all problems.

Burke et al. [21] present a case-based reasoning system for selecting metaheuristics for university course and exam timetabling problems. Domain dependent features and ratios of features are employed in a similar fashion to Gebruers et al. Burke et al. use a case-based reasoning system in two contexts. In the first context, similar to Gebruers et al., a single algorithm is selected and applied to a particular problem instance. In second context, the selection is performed many times, a different algorithm is selected at regular intervals during problem solving. When making multiple decisions, features such as the number of unscheduled exams are included. The case-based approach outperforms any single algorithm.

Both of these case-based reasoning approaches avoid the complexity of trying to determine an accurate approximation of runtime and work directly to determine the best algorithm from a case-base of similar problems. However, they suffer from the same problem as the approximation models in Section 3.5.1: relevant features and appropriate weights are required for an accurate measure of similarity.

Beck & Freuder [10] use a different approach applied to scheduling algorithms. Rather than developing domain specific features and weights, the only feature measured is algorithm performance over time. Performance information is collected from short on-line runs of the available algorithms on the instance being solved. The only off-line training used in this case is to determine the length t_s of the short run. The method proceeds as follows. Each algorithm has a short run for t_s seconds and performance information is gathered. After the short runs are complete, the performance is examined and an algorithm is selected. The selected algorithm is then allowed to continue for the time remaining. Several rules for selecting an algorithm are explored with the most promising one being to simply choose the algorithm which has the best performance so far.

While this approach may seem simple-minded, it tackles the problem of feature engineering. Since no complex features are used, the method remains general and no engineering is required to apply it to new problem domains. However, a limitation is that as the number of algorithms grows, the time

spent executing short runs will require more of the available time. In practice, that means this approach is suited to situations where the choice is made from only a few algorithms.

3.5.3 Monitoring Search

Algorithms for hard computational problems are prone to exceptionally long running times in certain cases. A monitoring approach detects when a search algorithm has entered a long run and continuing search is unlikely to be worthwhile. When this situation is identified the control method changes the search strategy.

In many real world optimization problems, the size of the problem means that the optimal solution cannot be found in a reasonable time. Instead, computation time is traded off against the quality of the solutions found. Optimization algorithms typically find many improved solutions at the beginning of search, but as search progresses more time elapses between improvements. In these cases, a decision must be made regarding the tradeoff in time spent versus the improvement in solution quality. Larson & Sandholm [62] present a method that uses performance profile trees to decide when is the best time to stop search. In an off-line training mode, they build a performance profile tree that encodes the probability that an algorithm will find an improved solution given the past improvements it has made. On-line, improvements in solution quality are observed and the performance profile tree indicates whether it is worthwhile to continue search. Interestingly, this control system performs very well with only the simple feature of improvement in quality of solution. They report that the addition of domain specific features only gave a marginal improvement.

Adaptive constraint satisfaction [18] uses a search monitor to determine when to try the next algorithm in a prioritized list that is ordered with the 'quickest first principle'. Based on their performance on a training set, the algorithms are sorted by increasing median constraint checks, a measure of effort. When a thrashing detector indicates that an algorithm is not making progress, search is restarted using the next algorithm in the list. The

thrashing detector is called the Monitor Search Level (MSL) and operates by observing features of search node exploration when solving a problem instance. While adaptive constraint satisfaction makes use of on-line features through the MSL monitor, most of the learning has been performed off-line on a training set. Both the statistical analysis, which produces the ranked list of algorithms, and the tuning of the MSL parameters are performed off-line. On-line, the monitor simply detects thrashing and switches to the next algorithm on the list.

Combining a restart strategy with randomized search has proved to be a very useful way of solving hard problems [45]. Randomized search with restarts operates on the observation that in many problems a solution is found in either a relatively short time or in an exponentially long time. By restarting search after a short time has elapsed, the aim is to avoid a long run. However, determining exactly when to restart search remains a challenging question. Horvitz et al. [49] use an approach that measures problem instance features during search to determine when to restart by predicting if search will take a long or short time. In an off-line training mode, they use Bayesian structure learning to infer predictive models from the training set results. A heuristic search is performed over the possible models and a Bayesian score is used to identify a model with the greatest ability to predict the training data. Once this prediction model has been created, it is used on-line to determine when to restart search. At each search node, they measure the problem instance being solved and predict if the algorithm will have a short or long running time. If the current search is predicted to take a long time, then randomized search is restarted with a new random seed.

Monitoring the progress of search appears to be a promising research direction. By observing how search progresses, more informed control decisions can be made that are specific to the instance currently being solved. Instead of trying to determine whether an algorithm will perform well *before* solving a problem instance, a search monitor makes control decisions based on how an algorithm actually performs. The fact that, in some cases, informed decisions can be made without requiring problem specific features points to an interesting opportunity to avoid the feature engineering issues discussed in

Sections 3.5.1 and 3.5.2.

3.5.4 Control Policies

A control policy determines the best action to apply given the current state. Actions relate to applying a particular algorithm, while states refer to the possible types of problems and stages of problem solving. Using a representation of actions and states appears to be a natural way to view the algorithm control problem. However, while actions are relatively straightforward in this model, representing the state space within a reasonable memory limit proves to be a more difficult challenge.

Lagoudakis & Littman [61] create a control policy for sorting algorithms. The problem of sorting a list is modelled as a Markov decision process (MDP) where the state space represents the number of elements in the list to be sorted and actions are the application of available sorting algorithms. This work is particularly interesting as multiple decisions are made. Since some sorting algorithms are recursive in nature, control decisions can be made multiple times. For example, merge sort splits a list in half requiring that each half is sorted and then merged together. A different algorithm can then be applied to each half of the list. The application of multiple algorithms adds an interesting wrinkle to the typical MDP since, when splitting a list, two actions must be executed creating two new states. A modification to the Q-learning algorithm is proposed to support this departure from standard MDPs. The MDP learning process consists of learning a reward function, $R(a, s)$ that indicates the benefit of applying action a in state s. Once the reward function has been learned, during an off-line training phase, it be used to produce a control policy for use on-line. The approach shows impressive performance, indicating that the control policy outperforms any single algorithm.

While the application of MDP methods to algorithm control is an exciting area of research, sorting lists is not a particularly interesting problem domain. In a subsequent work, Lagoudakis & Littman [60] applied a similar approach to backtracking algorithms for SAT problems. In this case, the actions are branching heuristics. The authors appear to have tried many

different state representations based on features such as the number of variables, number of clauses, number of literals, minimum size of clauses, number of minimum size clauses and ratio of variables to clauses. However, they note that as more features are added to the state representation, the state space grows exponentially making the learning task more challenging. After careful analysis, they settled on using the number of variables as the state representation. Their results show that the control policy performs as well as the best heuristics, but no better, and suggest that a better state representation could improve this situation.

The application of control policies, such as techniques for MDPs, is an appealing direction for future work. However, again we see the challenge of determining useful problem features. While the work presented above indicates the feasibility of MDP methods, it also demonstrates the problem of compact state representations that are required before these methods can be successfully applied to more complex problem domains.

3.5.5 Conclusion

We have presented a diverse collection of algorithm control systems that make use of off-line and on-line knowledge. While strong performance has been observed, it often comes with a price. Feature engineering appears to be a significant part of many control systems that use both off-line and on-line knowledge. Even though automated methods are available to evaluate the relevance of features, many of the methods require a detailed problem analysis to produce possible features in the first place. However, there are a few exceptions that do not require significant feature engineering for new problem domains. General features such as algorithm performance appear to be far more reusable than problem specific features. We return to the issue of feature engineering effort in Chapter 4.

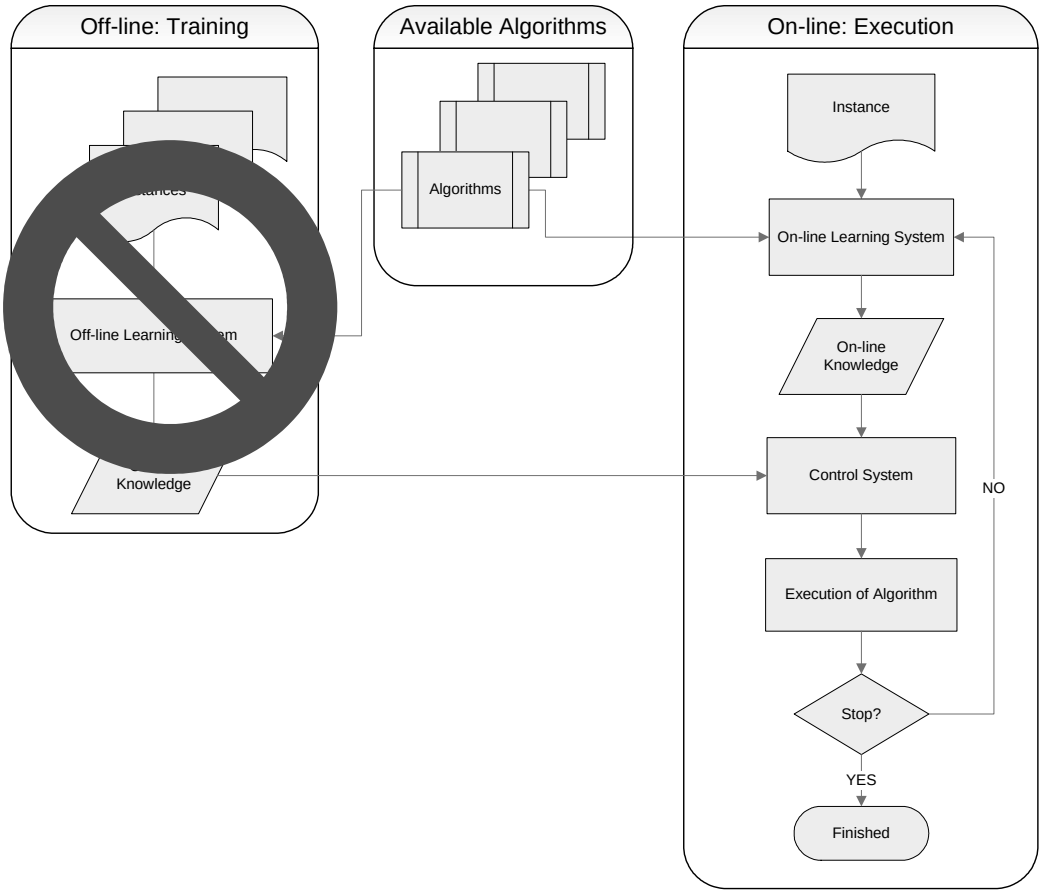


Figure 3.4: On-line knowledge control system.

3.6 On-line Knowledge Control Systems

In this section we review control systems that use on-line knowledge exclusively. The components instantiated in this context are shown in Figure 3.4. A purely on-line system requires no training phase and can be applied immediately to new problem instances and has the benefit that knowledge is specific to the problem instance being solved. There may be useful knowledge that can be carried over from one problem instance to the next, but a purely on-line system forgets all knowledge after each instance is solved. As we have pointed out in the past sections, it is not trivial to know which knowledge should be applied to a new problem instance.

It could be argued that no system is purely on-line, since some decisions

will be made during an off-line phase, like deciding which components to add to the system. To make our distinction clear, for a system to be considered a purely on-line knowledge system, no training phase using a set of sample problem instances is required. As we will show in the following systems, the aim is to produce a system which is not based on any training, but rather it adapts to solve the current problem instance in a purely on-line fashion.

3.6.1 On-line Reinforcement Learning

An on-line reinforcement learning system learns which underlying algorithms are most effective on a problem as it is being solved. There are typically two components in such a system: low-level heuristics that operate directly on the problem being solved and a high-level control strategy that decides which low-level heuristics to apply. We present three systems below that use reinforcement learning to perform on-line learning of effective heuristics.

Nareyek [78] applies reinforcement learning in a constraint-based local search system that learns effective heuristics for planning problems. Each constraint in the planning problem has a set of heuristics that can be used to modify the current solution. Heuristics are selected using a weighted roulette wheel approach so that heuristics that have a higher weight are more likely to be selected. When a heuristic is selected it attempts to improve the solution. After the heuristic is applied, its weight value is updated with a reward/penalty policy. The best policy discovered is to slowly increase weights, for heuristics that perform well, and severely reduce weights for heuristics that perform poorly. The system was tested on two planning domains, Orc's Quest and Logistics, and the reinforcement learning approach outperformed an optimal off-line assignment of weight values. Experiments are performed that show how different heuristics perform better at different stages in search, hence a learning method that adapts based on the on-line performance is superior to a fixed approach.

Burke et al. [20] also employ reinforcement learning to discover effective heuristics for local search, in this case to solve timetabling problems. The approach is called *hyper-heuristics* since it uses the ideas from metaheuristics

to select underlying heuristics. In this case, they use a tabu-list to prohibit the use of heuristics which have not performed well recently. The control procedure also maintains a rank value for each heuristic that is increased if the heuristic is able to make an improvement to the solution, and decreased otherwise. The system selects the heuristic with the highest rank that is not prohibited by the tabu-list. The authors compare the system to specialized solvers for nurse scheduling and exam scheduling and find that the hyper-heuristic system is able to compete with specialized solvers. When the problem structure is perturbed, the hyper-heuristic system outperforms the specialized solvers indicating that hyper-heuristics has the beneficial property of automatically adapting to problem variations.

Perron [88] uses reinforcement learning to bias the selection of search heuristics in a large neighbourhood search [99] applied to network design problems. Several heuristics with fast restart strategies are proposed, with the observation that each tends to perform well on some problem instances, but none outperforms all others on all instances. Heuristic selection is performed with a weighted roulette wheel approach, as in Nareyek [78]. Weight values are increased by 1 if an improved solution is found and decreased by 1 if no improvement is found after 20 attempts using that heuristic.

These variations of on-line reinforcement learning have proved to be both effective in practice and reusable. None requires the use of complex features. Instead, the sole feature employed is improvement in solution quality. The use of a simple, general feature has a positive effect on reusability, but provides enough knowledge to allow a control system to learn which algorithms are effective for solving the current problem instance. However, since a purely on-line system retains no knowledge between runs, it must expend effort to determine which algorithms are likely to be of use. This puts a practical limit on the number of algorithms that can be used in an on-line system, if too many algorithms are added then the system would expend significant effort determining useful ones.

There is also the issue of exploration versus exploitation. In all of the cases above, an algorithm that performs well is more likely to be selected, while an algorithm that has never been tried is less likely to be selected. This can lead

to the situation where an algorithm that makes a small improvement will be exploited extensively and the system does not explore any other choices. Of course, there is a cost in exploring alternatives since they may well prove to be inferior, but if alternatives are not examined then the system can miss the opportunity to select an algorithm with the best performance.

3.7 Summary

In this chapter, we have presented a review of the literature on algorithm control and classified it according to how knowledge of algorithm behaviour is gathered. We have shown the strengths and limitations of both off-line and on-line knowledge gathering, including how these two types of knowledge can be combined.

While our focus has been on knowledge gathering, it is clear that the methods used to gather knowledge can have a profound impact on the control system. An off-line knowledge control system, for instance, requires a training set that is indicative of problem instances to be solved later by the system. An on-line knowledge control system requires a limited number of algorithms to choose among, but can be used without any training. A system that combines off-line and on-line knowledge may be able to achieve better performance, but will require problem instance features to determine when knowledge is relevant to the problem instance being solved.

Our goal is to create systems that automatically configure themselves to achieve good performance on new problem instances and classes. Therefore, one of our prime criteria is that the systems should reduce the required human expertise to deploy them. From our review, we have seen that many approaches have shifted the required expertise into the development of relevant problem features. In the next chapter, we describe a *low knowledge approach* where the goal is to reduce expertise by using simple, general problem features.

Chapter 4

The Low Knowledge Approach to Algorithm Control

In this chapter we discuss the issue of the expertise required to implement algorithm control methods. In particular, we highlight the connection between the knowledge required by a control method and the human expertise required to capture the knowledge. In this context, we build on the discussion presented in Beck & Freuder [10] where low knowledge algorithm control was introduced. Put simply, a low knowledge approach to algorithm control is a practical way to reduce the expertise required to effectively use optimization tools.

4.1 Introduction

The primary goal of the research in this dissertation is the development of optimization tools that are easier to use. Many optimization methods have been developed, with new ones continuing to appear, but it is often unclear which methods will give the best performance when given a new problem instance or a new set of algorithms.

We assume the following context. A library of optimization algorithms that can solve a problem instance is available along with knowledge of the problem instance and algorithms. The algorithm control problem is to de-

cide how best to select or control this set of algorithms to solve the problem instance. The aim of the system is to reduce the expertise required by automatically determining the best choice of algorithm(s).

Our thesis is that a low knowledge approach to algorithm control is a practical way to reduce the expertise required to effectively use optimization tools. The distinction between low and high knowledge (or knowledge-intensive) approaches focuses on the number, specificity, and computational complexity of the measurements that need to be made of a problem instance. A low knowledge approach has very few, inexpensive metrics, applicable to a wide range of algorithms and problem types.

When a high knowledge approach is applied to a new algorithm or problem type, expertise is required to generate sufficient knowledge in the form of features, and to build a model that indicates how features relate to algorithm performance. This model is then used to determine the best algorithm(s) to use. We argue that high knowledge approaches have failed to achieve the goal of expertise reduction as expertise has been shifted from algorithm building into feature identification and model building. While methods exist to automate the process of model building to some degree [114], the identification of useful problem features remains critical to the performance of high knowledge approaches.

This chapter is organized as follows. In Section 4.2 we discuss the knowledge requirement for a problem solving method to make good decisions. We present two extreme approaches, high and low knowledge, and highlight concerns regarding the practical implementation and effectiveness of these approaches. In Section 4.3 we discuss these ideas in relation to algorithm control and classify examples from the literature as high or low knowledge. We show that low knowledge control methods are better suited to more problems despite being easier to implement.

4.2 The Knowledge Requirement

Simon [100] observes that one of the most challenging parts of solving a problem is identifying a model a computer can reason with. The problem

is then solved using this model as a knowledge representation. We refer to the information required by such a model as the knowledge requirement. In the context of control, the purpose of knowledge is to allow the control system to reason about the consequences of possible control decisions. A typical approach to building such a model is to capture all of the information required to get the best performance.

In this section, we focus our attention on the knowledge requirement when implementing a control system. We are interested in evaluating a control system not just on performance, but on how practical the control system is to implement. Recall that the goal of our research is to reduce the expertise required to use optimization tools. Hence, we are not only concerned with performance, but also the level of expertise required to create and maintain the knowledge required by a control system.

For the purpose of illustration, we use the example of controlling a heating system in a building. The quality of a heating control system is evaluated on how closely it matches the ambient temperature to the desired temperature. We describe a high and low knowledge approach to this control problem and then compare them.

4.2.1 High Knowledge

One extreme approach to the control problem is to create a complete model of the world. For example, describing the molecular structure of materials, the physical laws of heat transfer, weather systems, and so on. Clearly, this is impractical and often impossible. However, it highlights the functional requirement of the model to correctly capture the dynamics of the world.

In more realistic terms, a high knowledge model will contain all of the information, within reason, that is required to measure the impact of decisions. The aim of the model builder is to produce the most accurate model possible. The high knowledge approach to model building captures the relevant problem structure to create a model that serves as a proxy for the structure of the real world. Note that building such a model requires significant expertise, insight and care.

To illustrate this in the example of a heating system, a high knowledge model of the building might contain the inside and outside temperature, wind speed, rainfall, weather forecast, time of the year, when the heating system was last on, conductive properties of the materials in the wall, whether doors and windows are open or closed, the number of people and machines in the building, and so on. All of this information is to be used to give the control system a better understanding of the impact of turning on the heating system.

4.2.2 Low Knowledge

We now consider the other extreme approach in applying a control system, to represent as little information about the problem structure as possible. In this context we can use the world as the model. That is, the model becomes measurements of the real world rather than an abstract representation of it. This side steps the issue of capturing problem structure, as Brooks states [19]

> The key observation is that the world is its own best model. It is always exactly up to date. It always contains every detail there is to be known. The trick is to sense it appropriately and often enough.

In this context there is no high knowledge representation, there is simply the real world. Decisions are made by sampling features from the world. A low knowledge approach reasons by interacting with the world rather than using a proxy model.

In the context of a heating system, the most obvious (and common) choice is the ambient temperature as measured by a thermostat. This feature has an interesting property; it is directly related to the performance of the system.

4.2.3 Comparison

As stated, the primary evaluation criteria of a knowledge representation approach is the ability to make good decisions. However, we wish to also compare these systems on the level of expertise required to implement them and the generality of their use.

In the context of heating systems, the primary criteria is the ability to match the ambient temperature with the desired temperature. It is often presumed that a perfectly implemented high knowledge approach is able to perform better since it has a more complete model of the world and is able to make decisions that will avoid over or under heating the building. The low knowledge system may be slow to react to a drop in temperature or exceed the desired temperature since there may be a delay when the heating system is turned on and off. In contrast, the high knowledge system might be able to anticipate such delays and compensate in its decision making.

However, performance is effected through discrepancies between the model and the real world. The total error is a function of these discrepancies: the error in measuring each feature and the accuracy of the modelled relationship between features. Therefore, a possible consequence of increasing the number of features is that the system becomes more susceptible to errors. While these discrepancy errors can be partially addressed with the use of filtering methods and careful model design, the risk of error may increase when adding more features to create a more complex model of the world.

Now consider the expertise required to implement these systems. It is clear that the low knowledge approach is easier to implement. The requirement is simply the introduction of temperature sensors in the areas to be heated and the design and implementation of a control policy based on these sensors. The high knowledge approach on the other hand, requires all of the detailed information described in the previous section. Some of this information, such as the conductive properties of the walls, may not even be available except in the case of new buildings or unless significant effort is spent in measurement. The knowledge requirement makes the high knowledge system harder to implement and less general, since the configuration needs to be customized for each building.

So, while a high knowledge system can theoretically have better performance than a low knowledge system, we believe that in practice a low knowledge system is likely to be easier to implement, have less error, and may be able to outperform a high knowledge approach.

4.2.4 Related Work

The extensive field of control theory [102] has traditionally addressed control problems such as our heating example. The control models in this field are representations of the world based on measures and differential equations which are solved to determine optimal (with respect to the model rather than reality) control policies. However, the expertise of feature detection is still required to identify the measurements, in addition to the skill required to identifying a suitable system of equations.

The famous 'curse of dimensionality' [13] points to another problem with the use of numerous high knowledge features. Informally, the curse states that as the number of features increases, the model space grows exponentially. This has many consequences ranging from increasing the number of required samples to train a system to the computational complexity of modelling the possible states of the problem. Since adding relevant features not only requires expertise, but also increases the computational complexity of the model, this observation suggests a low knowledge approach with few features will avoid these issues.

4.3 Knowledge in Algorithm Control

In this section, we review examples of algorithm control from the previous chapter and analyze the type of knowledge that is used in each approach. The aim here is to evaluate existing work not only in terms of performance, but in terms of high or low knowledge requirements.

4.3.1 Knowledge Engineering Effort

Learning high knowledge problem features that are indicative of algorithm performance requires a significant amount of expertise and/or trial and error. In practice, this is typically the expertise of a practitioner who knows from experience which problem features are likely to impact algorithm performance.

Once a collection of features has been identified, a high knowledge model can be built to predict algorithm performance. While there are methods to help in model building, it is not entirely automatic. The Weka toolkit [114] provides a suite of machine learning tools, but even still, determining the correct subset of features and type of learning model is not trivial. Often, the results of the model building exercise will be that the model does not capture enough knowledge and therefore gives poor performance. Further effort is then required to understand why and to derive new features until a satisfactory model has been developed.

Significantly less effort is required for low knowledge algorithm control. For the case of increasing performance, it can be simply a measure of algorithm performance. The 'model' that produces this measure is the result of running the algorithm. The question, however, is can this be done in a way that remains computationally efficient.

4.3.2 High Knowledge Approaches

Here we present examples of algorithm control approaches that take a high knowledge approach. While these approaches achieve improved performance, we argue that the applicability is limited since they require significant expertise to adapt to new problem domains and algorithms.

The work on building empirical performance models to predict the best performing algorithm [68, 81, 83, 117] is fundamentally a high knowledge approach. The performance models take as input a set of high knowledge features and predict either run times or performance ranking. While the methodology is general and the authors suggest methods to reduce expertise, performance models rely heavily on the availability of problem specific features. The authors admit the performance of such a system is highly dependent on the features chosen. That said, Xu et al. [117] claim they have identified a general set of features for SAT problems and their system can be applied 'out of the box' on new classes of SAT problem instances. While their system requires a significant amount of computation time to train on a new set of SAT problems, it can be done without human intervention therefore

justifying the significant upfront investment of expertise. It remains unclear if similar sets of general features exist for other problem domains.

In a similar vein, the work on case-based reasoning [21, 42] to select a search algorithm is also highly dependent on the problem domain. For each problem type, a new set of features must be presented. Gebruers et al. [42] discuss the use of generic CSP features but unfortunately they do not appear to use them in their experiments. These approaches are therefore tailored to specific types of problem. While mechanisms for determining useful features are suggested, the user is still required to determine the problem features for new problem types.

Control methods using Markov Decision Processes (MDP) have been applied to sorting [61] and SAT [60] algorithms. The work on sorting uses a single feature (the size of the list to be sorted) which, while simple, requires the expert insight that size is a dominating factor in the performance of sorting algorithms. The work on SAT explored the use of many high knowledge features of SAT problems, yet the authors were unable to find a high knowledge approach that outperformed the best SAT heuristic. More problematic is the inability of MDP models to handle the state space explosion caused when more features are added to the model.

4.3.3 Low Knowledge Approaches

We now discuss methods that employ a low knowledge approach to algorithm control. With the exception of Beck & Freuder [10], the authors below present their work in the context of improving solving performance and robustness. Yet all of these methods meet the criteria of a low knowledge approach and therefore require less expertise to implement.

In Beck & Freuder [10], the low knowledge approach is explicitly defined and used in the context of classifying a problem instance to determine the best performing algorithm. Unlike the classification approaches using empirical models or case-based reasoning, the sole feature employed is the result of running each algorithm on a problem instance for a short time. Many simple rules to determine the best performing algorithm are presented and

evaluated.

The work on algorithm portfolios [50, 91] performs an analysis of algorithm performance on a training set of problem instances. From this performance analysis, a portfolio is built to improve performance, either quality of solution or runtime. Once a portfolio is built, it is executed on new problem instances according to the amount of time allocated to each algorithm. The method presumes no knowledge at all of the underlying algorithms or problem instances. The only measure required is one of algorithm performance.

The algorithm tuning work of Hutter et al. [53] is a low knowledge approach. The system is given as input a set of parameters to search and a set of problem instances to train on. Different combinations of parameters are explored by running them in order to find the combination with the best performance. The authors claim it has been very successful in discovering good parameter configurations in new domains without requiring any modifications of their control method. Once again, the only metric the algorithm is aware of is algorithm performance, which is be defined in a user specified way.

The work applying reinforcement learning to search heuristics [20, 78, 88, 103] is interesting as it measures incremental improvements in solution quality to determine effective heuristics during search. These methods can be used on any system which has a set of heuristics. As the methods are applied, a weight value for the method is increased or decreased depending on the performance it achieves. The only metric the control algorithm is aware of is the cost improvement in applying a method. These approaches have the benefit of avoiding the use of a training set, which can be problematic if unseen problem instances behave differently than those in the training set.

4.3.4 Summary

We have reviewed selected algorithm control approaches with a focus on the knowledge requirement for each method. While high knowledge approaches are successful once a suitable model has been found, they require a significant amount of expertise to apply to a new problem domain. In contrast, the low

knowledge methods presented have very few features that are independent of the problem instances they are solving.

From the point of view of reducing expertise, it is clear that, on a new problem, a low knowledge approach is easier to apply. What is unclear is the benefit of a high knowledge versus a low knowledge approach. In Chapter 5 we perform an explicit comparison of high versus low knowledge on a challenging set of scheduling problems. In Chapter 6, we demonstrate the suitability of the approach in a different control setting by applying it to neighborhood selection in large neighborhood search.

4.4 Conclusion

Building on the work of Beck & Freuder [10], we have presented the distinction between high and low knowledge algorithm control and made the case that this is a necessary direction in order to reduce the expertise required to effectively use optimization tools. In the following chapters, we evaluate the performance of these approaches.

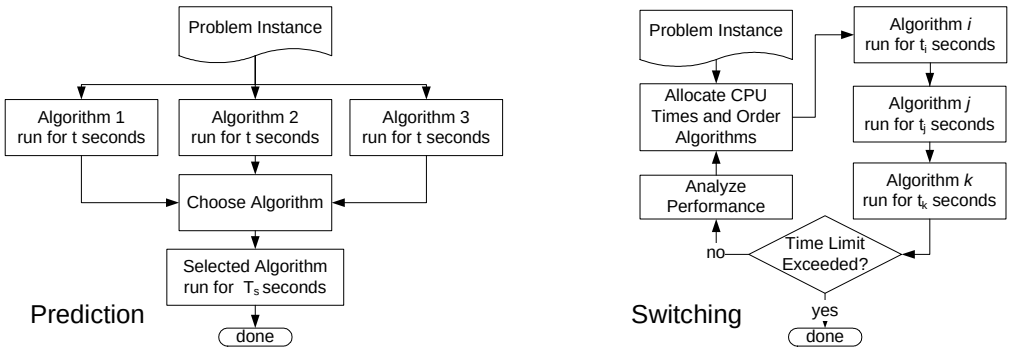
Chapter 5

Low Knowledge Control Applied to Scheduling Algorithms

In the previous chapter, we introduced the idea of low knowledge algorithm control and claimed it satisfies the requirement of reducing expertise. In this chapter, we evaluate the performance of algorithm control methods which make decisions based only on low knowledge features. We compare this against perfect high knowledge selection and show that a low knowledge switching approach is able to perform just as well. This is a significant contribution as it shows that increased performance can be achieved without the expertise required to develop a complex high knowledge prediction model.

5.1 Introduction

In this chapter we investigate algorithm control techniques aimed at achieving strong scheduling performance using off-the-shelf algorithms. The control techniques explored do not require significant human expertise to apply, and therefore increase the accessibility of these algorithms to non-expert users. To achieve this goal, we use machine learning methods to implement low knowledge algorithm control techniques. Rather than building knowledge-



The two control paradigms presented in this chapter. *prediction* makes a single decision based on performance observations over short run-times of a set of algorithms, while *switching* allocates computational resources to each algorithm over a series of iterations.

Figure 5.1: Predictive and switching algorithm control paradigms.

intensive models relating algorithm performance to problem features, we base control decisions on changes in solution quality over time.

Given a time limit T to find the best solution possible to a problem instance, we investigate two control paradigms, shown in Figure 5.1. The first, *predictive*, paradigm runs a set of algorithms during a prediction phase and chooses one to run for the remainder of T. Based on these short runs, we decide which algorithm to continue running for the remaining time. We compare a simple rule against a Bayesian classifier which attempts to correlate algorithm performance trends from the prediction phase with the best performing algorithm for that problem instance. Once it is trained, the Bayesian classifier is used to predict the algorithm that is expected to be the best performer on new problem instances based only on performance in the prediction phase.

In the second, *switching*, paradigm control decisions allocate computational resources to each algorithm over a series of iterations such that the total run-time is T. In an iteration, we run each algorithm, one after another, passing the best known solution from one algorithm to the next. Thus, rather than make a single decision about which algorithm to select for the remaining run-time, we revisit our choice over and over. We apply a reinforcement-

learning approach to allocate more run-time to algorithms that perform well.

The contribution of this chapter is the demonstration that a low knowledge approach to algorithm control can achieve performance significantly better than the best pure algorithm and as good as a perfect high knowledge selection approach.

5.1.1 Scenario

We address the following computational problem. A problem instance is presented to a scheduling system and that system has a fixed CPU time of T seconds to return a solution. We assume that the system designer has been given a set of pure algorithms, A, that are applicable to the given problem and a set of problem instances (the *learning* set) at implementation time. The problem instances in the learning set are representative of the problems that will be later presented. The task is to determine how to apply the algorithms in A on each instance to achieve the best quality of solution.

We make some basic assumptions about the pure algorithms for which our techniques are appropriate. First, after a brief start-up time (at most, a few seconds) an algorithm is always able to return the best, complete solution it has found so far. Second, we require that an algorithm is able to take an external solution and search for solutions that are better. If an algorithm has not found a better solution, when asked for its best solution, the algorithm returns the external solution. We believe these assumptions to be very general; many algorithms exhibit an anytime nature and are able to improve on an existing solution.

5.1.2 Outline

The balance of this chapter is organized as follows. Section 5.2 introduces the pure algorithms that are used in the experiments in this chapter. Section 5.3 describes the problem instances and evaluation criteria used to evaluate algorithm performance. In Section 5.4, pure algorithms are run independently and the need for algorithm control is shown. Section 5.5 presents a study

on predicting algorithm performance and a low knowledge classifiers is compared against a perfect high knowledge classifier. Section 5.6 introduces a low knowledge switching strategy with performance equivalent to the perfect high knowledge classifier. We follow with conclusions.

5.2 Pure Algorithms

Three pure algorithms are taken from the scheduling literature and implemented in C++ using ILOG Scheduler 5.3 [98], a constraint programming library. The algorithms described in this section were chosen from a set of eight algorithms because they have generally comparable behavior on the learning set. The other techniques performed much worse (sometimes by an order of magnitude) on every problem.

We assume that these pure algorithms are able to find a sequence of increasingly good solutions. As we are minimizing a cost function, an initial solution (with a very high cost) is always easily available. Each algorithm successively finds better solutions as it progresses either through local search or branch-and-bound, terminating when it has either proved optimality or exceeded a time limit. At any given time, each algorithm can return the best complete solution that it has found so far.

5.2.1 SetTimes

The *settimes* algorithm configuration is an example of a constructive search technique. It uses the SetTimes heuristic [64, 98], propagation (precedence graph, disjunctive [63] and edge-finding [7]), and slice-based search [11], a type of discrepancy-based search.

5.2.1.1 SetTimes Heuristic

The SetTimes heuristic [64], as implemented in ILOG Scheduler, sets the start times of activities using the following logic:

1. Let S be the set of selectable activities. Initialize S to the complete set of activities to be scheduled.

2. If all the activities have a fixed start time then exit: a solution has been found. Otherwise let S contain all activities that are 'selectable' and do not have a fixed start time. See step 4b for the definition of selectable.

3. If the set S is not empty: (a) Select an activity from S which has the minimal earliest start time. Use the minimal latest end time to break ties. (b) Create a choice point (to allow backtracking) and fix the start time of the selected activity to its earliest start time. Goto to step 2.

4. If the set S is empty: (a) Backtrack to the most recent choice point. (b) Upon backtracking, mark the activity that was scheduled at the considered choice point as 'not selectable' as long as its earliest start time has not changed. Goto step 2.

After each decision in step 3b, the earliest start times and latest end times of activities are updated by constraint propagation. The status 'not selectable' in step 4b is also maintained by constraint propagation.

5.2.1.2 Slice-Based Search

Slice-Based Search [11, 54] performs depth first search on all nodes with paths from the root that have up to some bounded number of discrepancies. Given a bound of k, the ith iteration visits all leaf nodes with discrepancies between $(i-1)k$ and $ik-1$ inclusive. The bound, k, is referred to as the width of the search.

A discrepancy is a deviation from the way the search tree is explored. For example, with the SetTimes heuristic, the current search node may indicate the selection and scheduling of $A1$ and a discrepancy is to not schedule $A1$ at that node, but rather schedule another activity.

5.2.2 Texture

The second constructive search strategy we describe, *texture*, uses precedence constraint posting and texture measurements [8, 9]. Search proceeds by sequencing pairs of activities that compete for the same resource. Texture measurements are used to identify activity pairs that are critical, implying that these pairs will be the hardest ones to schedule and therefore they should be sequenced first. The same constraint propagation algorithms described in Section 5.2.1 are used here. Bounded backtracking and randomization are added to diversify search.

5.2.2.1 Texture Heuristic

The texture heuristic [8, 9] operates by interleaving texture measurements with search. As search proceeds, texture measurements are updated based on the current search state. The search heuristic makes choices based on the current texture measurements.

A texture measurement is a measure of the current search state regarding how likely a constraint is to be violated, called the probability of breakage. In the context of scheduling, this is is applied to resource requirement constraints. Specifically, a texture measurement is an estimation of the probability that an activity will require resource r at time t. Each such measurement produces an individual demand curve which can be combined to form an aggregate demand for a particular resource. The texture measurement we use is included in ILOG Scheduler [98] and is an implementation of the SumHeight measurement. The SumHeight measurement is a sum of the individual activity demand curves on a resource.

Given a texture measurement, the search procedure operates as follows:

1. Compute texture measurements.
2. Find the critical resource R and time point t with the maximum texture measurement.
3. Select two activities, A and B, that rely the most on resource R at time t. This can be determined from the individual texture measurement

of each activity. Pairs of activities that already have a precedence constraint between them are excluded from selection.

4. Add a choice point $A \rightarrow B$ or $B \rightarrow A$. The ordering that preserves the most local slack between the activities is applied first.

5. Repeat until all activities are completely sequenced on each resource.

5.2.2.2 Randomization and Bounded Backtracking

In order to diversify search, randomization and bounded backtracking are employed. Bounded backtracking ensures that search does not get 'stuck' in a fruitless area of the search space. Randomization is required so that when search restarts, it does not visit the same area of the search space.

The bound on backtracks follows the pattern of

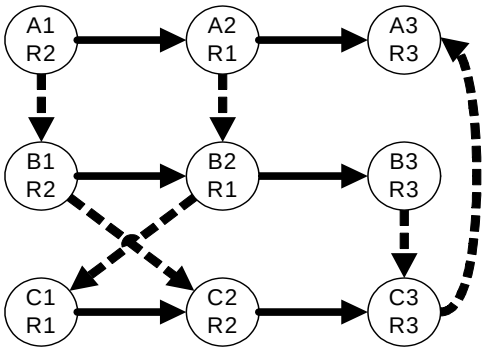
$$BT = \{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1, 1, 2, 4, 8, 16, \ldots\}$$

Formally, $BT_i = \lceil 2^{k-2} \rceil$ where $k = n - (m(m+1)/2) + 1$, $m = \lfloor (\sqrt{8n+1} - 1)/2 \rfloor$ and $n = i + 2$. This was inspired by the optimal, zero-knowledge pattern of Luby et al. [69] but is more aggressive in how it increases the bound [116].

The randomization element is introduced in the selection of the critical resource and time point. Instead of taking the resource and time point with the maximal value, any such point within 10% of the maximal is considered by random selection.

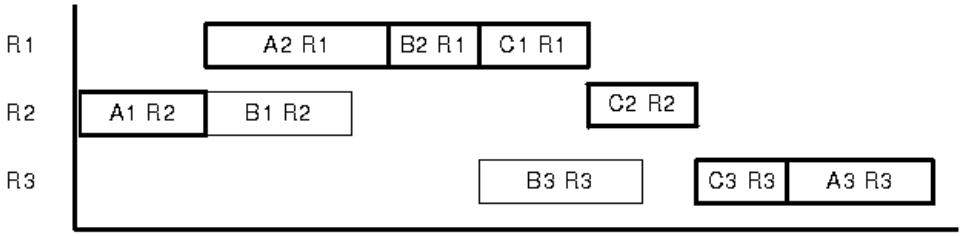
5.2.3 Tabu-TSAB

The algorithm configuration we refer to as *tabu-tsab* is a sophisticated local search procedure for the job shop scheduling problem inspired by the work of Nowicki & Smutnicki [79]. The neighborhood is based on swapping pairs of adjacent activities on a subset of a critical path. An important aspect of *tabu-tsab* is the use of an evolving set of the five best solutions found. Search returns to one of these solutions and moves in a different direction after a fixed number (1000 in our experiments) of iterations without improvement.



Each node represents an activity. Solid arrows represent a job precedence while dashed arrows represent an ordering on a resource.

Figure 5.2: Graph representing a solution to a JSP.



Activities on the critical path are shown with thick borders.

Figure 5.3: Gantt chart of the example JSP showing the critical path.

5.2.3.1 N5 Neighborhood

The *tabu-tsab* algorithm represents a solution using a precedence graph that contains a total order for each resource. For a given solution, the makespan is determined by the longest path in this graph, which is referred to as the critical path. In order to improve the makespan, the length of this path must be reduced.

In Figure 5.2 an example of a 3-job, 3-machine JSP is shown. The solid arrows represent the job-based precedence constraints indicating the order of the activities of each job. The dashed lines represent a current solution, indicating the order of activities on each resource. For example, job A requires resources in the order R2, R1 and then R3. The current solution indicates that resource R2 has the activity ordering A1R2, B1R2, C2R2. In Figure

5.3 a Gantt chart of this solution is shown with the activities on the critical path highlighted using thick borders. In general there can be multiple critical paths, however in this example there is only one.

We now introduce some definitions that will be used in this section. A *move* is defined as a modification of the current solution. In *tabu-tsab* this corresponds to modifying a precedence constraint of the form $v = (x, y)$. A precedence constraint v indicates that x precedes y in the current schedule. A neighborhood is a set of moves that are possible given the current schedule.

The efficiency of the *tabu-tsab* procedure lies in the ability of the neighborhood heuristic to focus on moves that are likely to improve the makespan of the current schedule. On a randomly selected critical path, activities are grouped into blocks on each resource. A block is a group of consecutively sequenced activities on a resource where all activities are on the critical path. The moves considered are swapping of the first or last pair of activities in each block, with two exceptions. In the first block in the critical path, only the last pair of activities in that block are considered. Similarly, the move for the last block on the critical path only considers swapping the first pair of activities.

In Figure 5.3 the critical path is shown where the activities in blocks are highlighted with thick borders. The critical path contains the following blocks, $B_1 = (A1R2)$, $B_2 = (A2R1, B2R1, C1R1)$, $B_3 = (C2R2)$, $B_4 = (C3R3, A3R3)$. Blocks B_1 and $B3$ do not contain valid moves since they contain only single activities. The valid moves in this solution are swapping the ordering of:

- $A2R1$ and $B2R1$ from block B_2
- $B2R1$ and $C1R1$ from block B_2
- $C3R3$ and $A3R3$ from block B_4

5.2.3.2 Search Procedure

The *tabu-tsab* search operates using the following procedure. Given the neighborhood of moves as defined in Section 5.2.3.1, moves are classified as

unforbidden (U), forbidden but profitable (FP), and forbidden but nonprofitable (FN). Forbidden moves are moves that are currently stored in a tabu list. Unforbidden moves are simply all the moves in the neighborhood except the ones in the tabu list. The general idea is that search will take the best move from set $U \cup FP$. In other words, search will evaluate each move and select the one that produces the best makespan. Moves in the tabu list can be taken, but only if they deliver the best makespan found so far. Moves not in the tabu list (U) may increase the makespan, in which case the move with the smallest increase is taken.

The tabu list is maintained as a first-in-first-out queue of moves of the form $v = (x, y)$. When the search procedure accepts a move, the inverse move is added to the tabu list. As an example consider swapping the order of activities $B2R1$ and $C1R1$ from the example in Figure 5.3. This produces the move $v = (C1R1, B2R1)$ indicating that $C1R1$ precedes $B2R1$. The inverse move $\bar{v} = (B2R1, C1R1)$ is then added as the newest item in the tabu list. If the tabu list is full, then the oldest move is removed from the list. In the case that no improving moves are found and all moves are in the tabu list, the oldest item in the tabu list will be removed until a move becomes possible.

5.2.3.3 Backtracking in Local Search

Despite the use of a tabu list, there is still a chance that search will become stuck in an area of the search space where it is unable to find an improved solution. A common solution to this problem is to simply restart search from the beginning using a new random seed. While this approach is general, it wastes a significant amount of effort as search restarts from a poor quality solution. *tabu-tsab* uses a less extreme strategy called Back Jump Tracking where search is restarted from a previous state during search.

The rationale is that it is more efficient to start exploring part of the search space from a point which contained a good quality solution. So, when *tabu-tsab* finds an improved solution, it stores the solution, the tabu list, and the next move that was taken. When search restarts, the move is added to

the tabu list. This ensures that when a restart occurs at this search state, a different search trajectory will be taken which will hopefully lead to a different part of the search space.

5.2.3.4 Optimality Condition

There exist two cases where search can terminate because it has found the optimal solution. The first case is that every activity on the critical path is on the same machine, referred to as the resource bound. The second case is that the critical path contains only the activities of a single job, referred to as the job-based bound. The optimality condition is an interesting result since local search methods often do not have the ability to prove optimality. Empirical results in Nowicki & Smutnicki [79] report that *tabu-tsab* was often able to find solutions where this criteria was met.

5.3 Experimental Details

In this section we describe the problem instances used in the experiments and the criteria used to evaluate algorithm performance.

5.3.1 Problem Instances

The job shop scheduling problem (JSP) involves scheduling n jobs across m machines. Each job j consists of m ordered activities, such that each activity is scheduled one after another. For each job, every activity requires a unique machine for a specified duration, and the sequence of these requirements differs among jobs. We look at the objective of minimizing makespan: the total time required from the start of the earliest scheduled activity to the finish of the latest scheduled activity. This problem is known to be NP-hard [41]. For further details see Section 2.2.

Three sets of 20×20 job shop scheduling problems are used. A total of 100 problem instances in each set were generated and 60 problems per set were arbitrarily identified as the learning set. The rest were placed in the test set.

The difference among the three problem sets is the way in which the activity durations are generated.

- In the **Rand** set, durations are drawn randomly with uniform probability from the interval [1, 99].
- The **MC** set has activity durations drawn randomly from a normal distribution. The mean and standard deviation are the same for the activities on the same machine but different on different machines. The durations are, therefore, machine-correlated (MC).
- In the **JC** set the durations are also drawn randomly from a normal distribution. The means and standard deviations are the same for activities in the same job but independent across jobs. Analogously to the MC set, these problems are job-correlated (JC).

These problem structures have been studied for flow-shop scheduling [111]. They were chosen based on the intuition that the different structures may differentially favor one pure algorithm and therefore the algorithms would exhibit different relative performance on the different sets.

5.3.2 Software and Hardware

All algorithms were implemented in C++ using the ILOG Scheduler 5.3 library. Experiments were executed on a Pentium IV 1.8 Ghz CPU with 512MB of RAM running the Linux operating system.

5.3.3 Evaluation Criteria

Our primary evaluation criteria is mean relative error (MRE), a measure of the mean extent to which an algorithm finds solutions worse than the best known solutions. MRE is defined as follows:

$$MRE(a, K, R) = \frac{1}{|R|}\frac{1}{|K|}\sum_{r\in R, k\in K}\frac{c(a,k,r) - c^*(k)}{c^*(k)} \tag{5.1}$$

where R is a set of independent runs with different random seeds, K is a set of problem instances, $c(a, k, r)$ is the lowest cost solution found by algorithm a on problem instance k during run r, and $c^*(k)$ is the lowest cost solution found during our experiments for problem instance k.

For the learning set, $c^*(k)$ is the best performance of the pure algorithms. For the test set, $c^*(k)$ is the best performance of the pure algorithms and the switching techniques, which outperform the pure algorithms in some cases. Due to the stochastic nature of the algorithms, we run each algorithm ten times ($|R| = 10$) on every problem instance.

We also report the mean fraction of problems in each set for which the algorithm found the best known solution, known as mean fraction best (MFB).[7] MFB can show that an algorithm has good performance on some problem instances even if it has poor performance on other instances, something that is obscured with MRE.

MFB is defined as:

$$MFB(a, K, R) = \frac{1}{|R|} \sum_{r \in R} \frac{|best(a, K, r)|}{|K|} \tag{5.2}$$

where $best(a, K, r)$ is the set of solutions for $k \in K$ during run r when $c^*(k) = c(a, k, r)$.

5.4 Pure Algorithm Performance

In this section we evaluate the performance of the pure algorithms. We will see from the results that there is a case for algorithm control on this problem domain. The pure algorithms were run for $T = 1200$ CPU seconds on the learning set of problem instances. Table 5.1 displays the mean fraction of problems in each subset and overall for which each algorithm found the best solution (MFB) and Table 5.2 shows the mean relative error (MRE). The lowest MRE value for each table is shown in bold font. Across all problems,

[7] MFB reports the mean fraction best of 10 runs on the same instance. In some cases the best solution will not be found in all runs. This results in lower MFB scores than if we had experimented with a single run of each algorithm.

	MC	Rand	JC	All
settimes	0.03333	0	0.38333	0.13889
tabu-tsab	0.39833	0.19333	0.2	0.26389
texture	0.03833	0.06	0.87333	0.32389

Table 5.1: Mean fraction of problems in each learning problem set for which the best solution was found by each pure algorithm.

	MC	Rand	JC	All
settimes	0.04243▲	0.04210▲	0.01152	0.03202▲
tabu-tsab	**0.00740**△	**0.01425**	0.01308	**0.01158**
texture	0.02027	0.01608	**0.00197**△	0.01277

△ indicates significantly *better* performance than all other pure techniques
▲ indicates significantly *worse* performance than all other pure techniques

Table 5.2: Mean relative error of solutions found by each pure algorithm on the learning set.

the difference in MRE performance between *texture* and *tabu-tsab* is not statistically significant.[8] However, there are significant differences among the problems sets: while *tabu-tsab* dominates in the machine-correlated (MC) problem set, the results are more uniform for the random (Rand) problem set and *texture* is superior in the job-correlated (JC) set.

These results demonstrate the opportunity to solve an algorithm selection problem as no one pure algorithm dominates. In practice, a common approach is to simply choose the pure algorithm that is best on average on the learning set (e.g., the algorithm with the lowest MRE) and run that algorithm for all subsequent problem instances. Clearly, with no dominant algorithm, such an off-line algorithm selection approach will not result in the best performance possible on each problem instance. It is unclear, however, that an on-line approach (either high knowledge or low knowledge) will be better as some on-line computational time must be spent in selecting the

[8]All statistical results in this chapter are measured using a randomized paired t-test [27] with a significance level of $p \leq 0.005$. To readers concerned with the issue of Type I errors due to multiple comparisons, a Bonferonni adjustment implies that our results are significant to $p \leq 0.05$, with at most 10 comparisons.

algorithm.

5.5 Prediction

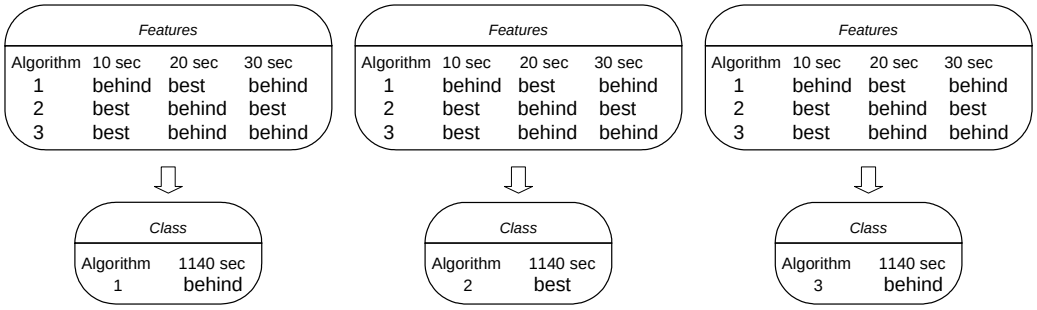
On-line algorithm selection chooses an algorithm to run only after the problem instance has been presented to the system. In this case, the time to make the selection must be taken into account. To quantify these times, let t_p represent the prediction time and t_r represent the subsequent time allocated to run the chosen pure technique. It is required that the total run-time $T = t_p + t_r$.

In the prediction techniques investigated here, during the prediction phase each pure algorithm, $a \in A$, is run for a fixed number of CPU seconds, t, on the problem instance. We require that $t_p = |A| \times t$. The quality of solutions found for each run is used to select the algorithm that will achieve the best performance given the time remaining. This is the same low knowledge prediction framework that was introduced in Beck & Freuder [10].

We assume that when a pure algorithm has been selected it does not have to re-start: it can continue the search from where it left off in the prediction phase. The total run-time for the selected algorithm is $T_s = t_r + t$. The learning set is used to identify t^*, which is the value of t that leads to the best prediction performance.

5.5.1 Control Rule

Algorithm selection can be performed based on the application of a simple control rule. In Beck & Freuder [10], several control rules are investigated that make a selection based on observations of the quality of solutions found. Of these methods, choosing the method that found the best solution so far dominated other approaches. We define this approach as $pcost_{min}$. Let $c_p(a, k)$ be the solution quality found by algorithm a during the prediction phase on problem instance k. The algorithm selected by $pcost_{min}$ is the one which has the minimum value of $c_p(a, k)$. Ties are broken randomly.



Three Bayesian classifiers are used together to predict the best algorithm at time $T_s = 1140$. In this example the prediction time is $t_p = 90$ seconds which means each algorithm is run for 30 seconds during the prediction phase. The performance trends of these 30 second runs are input as features to each classifier. Each classifier then predicts the final performance for a specific algorithm.

Figure 5.4: Bayesian classifiers.

5.5.2 Bayesian Classifier

In an off-line training phase, a Bayesian classifier is constructed for each pure algorithm given a predefined prediction time, t_p, for the purpose of predicting the final performance of that algorithm at time T_s. In each classifier, for every 10 seconds of prediction time, a feature variable is created with possible values of "best" or "behind" for each pure algorithm. The value is assigned depending on whether the algorithm has found the best solution out of all the algorithms, $a \in A$, at that time point. More than one algorithm may have found the best solution at a particular time. Each classifier has a class variable representing the final performance of the algorithm (i.e, at time T_s). It also has the value "best" or "behind." For the learning phase, feature and class values are input. For the test phase, only feature values are given and the classifier predicts the final performance of each algorithm.

For example, let $T = 1200$, $t_p = 90$, and $|A| = 3$, as shown in Figure 5.4. In this example we input nine feature variables: three for each algorithm at 10, 20 and 30 seconds and the classifiers predict the final performance for each algorithm at $T_s = 1140$ seconds. The classifiers are trained using the algorithm performance on the learning set. Once trained, the hybrid algorithm executes each pure algorithm for $t_p/|A| = 30$ seconds to gather

the input features. The classifiers then predict which of the algorithms will be best at 1140 seconds, and the selected algorithm is run for the remaining 1110 seconds, continuing search from where it left off.

To determine the best value of t_p, the procedure is repeated by learning a new set of Bayesian classifiers for values of $t_p \in \{30, 60, 90, \ldots 1200\}$ providing an assessment of hybrid algorithm performance for each prediction time. The prediction time with the best performance is t^*. The Bayesian classifiers for $t_p = t^*$ are then used on the test set.

In the case that more than one algorithm is predicted to be best, ties are broken by choosing the algorithm with the best mean solution quality on the learning set at time T. Therefore the tie breaking order is (from highest to lowest preference): *tabu-tsab*, *texture*, and then *settimes*. The Bayesian classifiers are learned using WinMine 2.0 [26] with default parameters. We refer to this techniques as *Bayes*.

5.5.3 Perfect Knowledge Classifiers

We can perform an *a posteriori* analysis of our data to compare our low knowledge approach with the best performance that can be achieved with a high knowledge classifier which seeks to select a pure algorithm for a given problem instance. The *best-all* classifier chooses the pure algorithm that is best, on average, over all problem instances. This algorithm is then applied to every problem instance. The *best-sub* classifier chooses the pure algorithm that is best, on average, for the problem set to which the test instance belongs. *best-sub* is the best result we can achieve with a high knowledge classifier that infallibly and with zero CPU time can categorize instances into the correct problem set. The *best-instance* classifier makes a stronger assumption: that, with zero CPU time, the high knowledge classifier can always choose the best pure algorithm for a given *instance*. No technique based on choosing a pure algorithm for a given instance can perform better.

5.5.4 Experiments: Prediction Control

For these experiments, once again the overall time limit, T, is 1200 CPU seconds. Each pure algorithm is run for T seconds with results being logged whenever a better solution is found. This design lets us process the results to examine the effect of different settings for the prediction time, t_p, and different values for $T \leq 1200$. As noted, the number of algorithms, $|A|$, is 3.

Recall that the learning set is used to build a set of Bayesian classifiers for each possible prediction time and identify the one with the smallest mean relative error (MRE). The best t_p for the Bayesian classifiers was found to be 270 seconds, meaning that each pure algorithm is run for 90 seconds and then the Bayesian classifiers are used to select the pure algorithm to be run for the subsequent 930 seconds. The same process was used to determine the best prediction time for $pcost_{min}$. The best t_p value for $pcost_{min}$ was 120 seconds, where each pure algorithm is run for 40 seconds.

Again we use the measure of MRE to compare algorithm performance, this time on a test set of new problem instances. Recall that MRE is calculated by comparing how well an algorithm did on a problem instance k against the best-known solution, $c^*(k)$. The best-known solution is not necessarily the optimal solution, but rather the best solution found during our experiments. While no prediction technique can perform better than the best pure technique, some of the switching techniques which we will present in Section 5.6.1 do perform better than the best pure technique on the test problem set. This has the effect of increasing the MRE of the pure algorithms on the test set. Similarly, the mean fraction of best solutions found decreases for the pure algorithms. For example, *best-instance* indicates that pure methods now find the best solutions on only 25.8% of all problem instances.

Table 5.3 and Table 5.4 present the results of the experiment and the statistical significance of the prediction and perfect classifier techniques compared with the pure algorithms and with all prediction algorithms on the test set. Only the *best-instance* perfect classifier finds an MRE significantly different (lower) than either prediction technique. In other words, there are no significant differences among the two on-line prediction variations, and

	MC	Rand	JC	All
settimes	0	0	0.32500	0.10833
tabu-tsab	0.00750	0.00500	0.17500	0.06250
texture	0.01750	0.02250	0.72250	0.25420
Bayes	0.0175	0.01500	0.62500	0.21917
pcost_{min}	0.0150	0.00750	0.66750	0.23000
best-all	0.00750	0.00500	0.17500	0.06250
best-sub	0.00750	0.02250	0.72250	0.25000
best-instance	0.02500	0.02750	0.72250	0.25833

Table 5.3: Mean fraction of best solutions found by each prediction algorithm on the test set.

	MC	Rand	JC	All
settimes	0.04602	0.05185	0.01684	0.03823
tabu-tsab	0.01985	0.02115	0.01574	0.01891
texture	0.02919	0.02056	0.00894	0.01956
Bayes	0.01918	0.01863	0.00737△	0.01506△
pcost_{min}	0.02038	0.02110	0.00729△	0.01626
best-all	0.01985	0.02115	0.01574	0.01891
best-sub	0.01985	0.02056	0.00894	0.01645
best-instance	**0.01482□**	**0.01498□**	**0.00491□**	**0.01157□**

△ indicates significantly *better* performance than best-all
□ indicates significantly *better* performance than all prediction methods

Table 5.4: Mean relative error of solutions found by each prediction algorithm on the test set.

the *best-sub* classifier. This demonstrates that even a simple rule can be applied in a low knowledge context to achieve competitive performance. Both prediction methods outperform *best-all* on the JC problem set, and *Bayes* outperforms *best-all* when compared over all problem instances.

To explore the impact of different time limits, we performed a series of experiments using the same design as above with $T \in \{120, 240, \ldots, 1200\}$. For each time limit, the learning set was used to create a set of Bayesian classifiers and to identify the optimal prediction time, t^* for both *Bayes* and $pcost_{min}$ individually (i.e. the same t^* is not used for both prediction methods). Table 5.5 presents the results. The *best-all* column is the MRE of the best pure technique on the test set at each time limit. For $T \in \{120, \ldots, 960\}$, *best-all* is *texture* which achieves the lowest MRE of all pure techniques. For $T \geq 1080$, *best-all* is *tabu-tsab*. There is no evidence of a significant difference between either prediction method and *best-sub* for any time limit with the exception of $T = 120$, when they are both worse and where *Bayes* is also worse than *best-all*. Interestingly, *Bayes* outperforms *best-all* at most time limits after 480 seconds. We see that *best-instance* is significantly better than all prediction techniques at all time limits.

We conclude that the results for $T = 1200$ are applicable to other choices for an overall time limit. The prediction approaches are no worse than a high knowledge classifier that can identify the problem set, *best-sub* except at very low time limits. The $pcost_{min}$ technique does not perform significantly worse than *Bayes*, however *Bayes* outperforms *best-all* at many time limits.

5.5.5 Summary: Prediction Control

Our results demonstrate that a prediction-based low knowledge algorithm selection technique can perform as well as choosing the best pure algorithm (on average) based on a learning set and as well as a reasonable, though idealized, high knowledge algorithm selection technique *best-sub* that is able to infallibly classify problem instances into the correct underlying subset. Performance is not as good as the optimal high knowledge approach *best-instance* that is able to infallibly choose the pure algorithm that will lead to

Time Limit	Algorithm				
	Bayes	pcost_{min}	best-all	best-sub	best-instance
120	0.04065▲	0.04086	0.03867	0.03665□	**0.03129**□
240	0.02980	0.03151	0.03108	0.02911	**0.02424**□
360	0.02576	0.02664	0.02772	0.02586	**0.02053**□
480	0.02318△	0.02378	0.02557	0.02337	**0.01837**□
600	0.02076△	0.02160	0.02413	0.02191	**0.01698**□
720	0.01957	0.02030	0.02246	0.02027	**0.01555**□
840	0.01814△	0.01928	0.02146	0.01900	**0.01441**□
960	0.01694△	0.01815	0.02073	0.01798	**0.01332**□
1080	0.01582△	0.01714	0.01975	0.01709	**0.01222**□
1200	0.01506△	0.01626	0.01891	0.01645	**0.01157**□

△ indicates significantly *better* performance than best-all
□ indicates significantly *better* performance than all prediction methods
▲ indicates significantly *worse* performance than best-all

Table 5.5: Mean relative error of each prediction algorithm for different time limits over all problem sets.

the best solution for each problem instance.

A simple rule choosing the algorithm that returned the best solution in the prediction phase achieves good performance. The more sophisticated prediction technique, based on learning Bayesian classifiers, does not perform significantly better than the simple rule. While these experiments do not allow us to make general conclusions about the use of machine learning techniques (or even Bayesian classifiers) within a predictive control paradigm, they suggest that our simple learning model provides insufficient information for the learning mechanism. It might be the case that further problem features can be identified that can be used to build a more accurate prediction model. However, given that our goal is to minimize the expertise necessary, building more complicated models with additional problem features is not a reasonable direction.

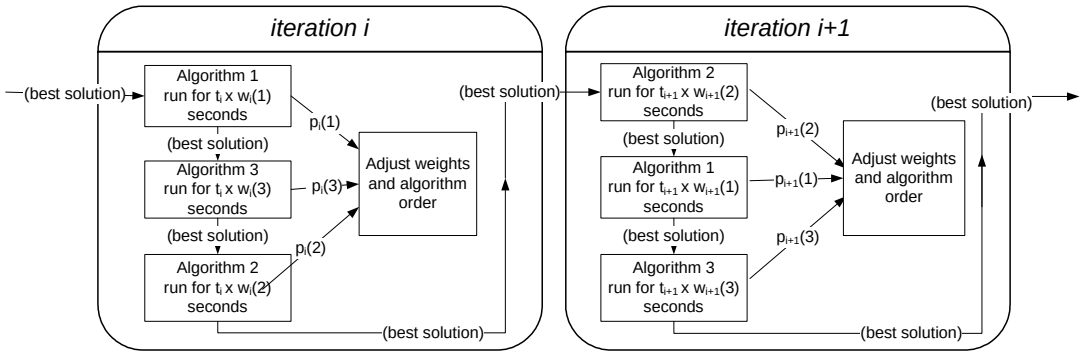
5.6 Continuous Control

Our second approach to algorithm selection generalizes the single decision of the predictive paradigm to a control structure where the selection decision can be made multiple times and, more generally, the decision is not one of selection but of allocation of computational resources to the pure algorithms. There are a number of ways that this paradigm could be instantiated. In this section, we investigate running a series of iterations where the control problem consists of deciding what portion of the iteration time is given to each pure algorithm.

5.6.1 Switching

The switching paradigm allocates run-time to the pure algorithms during search. In contrast to predictive selection, the allocation decision is made multiple times. n iterations are performed such that each iteration, i, has a time limit of t_i CPU seconds and $\sum_{1 \leq i \leq n} t_i = T$. During each iteration, the run-time of each pure algorithm, $a \in A$, is determined using a weight, $w_i(a)$, which is learned as the search progresses. The weights for an iteration are normalized to sum to 1 and therefore, $w_i(a)$ corresponds to the fraction of time allocated to algorithm a. In other words, algorithm a is run for $t_i \times w_i(a)$ seconds. For example, if algorithm a has weight $w_i(a) = 0.2$ and $t_i = 60$ seconds, then a is run for 12 seconds during iteration i. All weights are initialized to $1/|A|$.

Weights are updated after each iteration by considering the current weight and the performance of each algorithm during the last iteration. Performance is measured in cost improvement per second, which allows us to compare algorithms despite the differing run times. We normalize the cost improvement per second to sum to 1 producing a performance value, $p_i(a)$. The weights are then adjusted using a standard reinforcement learning formula: $w_{i+1}(a) = p_i(a) \times \alpha + w_i(a) \times (1 - \alpha)$. The α value controls the influence of the previous weight and the performance value. A diagram of the switching approach is shown in Figure 5.5.



Algorithm switching passes the best solution found from one algorithm to the next. Each algorithm is run for a proportion of the iteration time t_i based on the algorithm's weight value $w_i(a)$. At the end of an iteration, the performance, $p_i(a)$ of each algorithm a is used to adjust the weight values used in the next iteration. Here we show two iterations.

Figure 5.5: Algorithm switching.

Sharing Solutions Unlike the predictive paradigm, switching shares information among algorithms. Each invocation of an algorithm begins with the best solution found so far. However, different pure algorithms are able to exploit this information differently. The constructive algorithms, *texture* and *settimes*, only make use of the bound on the solution quality, searching for a solution that is strictly better than the best solution found so far. Each iteration of *tabu-tsab*, however, must begin with an initial solution. The best solution found so far is used. Recall that at any given time, each algorithm can return the best complete solution that it has found so far. If, in a given time interval, no better solution has been found, the solution found in the previous time interval is returned.

Algorithm Order We switch between algorithms $|A| - 1$ times during an iteration, running each algorithm exactly once. For each iteration, the order in which the algorithms are run is randomly generated with uniform probability. Learning a good order of algorithms is an interesting area for future work.

Learning Rate No weight updating is done until after the second iteration, as it is very easy to find large improvements in solution quality in the first iteration. Therefore learning is postponed until search becomes more challenging. In all variations that use learning, we set the learning rate α to 0.5. Experimenting with different values of α is an interesting direction we intend to pursue in our future work.

Iteration Times Two variations on iteration times are used here. In *rl-static*, the length of each iteration does not vary: $t_i = 60$, for all i. For *rl-double* the time for the first two iterations is fixed until learning has started, $t_1, t_2 = 60$, and subsequently $t_{i+1} = 2 \times t_i$. The motivation for increasing iteration times comes from the observation that the algorithms take more time to find better solutions since the problems become harder as we minimize the cost. Experimenting with different values of of t_i and different growth rates is another interesting direction we intend to pursue in future work.

Learning Set One distinct advantage of the switching approach is that it does not rely on a learning set. All learning is done on-line, after the problem instance has been presented. The system does not require any off-line configuration and so the switching technique is not dependent on the learning set being representative of the real problems that will be encountered. However, a learning set may be useful for tuning the switching parameters such as the learning rate α, the duration of iteration times, and the ordering of algorithms. Another possible use of the learning set is to filter out algorithms that are not competitive at all with a particular problem class.[9]

5.6.2 Experiments: Algorithm Switching

Table 5.6 and Table 5.7 display the results on the three problem sets averaged over ten independent runs, with $T = 1200$. We use the same method to compute MRE as described in Section 5.2. The two reinforcement learning

[9] As mentioned in Section 5.2, we performed this filtering by hand but this could easily be automated.

	MC	Rand	JC	All
settimes	0	0	0.325	0.10833
tabu-tsab	0.0075	0.005	0.175	0.0625
texture	0.0175	0.0225	0.7225	0.2542
Bayes	0.0175	0.015	0.625	0.21917
pcost_{min}	0.015	0.0075	0.6675	0.23
best-all	0.0075	0.005	0.175	0.0625
best-sub	0.0075	0.0225	0.7225	0.25
best-instance	0.025	0.0275	0.7225	0.25833
rl-double	0.0525	0.0575	0.7575	0.28917
rl-static	0.015	0.0075	0.76	0.26083

Table 5.6: Mean fraction of best solutions found by switching techniques on the test set.

	MC	Rand	JC	All
settimes	0.04602	0.05185	0.01684	0.03823
tabu-tsab	0.01985	0.02115	0.01574	0.01891
texture	0.02919	0.02056	0.00894	0.01956
Bayes	0.01918	0.01863	0.00737	0.01506
pcost_{min}	0.02038	0.02110	0.00729	0.01626
best-test	0.01985	0.02115	0.01574	0.01891
best-sub	0.01985	0.02056	0.00894	0.01645
best-instance	0.01482	0.01498	0.00491	0.01157
rl-double	**0.01419**$\circ$	**0.01407**$\triangledown\circ$	**0.00417**$\triangledown\square$	**0.01081**$\triangledown\circ$
rl-static	0.02647$\blacklozenge$	0.02431$\blacktriangledown$	0.00485$\triangle$	0.01854$\blacklozenge$

$\triangle$ indicates significantly *better* performance than best-all
$\triangledown$ indicates significantly *better* performance than best-sub
$\square$ indicates significantly *better* performance than all prediction methods
$\circ$ indicates significantly *better* performance than all other control methods
$\blacktriangledown$ indicates significantly *worse* performance than best-sub
$\blacklozenge$ indicates significantly *worse* performance than best-instance

Table 5.7: Mean relative error performance of switching techniques on the test set.

	Algorithm				
Time Limit	rl-double	rl-static	nl-double	nl-static	no-com
120	0.03555♦△	**0.03401**♦△	0.03637♦	0.03474♦	0.05605 ▲●
240	0.02698♦△	**0.02459**▽○	0.02788♦△	0.02696♦	0.04399 ▲●
360	0.02158▽	**0.02113**▽	0.02324♦△	0.02379♦	0.03867 ▲●
480	**0.01913**▽	0.01971▽	0.02030♦▽	0.02249♦	0.03531 ▲●
600	**0.01680**▽○	0.01898△	0.01882♦▽	0.02198♦	0.03298 ▲●
720	**0.01467**▽○	0.01870♦△	0.01739♦▽	0.02182♦	0.03108 ▲●
840	**0.01339**▽○	0.01860♦△	0.01577▽	0.02174♦	0.02980 ▲●
960	**0.01251**▽○	0.01857♦	0.01462▽	0.02171▼	0.02876 ▲●
1080	**0.01156**▽○	0.01856♦	0.01386▽	0.02168▼	0.02772 ▲●
1200	**0.01081**▽○	0.01854♦	0.01319▽	0.02165▼	0.02682 ▲●

△ indicates significantly *better* performance than best-all
▽ indicates significantly *better* performance than best-sub
□ indicates significantly *better* performance than all prediction methods
○ indicates significantly *better* performance than all other control methods
▲ indicates significantly *worse* performance than best-all
▼ indicates significantly *worse* performance than best-sub
♦ indicates significantly *worse* performance than best-instance
● indicates significantly *worse* performance than all other control methods

Table 5.8: The mean relative error of variations of the switching algorithm at different time limits.

algorithms achieve strong performance. The better one, *rl-double*, outperforms all other control algorithms in every problem set. An advantage of the switching paradigm is that different pure algorithms can be used in different parts of the search, and therefore, better performance can be achieved on a problem instance than the best pure technique. In contrast, predictive selection techniques cannot perform better than the best pure technique. The doubling of the time in the iterations appears to be an important part of the performance of *rl-double*. Nonetheless, *rl-static* is able to achieve performance that is overall not significantly worse than the best pure technique.

As with the prediction techniques, we also varied the overall time limit. Table 5.8 presents the results. The *rl-static* approach achieves significantly lower MRE than *best-sub* for three time limits: $T \in \{240, 360, 480\}$ and performs better than all other control algorithms at $T = 240$. From 240 to

600 seconds, no significant difference is observed when *rl-static* is compared to *best-instance*. Finally, *rl-double* achieves MRE significantly lower than *best-sub* for $T \geq 360$ with no evidence of a significant difference against *best-instance* except when $T \leq 240$. For $T \geq 600$, *rl-double* achieves significantly better performance than all other algorithms shown in Table 5.8. The strong results of *rl-double* are compared to the high knowledge selection methods in more detail in Table 5.9 and discussed at the end of this section.

To assess reinforcement learning, we run *rl-static* and *rl-double* with $\alpha = 0$, which disables reinforcement learning by ignoring the current performance result and keeps the weights constant. Each pure algorithm receives an equal portion of the iteration time, regardless of previous performance. We call these "no-learning" variations, *nl-static* and *nl-double*. Results, also in Table 5.8, demonstrate that reinforcement learning has a strong impact. Although the significant difference is not marked in the table, *rl-static* is significantly better than *nl-static* for all T, and *rl-double* is significantly better than *nl-double* at all time limits except $T = 120$. Interestingly, *nl-double* not only achieves significantly better performance than *best-sub* on seven of the ten time limits but also achieves a significantly lower MRE than all algorithms, except *rl-double*, for $T = 840$ and $T = 960$.

To investigate the impact of communication among the algorithms, the *no-com* technique is *nl-double* without sharing of the best known solution among the algorithms; each algorithm is allowed to continue from where it left off in the previous iteration.[10] *no-com* performs worse than any other technique at all time limits; communication between algorithms is clearly an important factor for performance.

Turning to the comparison of the best switching algorithm to the perfect knowledge classifiers, in Table 5.9 we present a very strong result. The *rl-double* technique achieves a lower MRE than *best-sub* for $T \geq 360$. Furthermore, *rl-double* performs as well as *best-instance* for all time limits $T \geq 360$. In other words, *without complex model building our best low knowledge ap-*

[10] Another way to see this is that *no-com* runs each pure technique independently for $\frac{T}{|A|}$ seconds and returns the best solution found. This is often referred to as a parallel portfolio of algorithms.

	Algorithm			
Time Limit	rl-double	best-all	best-sub	best-instance
120	0.03555♦△	0.03867	0.03665	**0.03129**
240	0.02698♦△	0.03108	0.02911	**0.02424**
360	0.02158▽	0.02772	0.02586	**0.02053**
480	0.01913▽	0.02557	0.02337	**0.01837**
600	**0.01680**▽	0.02413	0.02191	0.01698
720	**0.01467**▽	0.02246	0.02027	0.01555
840	**0.01339**▽	0.02146	0.01900	0.01441
960	**0.01251**▽	0.02073	0.01798	0.01332
1080	**0.01156**▽	0.01975	0.01709	0.01222
1200	**0.01081**▽	0.01891	0.01645	0.01157

△ indicates significantly *better* performance than best-all
▽ indicates significantly *better* performance than best-sub
♦ indicates significantly *worse* performance than best-instance

Table 5.9: Mean relative error of the best switching algorithm against perfect knowledge prediction for different time limits.

proach achieves performance that is as good as the best possible high knowledge predictive classification approach.

5.6.3 Algorithm Switching Summary

The experiments in this section show that a low knowledge switching approach to algorithm control can achieve strong performance. Not only is the strongest variation (*rl-double*) significantly better than the best pure technique (*best-all*) at all time limits and better than *best-sub* after 360 seconds, *rl-double* achieves performance that is equivalent to *best-instance*, the optimal high knowledge selection[11] approach. The main components of *rl-double* are: repeated control decisions during search, communication of the best solution among the pure algorithms, the doubling of the iteration time, and the reinforcement learning. Comparisons with variations that selectively re-

[11]We note that we have not shown that low knowledge switching outperforms high knowledge switching. The analysis of high knowledge switching methods is an interesting area for future work.

move each of these components shows that they all contribute to the overall performance.

The fact that we have created a technique that achieves significantly better performance than the best pure technique, and performance equivalent to an optimal high knowledge selection method, is not the main conclusion to be drawn from these experiments. Rather, the importance of these results stems from the fact that the technique does not depend on expertise, either in terms of development of new pure scheduling algorithms, or in terms of development of a detailed domain and algorithm model. A low knowledge approach is able to achieve better performance without requiring the expertise to develop a high knowledge model.

5.7 Conclusion

In this chapter we have shown that a low knowledge approach to algorithm control can be used to form a system that consistently and significantly outperforms the best pure algorithm. Machine learning techniques play an important role in this performance, however, even a simple-minded approach that evenly distributes increasing run-time among the pure algorithms performs very well.

Our motivation for investigating low knowledge algorithm control was to reduce the expertise necessary to exploit optimization technology. Therefore, the strong performance of our techniques should be evaluated not simply from the perspective of an increment in solution quality, but from the perspective that this increment has been achieved without additional expertise. We neither invented a new pure algorithm nor developed a detailed model of algorithm performance and problem instance features. In the next chapter we will continue to evaluate the performance of a low knowledge approach in a different context to see if similar gains in performance are attainable.

Chapter 6

Low Knowledge Control for Large Neighbourhood Search

This chapter explores the application of low knowledge control methods to the algorithm framework of large neighbourhood search (LNS). The primary focus of this chapter is the use of low knowledge control methods to determine the best performing algorithm components. Experiments are performed on three sets of challenging job shop scheduling problems and strong performance is observed across all problem sets and time limits. We believe these control methods are a significant step towards producing a system that will consistently produce high quality solutions and reduce the expertise required to apply optimization technology.

6.1 Introduction

Large neighbourhood search (LNS) is a combination of constraint programming and local search that has proved to be a very effective tool for solving complex optimization problems. Constraint programming methods are used to provide a general and reusable way to specify and enforce constraints on neighbourhood solutions that are visited in a local search style that efficiently explores the search space of optimization problems. However, the practice of applying LNS to real world problems remains an art which requires a great

deal of expertise: effective neighbourhood heuristics and their parameters must be discovered for each problem class. In this chapter, we show how to reduce the expertise requirement by using low knowledge control methods to create a system that adjusts its behaviour to suit the problem instance being solved.

A great deal of research over the last decade has produced constraint programming algorithms that perform well on many small to medium sized problems. The core strengths of constraint programming are strong inference methods, heuristic search and a natural problem modelling approach. Inference is often able to reduce the search space and quickly detect infeasibility while heuristics are used to guide search toward areas of the search space that are likely to contain solutions. Constraints are specified declaratively in a rich modelling language that is arguably more accessible to non-experts, than say, models using only a linear programming approach. This appears to have all of the elements of a perfect system to allow non-experts to use optimization technology.

The trouble, however, is that standard constraint programming search methods do not scale well. The typical approach to optimization is the use of tree search with branch and bound. These tree search methods are often ineffective at optimizing larger problem instances without significant tuning and effort on the part of an expert in constraint programming. The sheer complexity of many problems renders even the most advanced constraint based search methods ineffective unless careful problem modelling and, often, a decomposition strategy is applied.

In this chapter, we show how to apply constraint programming techniques to large optimization problems while reducing required expertise. We achieve this through a combination of large neighbourhood search and low knowledge algorithm control methods. Our results show strong performance, however, more impressive is the ability of the control methods to adapt to suit the problem instance being solved. Robust solving performance is achieved across all problem sets and time limits indicating that it is possible to reduce the human expertise required to effectively apply optimization tools.

6.1.1 Scenario

We address the following computational problem. A problem instance is presented to a scheduling system and that system has a fixed CPU time of T seconds to return a solution. We assume that the system designer has been given a set of neighbourhood heuristics that are applicable to the given problem and a set of problem instances (the *learning* set) at implementation time and that these instances are representative of the problems that will be later presented. The task is to determine how to apply the neighbourhood heuristics, on each instance, to gain the best quality of solution. This problem solving scenario is similar to the one described in Chapter 5 except in this context we are applying neighbourhood heuristics inside an LNS algorithm rather than choosing between search algorithms.

6.1.2 Outline

This chapter is organized as follows. In Section 6.2 we discuss previous work on LNS, including the application of control mechanisms. In Section 6.3 we describe the large neighbourhood search framework, search methods and neighbourhood heuristics. In Section 6.4, an overview of the algorithm configuration process is described that is expanded in the later sections. In Section 6.5 we describe some details of our experimental setup, including the problem instances, hardware and software. In Section 6.6 we present an algorithm tuning method to configure neighbourhood heuristics. In Section 6.7 we evaluate the performance of individual neighbourhood heuristics and show the need for algorithm control methods. In Section 6.8 we apply combinations of neighbourhood heuristics and different control methods and show robust and strong performance results. In Section 6.9 we evaluate these methods on the Taillard set of benchmarks without tuning and again show strong and robust solving. In Section 6.10 we discuss two control methods for LNS that are very similar to our work. We conclude in Section 6.11.

6.2 Previous Work

The main idea of large neighbourhood search (LNS) is to iteratively apply search to a part of a problem (the large neighbourhood) while restricting the remainder of the problem to the values in the current solution. Many papers have appeared with algorithms that follow this approach. The first work in the constraint programming community using the term large neighbourhood search applied the idea to the problem of vehicle routing with time-windows [99]. In the metaheuristics community, the same concepts have been introduced as a type of variable neighbourhood search called variable neighbourhood decomposition search [47]. Other work in the constraint programming community is now considered an instance of LNS. For example, Nuijten et al. [84] show a random selection neighbourhood in the domain of job shop scheduling. Scheduling algorithms such as the shifting-bottleneck procedure [1] and the shuffling procedure [2] can be considered instances of LNS. The i-Flat algorithm [24, 73] is also directly related. All of these approaches have produced high quality results by applying heuristics to select a part of the problem to focus search on.

While some approaches have adopted a purely random selection process when determining the part of the problem to search, other work has focused on building more involved neighbourhood heuristics for particular problem domains. For example, neighbourhood heuristics have been developed for job shop scheduling [6, 22], earliness/tardiness scheduling [29], network design [25], and car assembly sequencing [89]. The work of Propagation Guided-LNS [90] and Relaxation Induced Neighbourhood Search [29] are interesting as these methods do not rely on domain specific problem structure but instead use properties of the incumbent solution and the constraint network. Although new neighbourhood heuristics often produce a performance gains, the increasing number of heuristics adds complexity to the algorithm configuration process: neighbourhood heuristics need to be selected and, in many cases, parameter values chosen for each neighbourhood.

We now turn to work that has applied algorithm control to LNS. Perron & Shaw [89] apply a low knowledge learning strategy in an LNS framework.

Algorithm 2: Large Neighbourhood Search for Scheduling

```
1: S := Create initial solution
2: while Not optimal and time left do
3:    NH := Select a subset of activities
4:    S' := Remove any information involving NH from S
5:    if Search retaining S' finds improvement then
6:       S := Update solution
7:    end if
8: end while
```

However, this learning strategy is applied only to select a search algorithm used to explore the neighbourhood, rather than the selection of a neighbourhood heuristic. Two recent low knowledge control methods for the selection of neighbourhood heuristics are adaptive large neighbourhood search (ALNS) [92] and self-adaptive large neighbourhood search (SA-LNS) [59]. We discuss these methods in more detail and how they relate to our work in Section 6.10.

6.3 LNS for Scheduling

We now describe a specific implementation of the LNS algorithm for scheduling, shown in Algorithm 2. In this implementation of LNS, the metaheuristic employed is hill climbing, meaning the solution S is only updated if we find an improvement. Other types of metaheuristics have been used such as random walk [24, 43] or simulated annealing [92]. The remaining components required to instantiate LNS are the solution representation, neighbourhood heuristics and search algorithms, which are described in the rest of this section.

6.3.1 Solution Representation

For many scheduling problems, a solution can be derived from a total ordering of activities on each resource. This information is sufficient to compute the start time range of each activity as well as the overall makespan of the schedule [32]. A convenient data structure to represent this type of solution

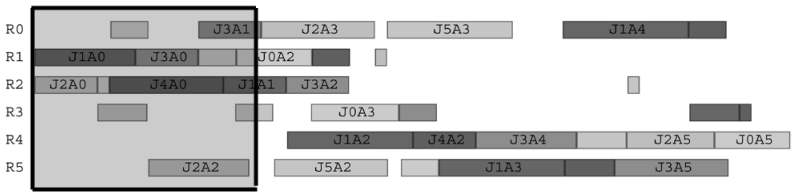
is a linked list. We define the following relations for the linked list. Let $a_j = next(a_i)$ be the activity a_j that *immediately* follows a_i (on the same resource) in the current solution. We define a special dummy activity, a_{r_j}, to indicate the start of the linked list for resource r_j. For example, the first activity a_i scheduled on resource r_j, will have the relation $a_i = next(a_{r_j})$.

To remove the assignment of a particular activity a_i, we simply remove all ordering information that refers to a_i. If there is an activity scheduled before a_i on the resource, e.g., $a_i = next(a_j)$ then we remove this relation from the solution. Likewise, if there is a following activity a_k on the resource, e.g., $a_k = next(a_i)$, then we also remove this relation. For example, if a resource r_j has the sequence $(a_{r_j}, a_1, a_2, a_3, a_4, a_5)$ and we remove a_3 from the solution, we create two sequences (a_{r_j}, a_1, a_2) and (a_4, a_5). Removing the special activity a_{r_j}, representing the start of the list, is not allowed.

The approach of maintaining the sequence of activities by a linked list is different than maintaining the sequence using precedence relations. A precedence relation is an ordering between a pair of activities. For example, the end time of a_1 must precede the start time of a_2. The precedence relations approach is called precedence constraint posting [24] or a partial order schedule [43]. In the above example using a linked list, there is only one possible way to reschedule a_3 to create a sequence that is different from the original sequence: $(a_{r_j}, a_1, a_2, a_4, a_5, a_3)$. In contrast, if precedence relations are used to retain the original order, it would be possible to insert a_3 after any of the activities in the sequence.

6.3.2 Search Algorithm

Once activities have been removed from the current solution, a search algorithm is required to reschedule them. The choice of algorithm is not explored here but is also an area where algorithm control can be applied. See Perron [88] for an example of control being applied to choose the search algorithm. Here, we use the *Texture* algorithm as described in Chapter 5. This choice was made since *Texture* was the strongest performing constructive search algorithm we tried. For further details on the *Texture* algorithm see Section



The time window neighbourhood heuristic selects all activities whose earliest start time is in the time window interval.

Figure 6.1: Time window neighbourhood heuristic.

5.2.2. The application of control methods to search algorithm selection is an interesting area for future work.

6.3.3 Neighbourhood Heuristics

A neighbourhood heuristic is used to select activities from the current solution. The goal of a neighbourhood heuristic is to select activities that, if rescheduled, are likely to lead to an improvement in solution quality. In this section, we describe the neighbourhood heuristics used in experiments in this chapter. For neighbourhood heuristics that take parameters, we denote these parameters with the notation $\pi(X)$ where X represents the name of the parameter.

6.3.3.1 Time Window Neighbourhood

The *Time Window* neighbourhood heuristic selects all the activities that are scheduled to start in a particular time interval. We refer to the interval as a time window. The heuristic returns a list of neighbourhoods by sliding the time window forward across the schedule to include different intervals. Each time window overlaps the preceding one. An example of the *Time Window* neighbourhood heuristic is shown in Figure 6.1 where all of the activities with start times in the shaded box are selected.

Let $est(a_i)$ represent the lower bound of the start time of activity a_i given

the current schedule S. The minimum start time in the schedule is defined by

$$est_{min} = min(est(a_i)|\ a_i \in A) \tag{6.1}$$

and the maximum start time by

$$est_{max} = max(est(a_i)|\ a_i \in A) \tag{6.2}$$

This gives the total time range of the current solution, $range = est_{max} - est_{min}$. We divide $range$ into $\pi(W_t)$ windows, thus each window has the interval length $I = \lceil range/\pi(W_t) \rceil$. Each time window is defined by t_{min} and t_{max} such that $I = t_{max} - t_{min}$. The neighbourhood heuristic returns a list of selections ordered by increasing time intervals, starting with $t_{min} = est_{min}$. Each neighbourhood consists of all activities whose start time is in the interval t_{min} to t_{max}.

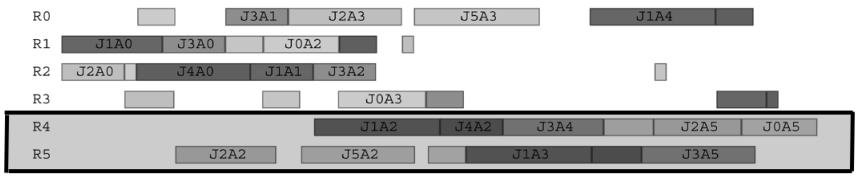
$$Time\ Window = \{a_i \mid a_i \in A, t_{min} \leq est(a_i) \leq t_{max}\} \tag{6.3}$$

For the next interval, we shift the window by a fraction $\pi(S_t)$ of the window interval length I, which gives us $shift_t = \pi(S_t) \times I$. Thus, after selecting the first interval, we add $shift_t$ to both t_{min} and t_{max}. In the case that the last window on $range$ extends past est_{max}, we set t_{max} to the value of est_{max} creating a smaller window for the last interval.

A similar time window neighbourhood is defined in Caseau et al. [22], however that heuristic returns the intervals in a random order. In this dissertation we visit the window neighbourhoods in a deterministic order of increasing times.

6.3.3.2 Resource Load Neighbourhood

The *Resource Load* neighbourhood heuristic returns a neighbourhood determined from a subset of resources. Each neighbourhood consists of all of the activities which are scheduled on the selected resources. Resources are selected in order of decreasing resource load. Resource load is the sum of the durations of all activities currently scheduled on a that resource. Formally,



The *Resource Load* neighbourhood heuristic selects all activities on resources in the resource window.

Figure 6.2: Resource load neighbourhood heuristic.

$$load(R) = \sum_{a_i \in assigned(R)} duration(a_i) \tag{6.4}$$

where $assigned(R)$ is the set of activities scheduled on resource R and $duration(a_i)$ specifies the duration of activity a_i.

The intuition for the *Resource Load* neighbourhood is that the most loaded resources will have a greater chance of conflicts when being scheduled. In a manner similar to the *Time Window* neighbourhood, a neighbourhood consisting of a 'window' of several resources is selected from an ordered list of resources, starting with the most loaded resource and working towards the least loaded resource. We show an example of this in Figure 6.2 where all of the activities on R_4 and R_5 are selected.

Let R be a sorted list n resources, where R_1 is the resource with the maximum load, and R_n is the resource with the minimum load. Given a ratio $\pi(W_r)$ of resources to select, $r = \pi(W_r) \times n$ determines the number of resources to select for each neighbourhood. Let r_{min} be the starting resource index of the window, and r_{max} be the ending resource index of the window. Hence in a neighbourhood, we select all activities on resources $R_{r_{min}}$ to $R_{r_{max}}$ from the sorted list of resources R. Given r_{min} and r_{max}

$$\textit{Resource Load} = \{a_i \mid a_i \in A, a_i \in assigned(R_j), r_{min} \leq j \leq r_{max}\} \tag{6.5}$$

The first neighbourhood sets $r_{min} = 1$ and $r_{max} = r$. For successive neigh-

bourhoods, we add $\pi(S_r)$ to both r_{min} and r_{max}. Therefore $\pi(S_r)$ determines how much the window slides on each successive call. We stop when $r_{max} = n$.

6.3.3.3 Random Neighbourhood

The *Random* neighbourhood heuristic is perhaps the simplest to describe. Each neighbourhood consists of a set of randomly chosen activities. Formally, given a ratio $\pi(R)$ of activities to select from the set of all activities A, we compute $s = \pi(R) \times |A|$. Each time the heuristic is invoked, a random sequence of activities *randseq* is generated with each activity $a_i \in A$ given an index $randseq(a_i)$ in this ordering.

$$Random = \{a_i \mid a_i \in A, randseq(a_i) \leq s\} \tag{6.6}$$

There is no fixed number of neighbourhoods because we continue to draw s activities, with replacement, for as long as we wish. Many existing approaches have used some element of randomization to select activities [6, 22, 29, 84, 89, 90, 99].

6.3.3.4 Cost-Based Neighbourhood

The central idea of the *Cost-Based* neighbourhood is that variables are ranked based on their impact on the objective in the current solution. The first neighbourhood consists of only the most highly ranked variables. Successive neighbourhoods include variables of a lower rank, but keep the previously selected variables as well. Hence, a *Cost-Based* neighbourhood starts with the variables with the highest cost impact and successively adds less important variables in every subsequent neighbourhood. The challenge is to determine an effective ranking strategy which focuses search on the variables that are contributing the most to the objective.

Given a ranking function *rank*, a solution S, and a set of activities A, the *Cost-Based* neighbourhood is defined as

$$Cost\text{-}Based = \{a_j \mid a_j \in A, rank(a_j, S) \leq i\} \tag{6.7}$$

where $rank$ produces a rank value for each activity and i indicates the number of successive calls to the neighbourhood on solution S. Let $maxRank$ be the maximum rank value and for each value $r \in 1..maxRank$ there exists at least one activity a_j where $rank(a_j, S) = r$. That is, the rank values are consecutive and there exists at least one activity per rank value, which implies that the set of activities in each neighbourhood is distinct. Note that the *Cost-Based* neighbourhood increases in size; each successive neighbourhood is a superset of the previous one, eventually including all of the variables.

Since identifying variables that have the highest impact is difficult, in general, a heuristic can be used as a proxy for this impact. For the task of minimizing makespan in the job shop scheduling problem it is well known that to improve a solution it is necessary to reorder some activities along a critical path. A critical path is defined as a sequence of critical activities that are connected by precedence constraints, either induced by a sequence of activities on a resource in the current solution, or by the problem definition. A critical activity a has a slack value of 0 in the current solution, where slack is computed by $slack(a) = lst(a) - est(a)$. In other words, slack represents how much an activity may slide in the schedule before another activity must move, with 'critical' denoting an activity which cannot move at all.

To compute the rank value from the measure of slack, the following procedure is applied. Activities are grouped into subsets containing activities of the same slack value. These subsets are then sorted by slack value, the order of the subsets indicating the rank value. For example, the subset containing activities with the smallest slack value has the highest rank value ($r = 1$) while the subset containing the activities with the highest slack value has the rank value $r = maxRank$. The *Cost-Based* neighbourhood selects the activities in all subsets with rank values of i or less, meaning that each selection is a superset of the last.

The idea of using the critical path to perturb the schedule is standard in many local search approaches [2, 79] as well as in other techniques such as iFlat [24] and iFlatRelax [73]. Likewise, Caseau et al. [22] present a large neighbourhood search scheme that selects fragments of the critical path, however, as far as we know no one has explicitly proposed a method that

reschedules all of the tasks on the critical path within an LNS framework. In the context of vehicle routing, Pisinger et al. [92] introduce a randomized cost-based neighbourhood called *worst removal* that selects a fixed number of activities within a given rank threshold.

6.3.3.5 Growing Neighbourhoods

The *Cost-Based* neighbourhood heuristic has an interesting property: the initial neighbourhood contains a small number of selected activities and each successive neighbourhood increases the number of selected activities. Eventually all of the activities are selected. This property avoids fixing a neighbourhood parameter which is useful since the size of the selected subproblem is free to change during search. Although we do not perform a detailed study here, our intuition is that this allows neighbourhoods to adapt to each problem instance and to adapt as search progresses. However, growing neighbourhoods also carries the risk that the neighbourhoods will become so large that they are too computationally expensive to search efficiently.

We now discuss how the other neighbourhood heuristics described previously can be modified so that they also have the property of growing in size.

- For the *Time Window* neighbourhood, the size of neighbourhood can grow by decreasing the number of time windows, $\pi(W_t)$, and thus increasing the number of activities selected. After one full sweep across the scheduling horizon, $\pi(W_t)$ is decreased until we reach $\pi(W_t) = 1$ where we select all activities.

- The *Resource Load* neighbourhood is modified in a similar fashion. The neighbourhood size grows by modifying $\pi(W_r)$ so that r, the number of selected resources, increases by 1. This has the effect of increasing the neighbourhood until all resources are considered, at which point all activities are selected.

- In our experiments we did not explore growing when applied to *Random* neighbourhoods. However, a policy suggested in Nuijten et al. [84] is

to slowly increase $\pi(R)$ if no solutions are found after a number of iterations.

In our experiments, all neighbourhoods revert to their initial parameter settings when a new solution is found. We determine the initial parameter settings using the method described in Section 6.6. A more sophisticated form of adjusting the parameters of a neighbourhood is presented in the MIMAUSA method [71] that adjusts the neighbourhood size based on experience gained while solving the problem instance. SA-LNS [59] adapts parameters during search using reinforcement learning. See Section 6.10 for further details on SA-LNS. Adaptively tuning parameters during search, such as the neighbourhood size is an interesting area for future work.

6.3.4 Computational Limits

Two computational limits are applied to neighbourhood heuristics. The purpose of these limits is to distribute computation effort among heuristics and the search of neighbourhoods. The time slicing limit shares time among the different heuristics while the neighbourhood search fail limit is applied to the exploration of each neighbourhood thereby sharing effort while applying a particular heuristic during a time slice. After the description of these limits, we give an example and details of how these limits are set.

6.3.4.1 Neighbourhood Heuristic Time Slicing

The difference in computation time between neighbourhood heuristics can be significant. Some heuristics may select small neighbourhoods which are very fast to search while other heuristics select larger neighbourhoods that take longer to search. This poses a problem when sharing computational resources among different heuristics. We address this by applying each neighbourhood for a fixed amount of time, referred to as a time slice. During a time slice, the heuristic will continue to select neighbourhoods to search until the duration of the time slice has elapsed.

duration	slice 1 - nh_i 1 second		slice 2 - nh_jnn 1 second				slice 3 - nh_i 1 second		
fail limit	nh_i1 100	nh_i2 100	nh_j1 100	nh_j2 100	nh_j3 100	nh_j4 100	nh_i3 100	nh_i4 100	nh_i5 100

Table 6.1: Example of neighbourhood heuristic time slicing and neighbourhood search limit.

6.3.4.2 Neighbourhood Search Fail Limit

Recall that a neighbourhood heuristic will select a series of different neighbourhoods to search. It is likely that some of these neighbourhoods may require a large amount of computation time to either find an improved solution or determine that no such improvement exists. For this reason, we restrict search effort on a each neighbourhood by limiting the total number of backtracks, or fails, which can occur. This allows each neighbourhood heuristic to explore many different neighbourhoods, increasing the chance that we find a neighbourhood that contains an improving move that can be found with a reasonable search effort.

6.3.4.3 Example

An example of time slices and fail limits is shown in Table 6.1. In this example, we alternate between two neighbourhood heuristics, nh_i and nh_jnnn. Each neighbourhood heuristic is given a time slice of 1 second. During this 1 second slice, multiple neighbourhoods are searched, each with a failure limit of 100. For example, nh_i2 represents the second neighbourhood selected by heuristic nh_i during time slice 1. The time taken to search each neighbourhood differs, which accounts for the different number of neighbourhoods seen in each slice. Note that when we get to slice 3, we continue with the next available neighbourhood, nh_i3, of the heuristic nh_i. In this example, no improved solution has been found. In the case that an improved solution is found, all neighbourhoods are reset and initialized to the new solution.

6.3.4.4 Adapting Limits

The duration of the time slice is initially set to 1 second and the fail limit set to 100. However, in the case that no improved solutions are found after 10 slices, we double these limits. The motivation for increasing limits is twofold. For unknown problems, it is hard to determine an effective limit *a priori*. The second motivation is the observation that as optimization successively finds improved solutions, search becomes more difficult, indicating that a fixed limit is not suitable. Increasing limits allows more effort to be spent exploring neighbourhoods as search proceeds. Increasing time limits was an important feature of the switching strategy in Chapter 5.

6.4 LNS Control Problem

It is clear that the configuration of an LNS algorithm is not a trivial exercise. The components and parameters described in the previous section highlight the complexity in applying these methods. In this section, we propose that the configuration of LNS is an algorithm control problem that can be addressed by low knowledge control methods. In the following sections, we perform a series of experiments that apply low knowledge control methods to large neighbourhood search.

Parameter Tuning In Section 6.6 we present a method that tunes the parameters of large neighbourhood search. This method applies a set of neighbourhood heuristics and parameters to a set of training problem instances. The output of the method is a ranking of the effectiveness of each parameter configuration. The best parameters obtained through this process are then used in the experiments that follow.

Pure Neighbourhoods The performance of each neighbourhood heuristic is evaluated in Section 6.7. We refer to an LNS configuration that uses only a single heuristic as a *pure* algorithm. We show that the performance of each neighbourhood heuristics varies over time, and across problem instances. It appears that choosing a single neighbourhood

does not give robust performance over time and that performance may be improved through the use of an on-line control method.

Control Methods Control methods for LNS are introduced in Section 6.8. These methods apply a set of neighbourhood heuristics to a problem instance and adapt weights based on the observed performance. These weights are used to bias selection or the allocation of computation time among heuristics. Experiments show the benefit of the control methods in providing robust performance and in some cases, synergy, where the combined methods outperform pure algorithms on a problem instance.

Evaluation on New Instances The experiments in Section 6.9 apply our methods to a set of challenging benchmark problem instances. In these experiments we do not perform any parameter tuning. Instead, we use the parameter settings from an earlier experiment on a different set of problem instances. The results show that when the performance on problem instances varies significantly from those seen in the training set, the control methods are able to adapt to perform well across all problem instances.

6.5 Experimental Details

In this section, we present the problem instances and evaluation criteria that are used in the experiments in this chapter. We also discuss some details required to perform the experiments.

6.5.1 Software and Hardware

All of the algorithms in this chapter were implemented in C++ using the constraint programming toolkit, ILOG Scheduler 6.0. Experiments were executed on a Pentium IV 1.8 Ghz CPU with 512MB of RAM running the Linux operating system.

6.5.2 Problem Instances

We evaluate our ideas on three sets of job shop scheduling problems (JSPs). We use two randomly generated problem sets for in-depth experiments and then provide an analysis of our approach on a set of standard benchmark problem instances that range in size. We generated our own problem instances in order to have a larger sample of problems to run experiments on. However, for comparison with other approaches we provide results on a standard benchmark, the Taillard JSP instances [104].

The first generated set, 20x20, consists of medium sized scheduling problems with 20 jobs and 20 machines, each instance containing 400 activities. The second generated set, 40x40, contains larger problem instances with 40 jobs and 40 machines, each instance containing 1600 activities. The activity durations are drawn randomly with uniform probability from the interval [1, 99]. The routing of each job is randomly assigned so that each job has exactly one activity on each machine, but each job requires machines in a random order. Each set consists of 100 problem instances, which are divided into a training set of 60 instances and a test set of 40 instances.

We also consider Taillard's set of standard job shop benchmarks [104]. Performance is evaluated on the problem instances ta11 to ta80, which represent problems ranging in size from 20 to 100 jobs and 15 to 20 machines. A limitation of these problem sets is that there are only 10 instances available of each problem size, so we use the benchmark problem set as a validation of our approach rather than for our detailed study.

6.5.3 Evaluation Criteria

Our primary evaluation criteria is mean relative error (MRE), a measure of the mean extent to which an algorithm finds solutions worse than the best solutions found during our experiments. MRE is defined as follows:

$$MRE(a, K) = \frac{1}{|K|} \sum_{k \in K} \frac{c(a, k) - c^*(k)}{c^*(k)} \tag{6.8}$$

where K is a set of problem instances, $c(a, k)$ is the lowest cost solution found

by algorithm a on problem instance k, and $c^*(k)$ is the lowest cost solution found during our experiments for problem instance k.

We also report the fraction of problems in each set for which the algorithm found the best known solution, referred to as fraction best (FB).

FB can show that an algorithm has good performance on some problem instances even if it has poor performance on other instances, something that is obscured with MRE.

FB is defined as:

$$FB(a, K) = \frac{|best(a, K)|}{|K|} \tag{6.9}$$

where: $best(a, K)$ is the set of solutions for $k \in K$ where $c^*(k) = c(a, k)$.

6.5.4 Best Algorithm

In addition to the performance results of the algorithms that are compared in each experiment we define two further results, *BestMRE* and *BestInstance*. The first result, *BestMRE(A,t)*, is the result of the algorithm $a \in A$ that had the best mean performance (lowest MRE) on a problem set K at time t. This represents applying the best single algorithm at time t to all instances in K. The second set of results, *BestInstance(A,t)*, is the best result found on each problem instance by any of the algorithms $a \in A$. If a single algorithm dominates all others then *BestMRE* will equal *BestInstance*. Conceptually, *BestInstance* is akin to running all of the algorithms in parallel on each instance and then taking the best result.

For example, *BestMRE(P,t)* are the results produced by algorithm $a \in P$, where P is the set of pure algorithms and a is the single algorithm that had the lowest MRE at time limit t. *BestMRE(P,t)* is equivalent to the *best-all* method from Section 5.5.3 in Chapter 5. Likewise, *BestInstance(P,t)* represents the best result on each instance that was observed by any of the pure algorithms $a \in P$. *BestInstance(P,t)* is the same as *best-instance*, as defined in Section 5.5.3.

6.5.5 Statistical Analysis Method

In this section, we describe the design of the statistical analysis used in the experimental results sections of this chapter. Differences in performance over time are compared between algorithms, against the best overall algorithm, and against the best solution found by any algorithm. The analysis considers observations at 30 time points across the runs of each algorithm. The result of the analysis is a post-hoc confirmation of differences over time.

The analysis procedure is performed on each set of experiment data to check the *experiment null hypothesis* that the mean performance observed is equal, which is the ANOVA result. If the experiment null hypothesis is rejected, we check each *comparison null hypothesis* using Tukey's Honest Significant Difference (HSD) test. A comparison null hypothesis determines the evidence that a particular pair of the population means is equal. If a comparison null hypothesis is rejected, we say that there is a significant difference between that pair of algorithms. Since multiple pair-wise comparisons are performed, the Tukey HSD test is used to reduce the occurrence of Type I errors (evidence that a difference exists when there is none).

The following analysis is applied to each set of experiment data. The factor, or independent variable, is the algorithm with each level representing a specific algorithm $a \in A$. The dependent variable is the algorithm performance, measured in relative error (RE), observed by running the algorithm on each problem instance in the data set $k \in K$. Therefore, each data set is analyzed by producing a single table containing two columns: *Algorithm* and RE. The data from one of these tables is fitted to a linear model which is passed to the ANOVA test to determine if there is evidence to reject the experiment null hypothesis, or in other words that there is a significant difference in algorithm performance in the set A. The Tukey HSD test takes the same input as ANOVA but is used to determine evidence of a significant pair-wise difference between each of the algorithms in A, the rejection of the comparison null hypothesis.

For a particular problem set K and set of algorithms A, we perform this procedure for a set of time points TP over the execution of the algorithms.

Therefore, the analysis is performed a total of $|TP|$ times for each set K. This analysis gives insight as to the performance difference over time. We refer to these experiment data sets as $performanceTable(A, K, t)$ and the related analysis as the *30 time points analysis*, since $|TP| = 30$ in our experiments. To determine the evidence of a significant difference in performance across all time limits, we merge the tables from each time limit into a single table, $performanceTable(A, K)$, which is used in the *all time points analysis*.

In our tests, we use the condition that $p < 0.005$ (or $\alpha = 0.005$) to determine statistical significance. However, in some cases, we are performing many comparisons between algorithms, specifically when we look at algorithm performance at 30 time points. The Bonferroni adjusted p value for $n = 30$ such comparisons is approximated by α/n which is 0.00002. The formula for the error rate in this experiment is $1 - (1 - \alpha)^n$ which is 0.13962, or a 14% chance that we will incorrectly reject the null hypothesis and claim there exists at least one significant difference when there is none.

However, we note the critique of Perneger [87] where a case is made against these types of adjustments. In particular, we take the argument that we are not concerned with the universal null hypothesis (for example, that there is no difference in all 30 time points) and are instead concerned with the differences at each time point. Further, to directly tackle the criticism of inflated error, we perform a separate test using all time points ($performanceTable(A, K)$) in single comparison test.

6.6 Parameter Tuning

In this section, we present a method for tuning the parameters of neighbourhood heuristics and the results of applying this method to two training sets of problem instances. The results are used to determine the best parameter combinations which are then used in the remainder of this chapter.

Algorithm 3: Tuning evaluation procedure

```
1: S := Create initial solution
2: while Not optimal and time left do
3:    for all Π ∈ P do
4:       S_Π := result of applying Π to S
5:       Record performance of Π
6:    end for
7:    minCost := min(c(S_Π)|Π ∈ P)
8:    BestS := selectRandom(S_Π|minCost = c(S_Π), Π ∈ P))
9:    if c(BestS) < c(S) then
10:      S := BestS
11:   end if
12: end while
```

6.6.1 Tuning Procedure

We now present a procedure for evaluating neighbourhood heuristics shown in Algorithm 3. The procedure takes as input a set of possible parameter configurations P and a problem instance. The procedure operates by applying every parameter configuration Π to every intermediate solution S that is found. A parameter configuration Π consists of a neighbourhood heuristic a and a set of parameter values for each parameter $\pi(X)$ that is required by a. The result of running each parameter configuration is recorded for analysis, and the best improving solution found by any of the parameter combinations is taken to be the next intermediate solution. In the case of ties, we randomly select one of the best solutions. If no improved solution is found, then we do not update S.

The result of this procedure is a history of performance for each neighbourhood heuristic on a problem instance. The procedure is applied to each problem instance in the training set and the information generated is then analyzed to determine the best performing parameter configurations. We present a simple analysis procedure in Section 6.6.3 and the resulting parameter configurations in Section 6.6.4.

Existing algorithm tuning mechanisms [16, 51, 53] operate by executing each parameter configuration independently on each problem instance. Al-

though these approaches can be used to tune heuristics for LNS, there is a serious shortcoming in evaluating each configuration in isolation, since we are interested in a system that will combine heuristics. In our procedure, we apply each heuristic configuration at every control point during search. We believe that our tuning procedure is superior in this context because all parameter configurations are evaluated together over the course of execution on a single problem instance.

There are clear improvements to this procedure that are not explored in this chapter. For example, evaluating every possible parameter configuration is expensive. We are interested in applying approaches that focus evaluation on the most promising configurations [53]. Furthermore, it would be interesting to learn synergies between configurations, for example, to detect if certain configurations work better when applied in a particular order. Investigations of tuning mechanisms for combinations of heuristics is a promising area for future work.

6.6.2 Parameter Configurations

The following parameters are explored for each neighbourhood heuristic. The window size for the *Time Window* and *Resource Load* neighbourhood heuristics must be determined, including how much to overlap these windows. The *Random* neighbourhood heuristic requires a ratio of activities to select given the problem size. While we could also optimize the parameters of the search procedure from Section 6.3.2 we do not explore that here.

A total of 21 different configurations were applied. Each configuration is applied for a limited amount of time using the slicing approach described in Section 6.3.4. For the *Time Window* neighbourhood heuristic, we tried time window values $\pi(W_t)$ of 2 to 20 in steps of 2 and used a single window overlap ratio of $\pi(S_t) = 50\%$. The *Resource Load* heuristic requires a parameter $\pi(W_r)$ to specify the ratio of resources to select in a subproblem. We tried the following values for $\pi(W_r)$ of $5\%, 25\%, 45\%, 65\%$ and 85%. The single overlap value of $\pi(S_r) = 1$ was used, indicating that 1 resource will be added and 1 resource will be removed as subsequent neighbourhoods are visited. Finally,

the *Random* neighbourhood heuristic requires a parameter $\pi(R)$ specifying how many activities to select. We tried values of $\pi(R) = 5\%, 25\%, 45\%, 65\%$ and 85%. We also included the *Cost-Based* neighbourhood heuristic in these experiments which takes no parameters.

6.6.3 Analysis of Results

The results of running the tuning procedure are shown in Table 6.2 and 6.3. Each tuning run was applied to 60 problem instances from the training set for a duration of 30 minutes on each instance. With 21 parameter configurations, 30 minutes is equivalent to running each configuration for approximately 85 seconds on every instance. The average improvement in solution quality (an absolute value in time units rather than a relative error) is shown along with the probability of each neighbourhood heuristic making an improvement. The probability is computed simply by taking the number of times an improvement was found divided by the number of times that neighbourhood heuristic was tried. We multiply the probability of improvement by the average improvement value to compute the utility of each neighbourhood heuristic.

We note that the above analysis of performance is only one of many possible approaches. For example, it would be interesting to analyze the results of each parameter configuration over time. This would give insight into how the best parameter configurations change at different stages in search. Such information could be exploited to develop set of parameter configurations that change as search progresses.

6.6.4 Best Parameters

The best parameters configurations change depending on the problem set. For the medium problem set, the best configurations are *Time Window* with $\pi(W_t) = 2$ windows, *Resource Load* with $\pi(W_r) = 65\%$ resources and *Random* with $\pi(R) = 45\%$ of the activities, and then finally, the *Cost-Based* neighbourhood heuristic. For the larger problems we see a different picture. Now the *Cost-Based* neighbourhood heuristic is the best configuration,

Parameter	Value	Probability of Improvement	Average Improvement	Utility (Prob x Avg)
Time Window $\pi(W_t)$	2	0.24	10.70	2.57
Random $\pi(R)$	45%	0.21	8.25	1.73
Resource Load $\pi(W_r)$	65%	0.21	8.16	1.71
Cost-Based	-	0.15	11.30	1.70
Resource Load $\pi(W_r)$	45%	0.15	9.09	1.36
Random $\pi(R)$	25%	0.16	6.34	1.01
Resource Load $\pi(W_r)$	85%	0.10	9.03	0.90
Time Window $\pi(W_t)$	4	0.12	7.23	0.87
Resource Load $\pi(W_r)$	25%	0.08	6.02	0.48
Random $\pi(R)$	85%	0.06	7.71	0.46
Time Window $\pi(W_t)$	6	0.09	4.25	0.38
Random $\pi(R)$	65%	0.08	3.79	0.30
Time Window $\pi(W_t)$	8	0.08	3.21	0.26
Time Window $\pi(W_t)$	10	0.07	3.07	0.21
Time Window $\pi(W_t)$	12	0.07	2.90	0.20
Time Window $\pi(W_t)$	14	0.06	2.67	0.16
Time Window $\pi(W_t)$	16	0.05	2.57	0.13
Time Window $\pi(W_t)$	18	0.04	2.27	0.09
Random $\pi(R)$	5%	0.03	2.21	0.07
Time Window $\pi(W_t)$	20	0.03	1.93	0.06
Resource Load $\pi(W_r)$	5%	0.02	1.71	0.03

Table 6.2: Tuning performance results for 20x20 problems.

Parameter	Value	Probability of Improvement	Average Improvement	Utility (Prob x Avg)
Cost-Based	-	0.45	6.46	2.91
Time Window $\pi(W_t)$	8	0.13	3.05	0.40
Time Window $\pi(W_t)$	10	0.12	3.05	0.37
Time Window $\pi(W_t)$	12	0.11	3.17	0.35
Resource Load $\pi(W_r)$	25%	0.11	2.90	0.32
Time Window $\pi(W_t)$	6	0.11	2.66	0.29
Time Window $\pi(W_t)$	14	0.09	3.02	0.27
Time Window $\pi(W_t)$	4	0.08	2.29	0.18
Time Window $\pi(W_t)$	16	0.07	2.38	0.17
Random $\pi(R)$	25%	0.07	2.32	0.16
Time Window $\pi(W_t)$	18	0.06	2.35	0.14
Resource Load $\pi(W_r)$	45%	0.07	1.71	0.12
Time Window $\pi(W_t)$	20	0.05	2.18	0.11
Resource Load $\pi(W_r)$	5%	0.05	1.94	0.10
Time Window $\pi(W_t)$	2	0.04	1.63	0.07
Resource Load $\pi(W_r)$	65%	0.03	1.36	0.04
Random $\pi(R)$	5%	0.02	1.43	0.03
Random $\pi(R)$	65%	0.02	1.00	0.02
Random $\pi(R)$	45%	0	0	0
Random $\pi(R)$	85%	0	0	0
Resource Load $\pi(W_r)$	85%	0	0	0

Table 6.3: Tuning performance results for 40x40 problems.

in terms of average improvement and probability of improvement. Next is *Time Window* with $\pi(W_t) = 8$, 10, and 12 windows. We choose only the best parameter setting of $\pi(W_t) = 8$. The next best neighbourhood heuristic is the *Resource Load* neighbourhood heuristic with a setting of $\pi(W_r) = 25\%$. Finally, we have *Random*, with a setting of $\pi(R) = 25\%$.

6.7 Pure Neighbourhood LNS

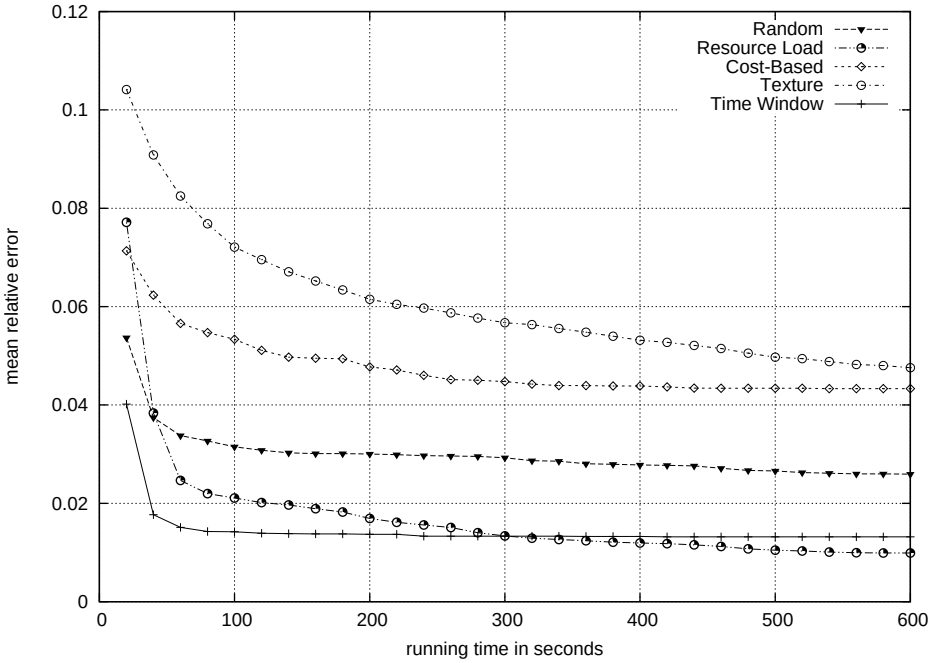
In this section we evaluate the performance of each neighbourhood heuristic independently. We refer to an LNS configuration that uses only a single heuristic as a *pure* neighbourhood heuristic algorithm. The results show strong performance but the best heuristic differs depending on the time limit and problem set. Since no single method dominates, this result indicates that algorithm control methods may boost performance by choosing the best performing neighbourhood(s) for each problem instance.

6.7.1 Experiments

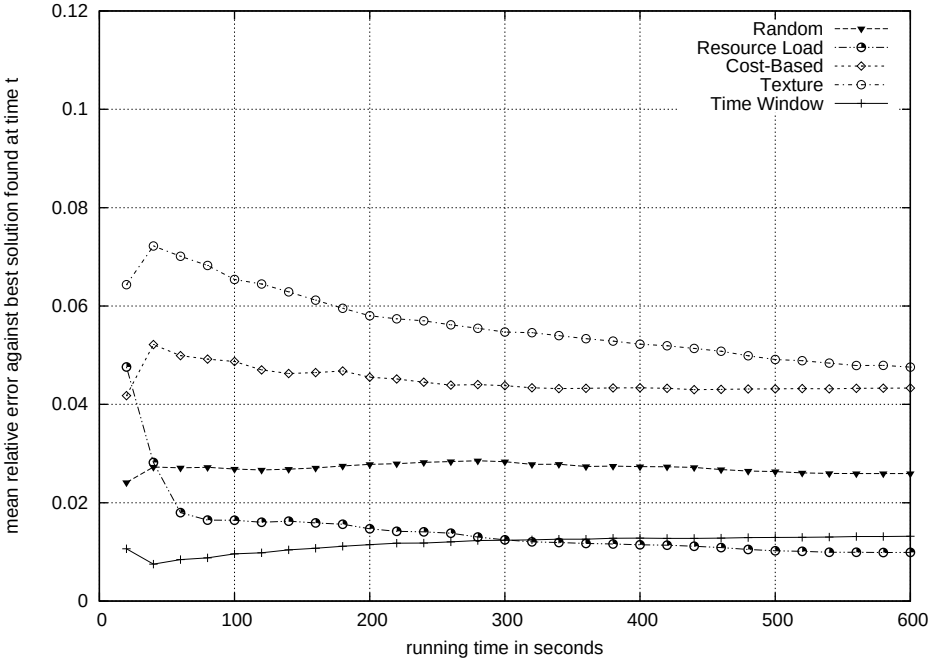
Using the parameter settings determined in Section 6.6, we ran each of the neighbourhood heuristics independently on the 40 problem instances from each test set. For the medium sized 20x20 problems, we used an overall time limit of 10 minutes. For the large 40x40 problems, we used a time limit of 20 minutes. For comparison, we include the results of applying the search method, *Texture*, without the large neighbourhood search procedure.

We show the mean relative error (MRE) results in Figure 6.3 and Figure 6.4. In the top of the each figure, we show the MRE against the best solution found overall, which gives an indication of how fast each algorithm converges. On the bottom, we show MRE(t) which is the mean relative error against the best solution found at each time t. MRE(t) is useful to see how far away each algorithm is from the best solutions found at each time limit t.

On the 20x20 problem set, the best performing neighbourhood heuristics are *Time Window* and *Resource Load*. Initially *Time Window* leads, but is overtaken by *Resource Load* mid way through the run. The *Random*

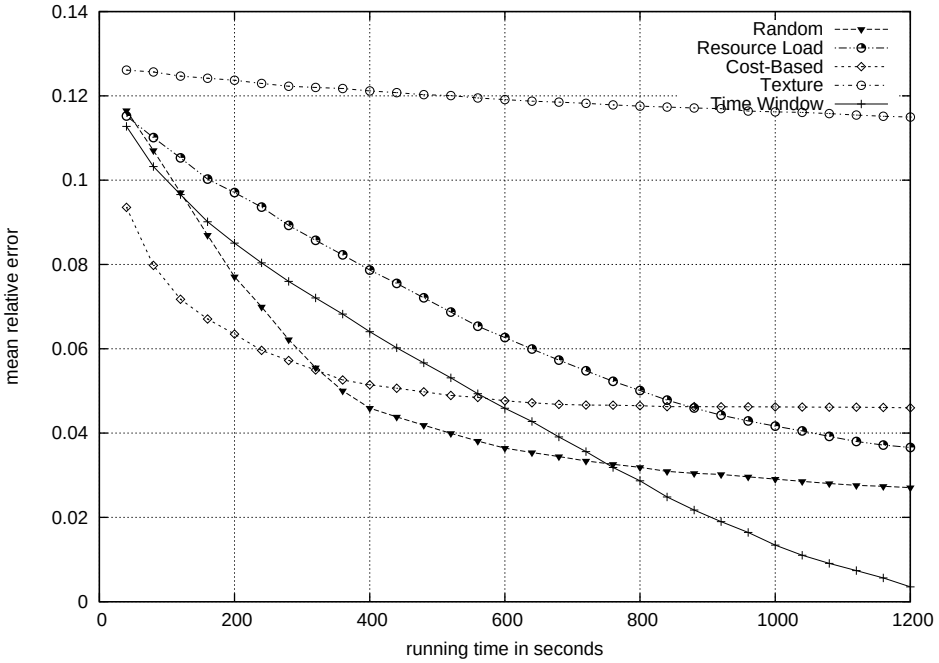


(a) Mean relative error against the best solution over all time limits.

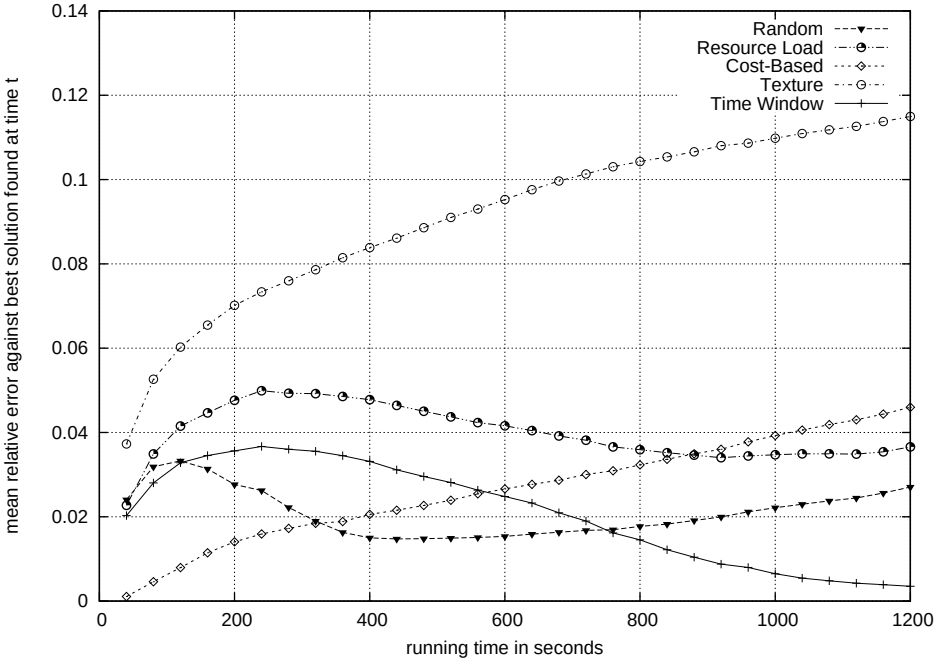


(b) Mean relative error (MRE(t)) against the best solution found at time t.

Figure 6.3: Pure neighbourhood heuristics run independently on the 20x20 problem set.

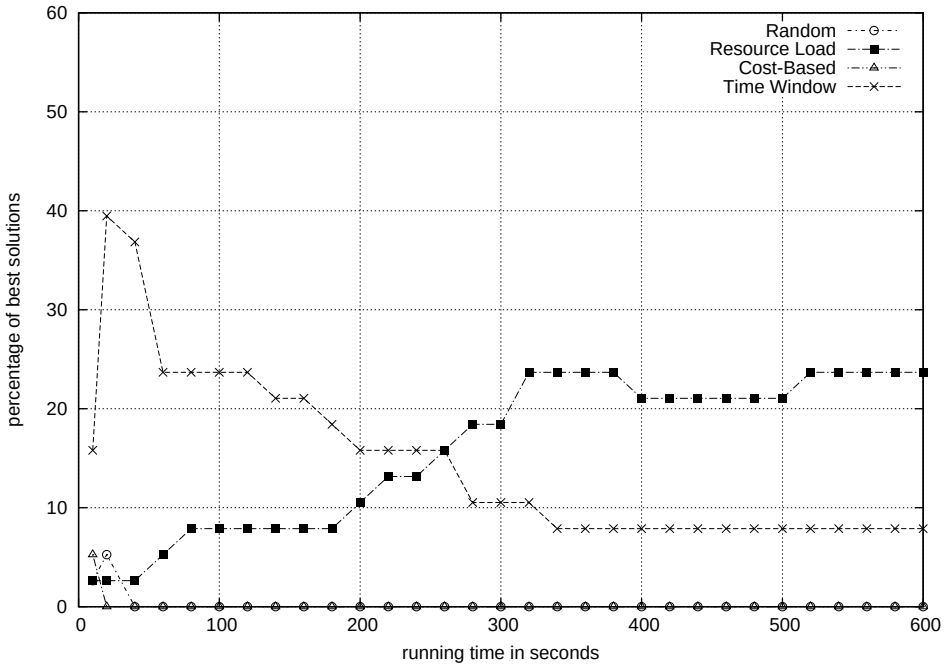


(a) Mean relative error against the best solution over all time limits.

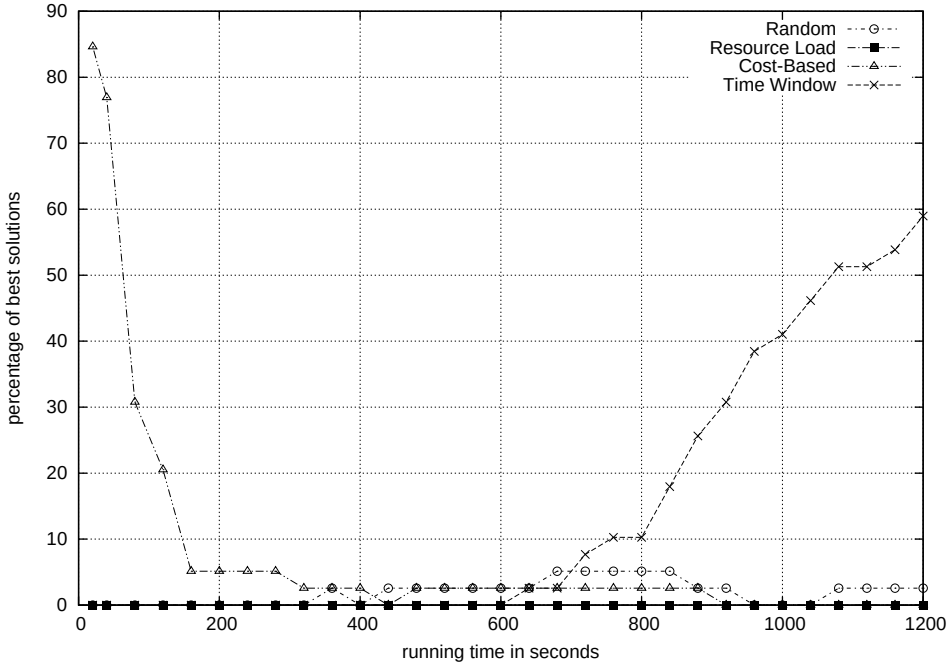


(b) Mean relative error (MRE(t)) against the best solution found at time t.

Figure 6.4: Pure neighbourhood heuristics run independently on the 40x40 problem set.



(a) 20x20 problem set.



(b) 40x40 problem set.

Figure 6.5: Best solutions found by pure neighbourhood heuristics.

neighbourhood heuristic performs worse, but not as badly as the *Cost-Based* neighbourhood which is far worse than all other heuristics. On the 40x40 problem set, the picture is different. Here the *Cost-Based* neighbourhood heuristic is a very strong performer at the start of search. At around 300 seconds the *Random* neighbourhood heuristic takes the leads until the *Time Window* neighbourhood heuristic overtakes it at 800 seconds. The *Resource Load* neighbourhood heuristic performs worse than the other neighbourhood heuristics on this problem set. The *Texture* method performs worse than any neighbourhood heuristics, far worse than the best performers. On the 40x40 set, the *Texture* method is hardly able to improve the solution quality at all. We do not include *Texture* in any further analysis.

In Figure 6.5 we show the fraction of best solutions found over time by each neighbourhood heuristic. This gives a different perspective that allows us to see if some heuristics perform well in some cases but poorly in others. On the 20x20 results we can see that, out of the pure neighbourhood heuristics, *Resource Load* and *Time Window* find a reasonable number of the best solutions. On the 40x40 results, we again see that two neighbourhood heuristics dominate the discovery of best solutions, *Cost-Based* and *Time Window* and again at either end of the time scale. In both problem sets, the number of best solutions found in the middle of the run drops. Recall that the best solution may be found by an algorithm from later experiments in this chapter, which is why the results in Figure 6.5 do not sum to 100%.

To determine the statistical significance of these results as they vary over time, we performed the analysis as described in Section 6.5.5. The analysis was performed at 30 time points through the run; every 20 seconds on the 20x20 set and every 40 seconds on the 40x40 set. We compared the $|P| = 4$ pure neighbourhood heuristics and *BestInstance(P,t)* for a total of 5 factors. We also included *BestMRE(P,T)* and *BestMRE(P,t)* however this does not introduce any additional factors as these results are already contained in the result for the pure algorithms. Each set of results contains $n = 40$ performance measurements for each factor. We refer to this analysis as the *30 time points analysis*. For each problem set we show a summary of the significant differences ($p \leq 0.005$) over the 30 time points.

The following tables are used to summarize the results of the 30 time points analysis:

- Table 6.4(a) shows the a comparison against the best pure algorithm at the end of each run (time T), referred to as *BestMRE(P,T)*. This comparison evaluates how well each neighbourhood heuristic does against applying a single heuristic across all problem instances over all time limits. *BestMRE(P,T)* is the typical approach used when selecting the best algorithm for a fixed time limit T.

- Table 6.4(b) shows the results of comparing against the single best, by MRE(t), pure method at each time point, which we refer to as *BestMRE(P,t)*. In this comparison each neighbourhood heuristic is compared against the best single method, for each time limit, across all problem instances. Unlike *BestMRE(P,T)*, the algorithm that is *BestMRE(P,t)* changes depending on the value of t.

- Table 6.4(c) shows the difference between each heuristic and *BestInstance(P,t)*, the best solution found by any of the neighbourhoods on each problem instance. No pure neighbourhood can perform better than *BestInstance(P,t)*.

In each of these tables we show a count of time points for which there was evidence of a significant difference for a neighbourhood heuristic, denoted by W (worse) and B (better). We also show when a heuristic was the same, denoted by S (same), which means the comparison was actually against the same data set, which occurs in the case of *BestMRE*. In the case that there is no evidence of a difference in performance,[12] this is represented with the symbol '?'.

We now discuss the results shown in these tables. In Table 6.4(a), we see that *BestMRE(P,T)* was *Resource Load* on the 20x20 set and *Time Window* on the 40x40 set, as indicated by $S = 30$. On the 20x20 set, *Random* and

[12]In statistical testing, not rejecting the null hypothesis implies that the two means *may* be the same, but is not conclusive. For this reason, we state this as 'no evidence of a significant difference'.

Table 6.4: Count of significant differences over *30 time points* for pure neighborhood heuristics and *BestMRE* and *BestInstance* of pure neighborhood heuristics.

(a) *BestMRE(P,T)* compared to pure neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
Random	27	0	1	2	9	0	10	11
Resource Load	0	30	0	0	27	0	0	3
Cost-Based	29	0	0	1	13	0	11	6
Time Window	0	0	2	28	0	30	0	0

(b) *BestMRE(P,t)* compared to pure neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
Random	29	0	0	1	15	10	0	5
Resource Load	2	15	0	13	30	0	0	0
Cost-Based	30	0	0	0	18	8	0	4
Time Window	0	15	0	15	15	12	0	3

(c) *BestInstance(P,t)* compared to pure neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
Random	30	0	0	0	16	0	0	14
Resource Load	5	0	0	25	30	0	0	0
Cost-Based	30	0	0	0	20	0	0	10
Time Window	0	0	0	30	16	0	0	14

W/S/B/? denotes significantly Worse, Same, significantly Better and ? no evidence of a significant difference.

Cost-Based perform worse than *BestMRE(P,T)* on nearly all of the time points, while *Time Window* performs better on 2 time points and there is no evidence of a significant difference on the remaining 28 time points. On the 40x40 problem set, the results are more mixed, with *Resource Load* performing the worst on 27 of the time points. The other heuristics, *Random* and *Cost-Based*, perform better and worse on roughly a third of the time points respectively. Similar results are shown in Table 6.4(b), where we compare each neighbourhood heuristic against *BestMRE(P,t)*. These tables make it clear how often each algorithm had the best MRE. On the 20x20 set *Resource Load* and *Time Window* are both the best at $S = 15$ time points each. On the 40x40 set, the best performance is split among *Random*, *Cost-Based* and *Time Window*. Table 6.4(c) shows a comparison against the best solution found by any heuristic on each problem instance. On the 20x20 set, there is no evidence that solutions found by the *Time Window* heuristic are worse. *Resource Load* also performs strongly with evidence that worse solutions are found on only 5 time points. Meanwhile, *Random* and *Cost-Based* perform significantly worse on all time points. On the 40x40 set, *Resource Load* is worse on all time points, *Cost-Based* is worse on 20 time points, and *Random* and *Time Window* are both worse on 16 time points. For the remaining time points, there is no evidence of a significant difference.

To determine the mean difference over the entire run, we merge the results from the 30 time points into a single result set for each algorithm containing $n = 30 \times 40 = 1200$ measurements. In this analysis there are 6 factors: the 4 pure neighbourhood heuristics, *BestInstance(P,t)* and *BestMRE(P,t)*. There is one more factor in this analysis since *BestMRE(P,t)* can change at each time point and this required the creation of a new data set. We refer to this analysis as the *all time points analysis*. For each problem set we show a table of the pair-wise difference in MRE and the statistical significance (p value) of the difference between algorithms.

The results of this comparison between pure neighbourhood heuristics is shown in Tables 6.5(a) and 6.5(b). On the 20x20 problem set, *Time Window* performs better than all other heuristics, with *Resource Load* following a close second, then *Random* and finally *Cost-Based* which performs worse

Table 6.5: Mean difference over *all time points* for pure neighborhood heuristics.

(a) 20x20 problem set.

	Random	Resource Load	Cost-Based	Time Window
Random		**0.01254**	**-0.01774**	**0.01533**
		(0.00000)	(0.00000)	(0.00000)
Resource Load	**-0.01254**		**-0.03028**	**0.00279**
	(0.00000)		(0.00000)	(0.00006)
Cost-Based	**0.01774**	**0.03028**		**0.03307**
	(0.00000)	(0.00000)		(0.00000)
Time Window	**-0.01533**	**-0.00279**	**-0.03307**	
	(0.00000)	(0.00006)	(0.00000)	

(b) 40x40 problem set.

	Random	Resource Load	Cost-Based	Time Window
Random		**-0.01884**	**-0.00558**	0.00004
		(0.00000)	(0.00000)	(0.99998)
Resource Load	**0.01884**		**0.01326**	**0.01888**
	(0.00000)		(0.00000)	(0.00000)
Cost-Based	**0.00558**	**-0.01326**		**0.00562**
	(0.00000)	(0.00000)		(0.00000)
Time Window	-0.00004	**-0.01888**	**-0.00562**	
	(0.99998)	(0.00000)	(0.00000)	

Significant differences ($p \leq 0.005$) are **shown in bold** and p values are shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

Table 6.6: Mean difference over *all time points* for pure neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics.

(a) 20x20 problem set.

	BestMRE(P,T)	*BestMRE(P,t)*	*BestInstance(P,t)*
Random	**0.01254**	**0.01633**	**0.02034**
	(0.00000)	(0.00000)	(0.00000)
Resource Load	0.00000	**0.00380**	**0.00780**
	(1.00000)	(0.00000)	(0.00000)
Cost-Based	**0.03028**	**0.03407**	**0.03808**
	(0.00000)	(0.00000)	(0.00000)
Time Window	**-0.00279**	0.00101	**0.00501**
	(0.00005)	(0.61551)	(0.00000)

(b) 40x40 problem set.

	BestMRE(P,T)	*BestMRE(P,t)*	*BestInstance(P,t)*
Random	0.00004	**0.00952**	**0.01144**
	(1.00000)	(0.00000)	(0.00000)
Resource Load	**0.01888**	**0.02836**	**0.03028**
	(0.00000)	(0.00000)	(0.00000)
Cost-Based	**0.00562**	**0.01510**	**0.01702**
	(0.00001)	(0.00000)	(0.00000)
Time Window	0.00000	**0.00948**	**0.01140**
	(1.00000)	(0.00000)	(0.00000)

Significant differences ($p \leq 0.005$) are **shown in bold** and p values are shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

than all other heuristics. On the 40x40 problem set both *Time Window* and *Random* outperform the other two pure heuristics, with no evidence of difference between them. Finally, *Cost-Based* outperforms *Resource Load*.

In Tables 6.6(a) and 6.6(b) we compare performance differences against *BestMRE* and *BestInstance* over all time points. On the 20x20 set, we see the best performing heuristics, *Resource Load* and *Time Window*, are very close (less than 1%) to the performance of *BestMRE* and *BestInstance*. *Random* and *Cost-Based* perform worse, and are 2% and 3.8% worse than *BestInstance* respectively. On the 40x40 set, the best performing neighbourhoods are *Time Window* and *Random*, which are 1.1% worse than *BestInstance*. *Cost-Based* is 1.7% worse than *BestInstance* while *Resource Load* performs the worst at 3% worse.

6.7.2 Discussion

On both problem sets, all of the neighbourhood heuristics outperform the state-of-the art tree search algorithm, *Texture*. The performance improvement on the 40x40 problems is even more dramatic than the 20x20 problems. The search algorithm is barely able to improve the solution at all on the larger problem instances. This type of result has led people to believe that constraint based search methods do not scale, and indeed, it seems that in the case of optimization, a single tree search is not an effective method on large problems. However, when we apply large neighbourhood search, the same search method performs well when neighbourhood heuristics focus and restrict search effort.

The best performing neighbourhood heuristic changes depending on the problem set, and indeed, even depending on the amount of time allowed for processing. On the 20x20 problems, the *Resource Load* neighbourhood heuristic eventually outperforms all of the other neighbourhood heuristics, except at the start when the *Time Window* neighbourhood heuristic performs best. Although it is obscured on the 20x20 graph, the *Cost-Based* neighbourhood heuristic is actually the strongest performer at the very beginning, however it is quickly overtaken. This behaviour is repeated on the

40x40 problems, although here it is the case that the *Cost-Based* neighbourhood heuristic outperforms other neighbourhood heuristics for a longer time.

In our parameter tuning experiment in Section 6.6, we showed that the *Cost-Based* neighbourhood heuristic consistently produced the best average improvement of all heuristics. However, as the performance on the 20x20 problems shows, it starts to stagnate quite quickly. Unlike other neighbourhood heuristics, the *Cost-Based* neighbourhood heuristic is intensifying, as it focuses search effort on high impact variables. It is interesting to compare the *Cost-Based* neighbourhood against the other neighbourhood heuristics, which are inherently diversifying. The diversifying neighbourhoods are slower to make improvements than the *Cost-Based* neighbourhood, as is clearly evident in the results on 40x40 problem set in Figure 6.4.

The results of the pure neighbourhood experiment demonstrate the need for algorithm control. The best performing neighbourhood heuristics vary significantly across time limits and problem sets. The aim of our work is to achieve the best performance at any time limit across any problem set. A robust control method will exhibit performance that is not significantly worse than the best pure method at any time limit on any problem set. In the next section, we perform experiments that explore the application of control methods to see if this can be achieved.

6.8 Combined Neighbourhood LNS

It seems reasonable that with several neighbourhoods to choose from, overall performance can be boosted by applying machine learning to choose the best performing neighbourhood heuristics. Simply alternating, often randomly, between different neighbourhood heuristics appears to be a promising approach and has been used extensively in much of the work on LNS [22, 90]. In this section we explore this approach and extend it with the application of low knowledge learning methods to determine the best performing heuristics.

6.8.1 Control Methods

We explore two learning methodologies here. The first method, *adaptive runtime*, allocates running time based on the performance, over time, of each neighbourhood heuristic. We investigate the use of allocating both a fixed amount of time as well as increasing the amount of time allocated after each iteration. The second method, *adaptive probability*, selects a neighbourhood heuristic to run for a short time. We evaluate the effectiveness of selection using either performance biased probabilities or uniform probability.

6.8.1.1 Adaptive Runtime

The *adaptive runtime* method allocates runtime based on past performance. This is the same switching approach used in Section 5.6.1 in Chapter 5. At each iteration i of the control algorithm, t_i seconds are divided among each neighbourhood heuristic. In the adaptive runtime procedure, the weight $w_i(a)$ determines the proportion of time to allocate to the neighbourhood heuristic a at iteration i. Let $t_i(a) = t_i \times w_i(a)$ represent the time allocation for neighbourhood heuristic a at iteration i.

Weights are updated after each iteration using the formula

$$w_{i+1}(a) = norm(p_i(a)) \times \alpha + w_i(a) \times (1 - \alpha) \tag{6.10}$$

where α is the learning rate and $norm(p_i(a))$ is the normalized performance over time realized by applying neighbourhood heuristic a in the last iteration. In the case that no improvement is found by any neighbourhood heuristic, $norm(p_i(a)) = 0$, otherwise normalized performance is defined as

$$norm(p_i(a)) = \frac{p_i(a)}{\sum_{k \in A} p_i(k)} \tag{6.11}$$

where $p_i(a) = imp_i(a)/t_i(a)$ is the improvement over time, $imp_i(a)$ is the improvement in solution quality achieved by neighbourhood heuristic a at iteration i, and A is the set of all neighbourhood heuristics. Normalization is applied so that performance values for all neighbourhood heuristics sum to 1. By normalizing performance, algorithms are compared on their relative

strength in the current iteration. The learning rate α can be any value from 0..1. Higher values of α update the weight values more rapidly. We use a learning rate of $\alpha = 0.5$.

We explore two variants of this approach. The first *AdaptR-double* doubles the amount of time t_i available at each iteration, starting from an initial value of 1 second. The second approach *AdaptR-static* has a constant time of $t_i = 10$ seconds. All weights are initialized to equal values. The adaptive runtime methods adjust the failure limit after 10 iterations with no improvements, as described in Section 6.3.4.

6.8.1.2 Adaptive Probability

The second method, which is applied for the first time in this chapter, is *adaptive probability*. Instead of allocating longer time slices to the neighbourhood heuristics that perform well, we increase the likelihood of running them. In this context, the normalized value of the weight $w(a)$ is the probability of selecting neighbourhood heuristic a as the next heuristic to run. In each iteration, a neighbourhood heuristic is selected and applied for one time slice. The weight for the neighbourhood heuristic is then updated based on the observed performance.

Since some neighbourhood heuristics may not be run as frequently as others, a different weight updating scheme is employed that takes into account the number of times each heuristic has been run. To implement this weight updating scheme, we define a step dependent discounted average. Let i be an index representing the i^{th} time that a neighbourhood heuristic a has been applied on a problem instance. Let $imp_i(a)$ be the improvement made the i^{th} time the neighbourhood heuristic a was applied.[13] A discount is applied to create a bias toward more recent improvements. That is, for each improvement $imp_i(a)$ we compute a discount of

$$d_i(a) = \frac{1}{(runs(a) - i) + 1} \tag{6.12}$$

[13]Since all heuristics are run for the same duration, we do not compute the performance over time value $p = imp_i(a)/t_{i,a}$.

where $runs(a)$ is the number of times neighbourhood heuristic a has been applied so far. The weight $w(a)$ for a neighbourhood heuristic a is then defined by the mean discounted improvement

$$w(a) = \frac{1}{runs(a)} \sum_{i \in (1..runs(a))} imp_i(a) \times d_i(a) \tag{6.13}$$

To infer the probability of selecting a neighbourhood heuristic, weight values are normalized so that the sum of all weights is 1. These weights are used with a biased roulette wheel to select the next heuristic to apply.

In this chapter we refer to this method as *AdaptP*. We also include a method, *RandP*, which operates in the same manner but does not alter the weights, hence with uniform probability it randomly chooses a neighbourhood heuristic to apply.

6.8.1.3 Comparison of Weight Learning Systems

In this section, we describe the differences between the *AdaptR* and *AdaptP* weight learning mechanisms. *AdaptR* employs normalization and is more aggressive in how fast it 'forgets' past performance. Meanwhile, *AdaptP* does not use normalization and the influence of past performance does not decay as quickly.

Normalization allows the comparison of the relative strength of each algorithm over time. For example, consider the situation where algorithm A makes a very large improvement the first time it is applied, but then makes no further improvements on successive applications. Meanwhile, algorithm B makes a modest improvement every time it is applied. Without normalization, algorithm A would have a higher weight value over time despite only making an improvement on the first application. With normalization, algorithm B would more rapidly gain a higher weight value, since on every application except the first it has outperformed A, albeit with smaller improvements.

It is not evident whether it is best to compare relative or absolute performance measures of each algorithm. While relative performance can be

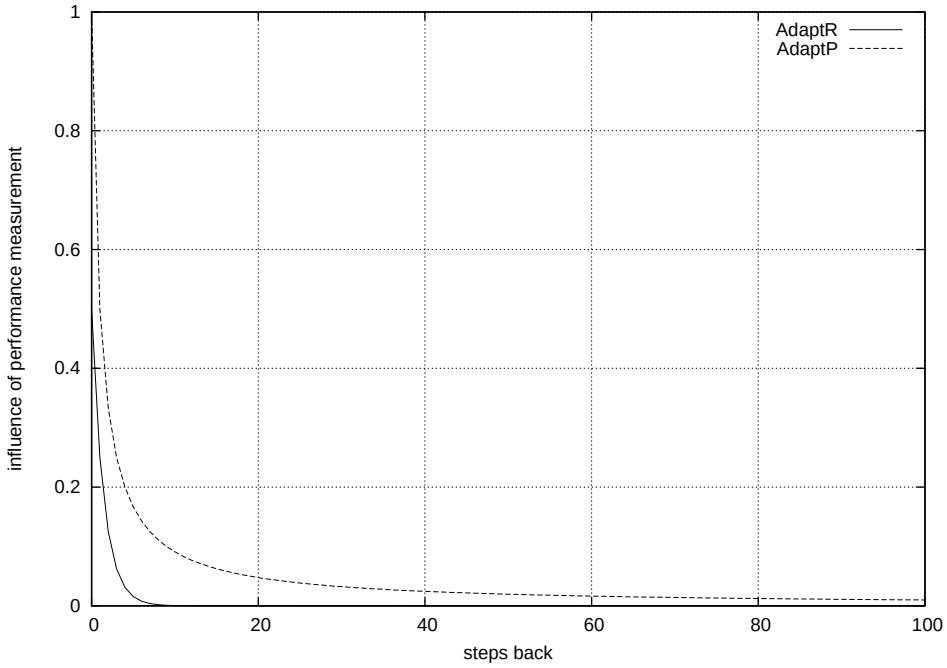


Figure 6.6: Weight decay of *AdaptP* and *AdaptR* mechanisms.

determined when applying *AdaptR*, since all algorithms are run in each iteration, when applying *AdaptP* the algorithms are not always run as frequently as each other. Some form of normalization could still be applied to *AdaptP*, e.g., comparing relative performance in different time intervals of the run, however, we do not explore this in this chapter.

We now turn to the mechanism that regulates how quickly a weight value adapts. Let imp_i represent the improvement by heuristic a the i^{th} time the heuristic was applied and let $runs$ be the total number of times the heuristic has been applied. The influence of imp_i on the current weight value for *AdaptR* is

$$\alpha^{(runs-i)} \tag{6.14}$$

while the influence of imp_i on the current weight value for *AdaptP* is

$$\frac{1}{(runs-i)+1} \tag{6.15}$$

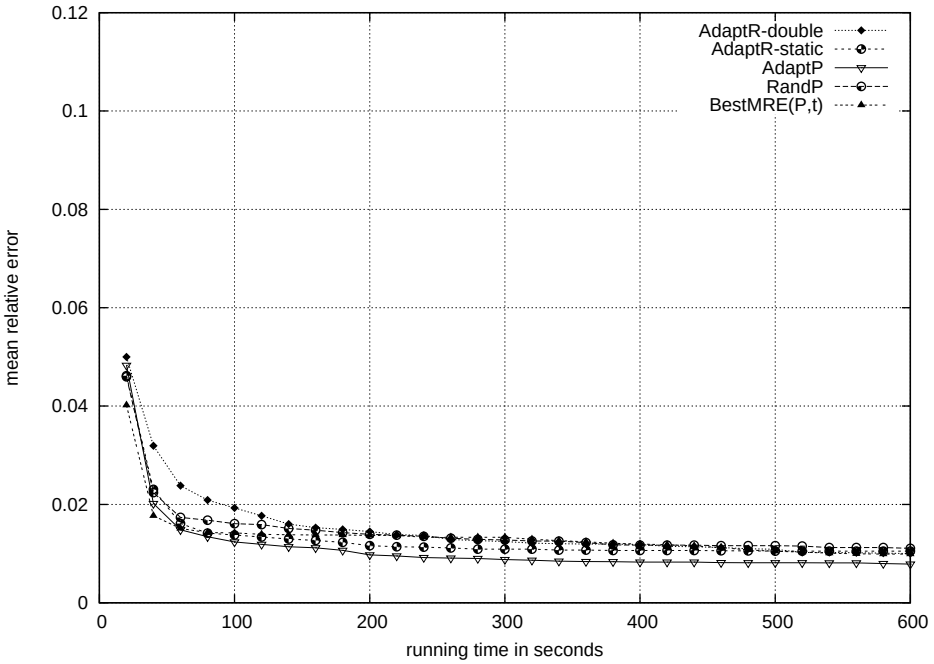
As the number of applications of a heuristic grows, *AdaptR* decays at an exponential rate while *AdaptP* decays at a rate which is a reciprocal of the number of applications. In Figure 6.6 we show a graph of the impact of past performance measures on the current weight value for *AdaptR* (with $\alpha = 0.5$) and *AdaptP*. On the X axis, the number of steps back in time is shown, with 0 indicating the current step. On the Y axis, we show the impact that the measurement has on the current weight value. We see that *AdaptR* has a more aggressive decay rate, while *AdaptP* decays at a far slower rate. Although the value of α can be adjusted to slow the decay rate of *AdaptR*, the decay rate remains exponential whereas the decay rate of *AdaptP* is a reciprocal.

The impact of the weight decay mechanism will likely have a significant impact on the performance of each weight system. The *AdaptR* weight mechanism will adapt faster as performance changes over time, but may tend to be 'greedy' and focus effort on the most recent best performers. Conversely, the *AdaptP* mechanism will retain past performance knowledge for a longer time, but may be slower to react to changes. It is unclear which strategy is better, and likely depends on the characteristics of the algorithms and problems. A detailed study of the impact of these mechanisms is an interesting direction for future work.

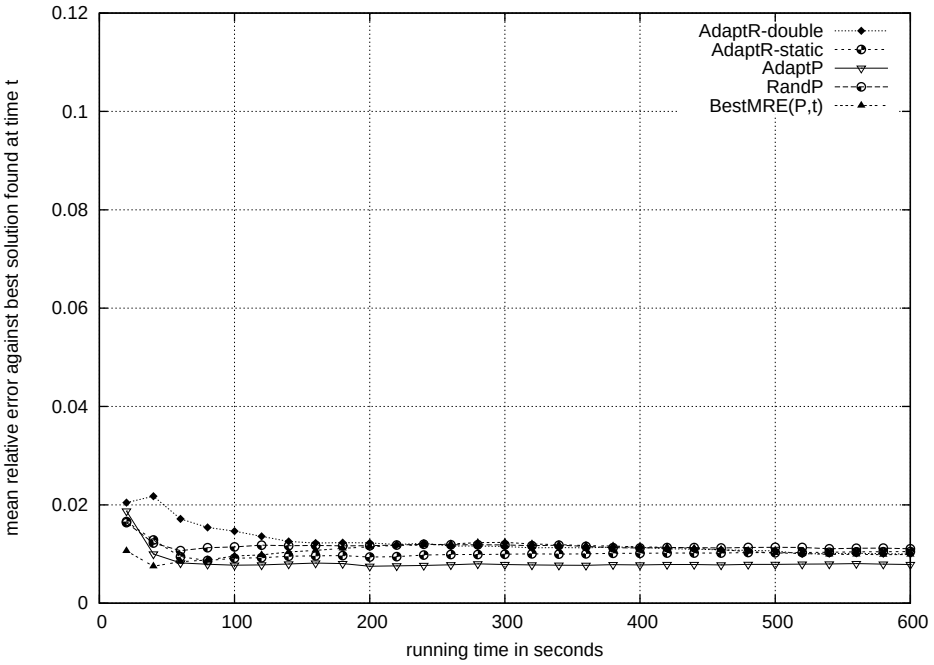
6.8.2 Experiments

We evaluate the combined neighbourhood heuristics using the same time limits and test sets that were applied to the pure neighbourhoods in Section 6.7. We take the same parameter settings for each pure neighbourhood but combine them using the methods described in Section 6.8.1.

In Figure 6.7 and Figure 6.8 we display the mean relative error (MRE) results of running the combined neighbourhoods. For comparison, we also show *BestMRE(P,t)* which is the best performing pure neighbourhood at each time limit t. Recall that *BestMRE(P,t)* is the pure neighbourhood with the minimum MRE(t) for each time point, therefore each point may represent the results of a different pure neighbourhood. We note that *BestMRE(P,t)* is

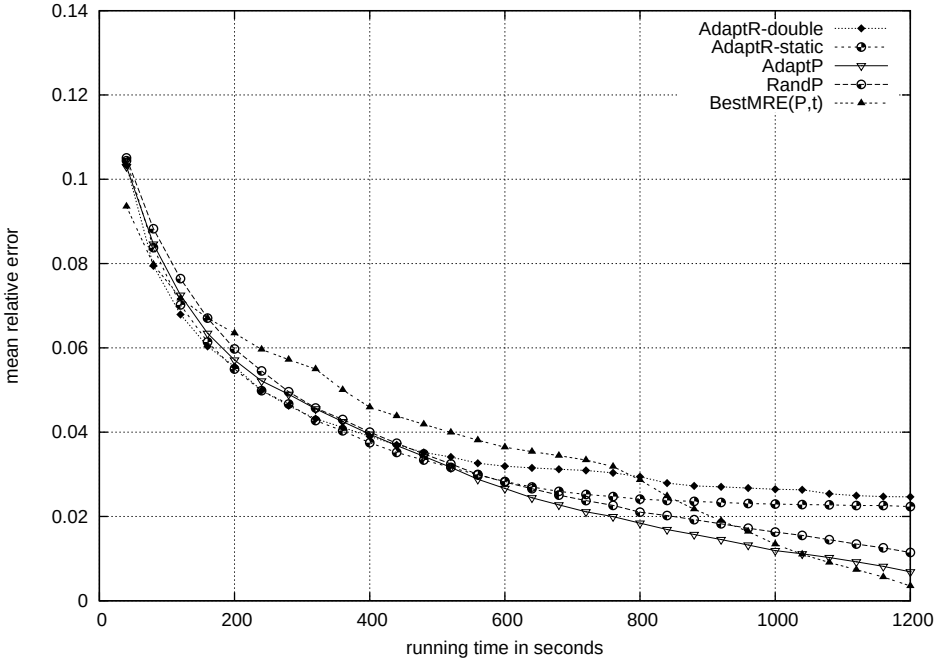


(a) Mean relative error against the best solution over all time limits.

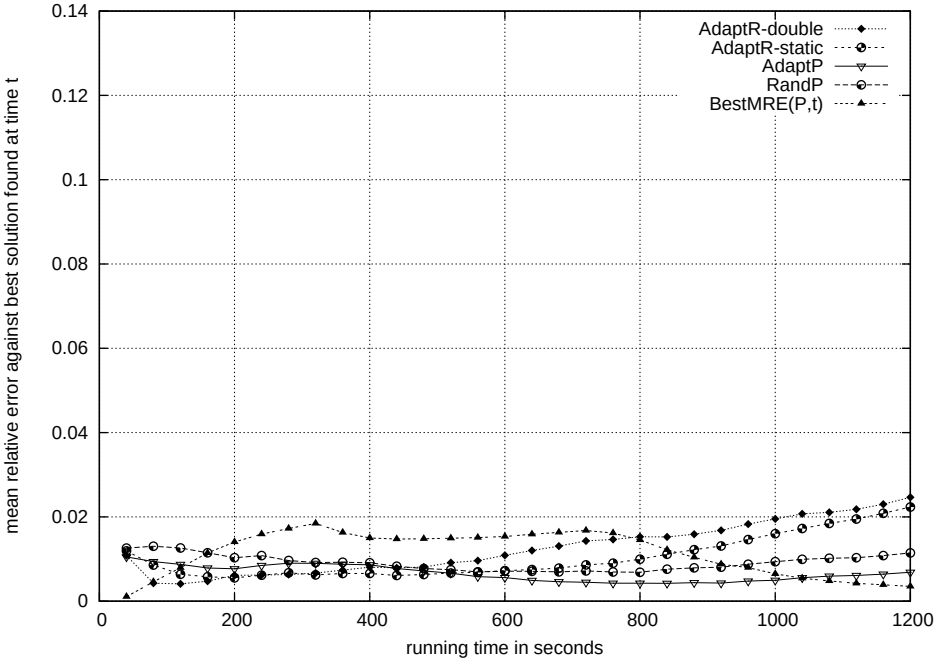


(b) Mean relative error (MRE(t)) against the best solution found at time t.

Figure 6.7: Combined neighbourhood heuristics run on the 20x20 problem set.

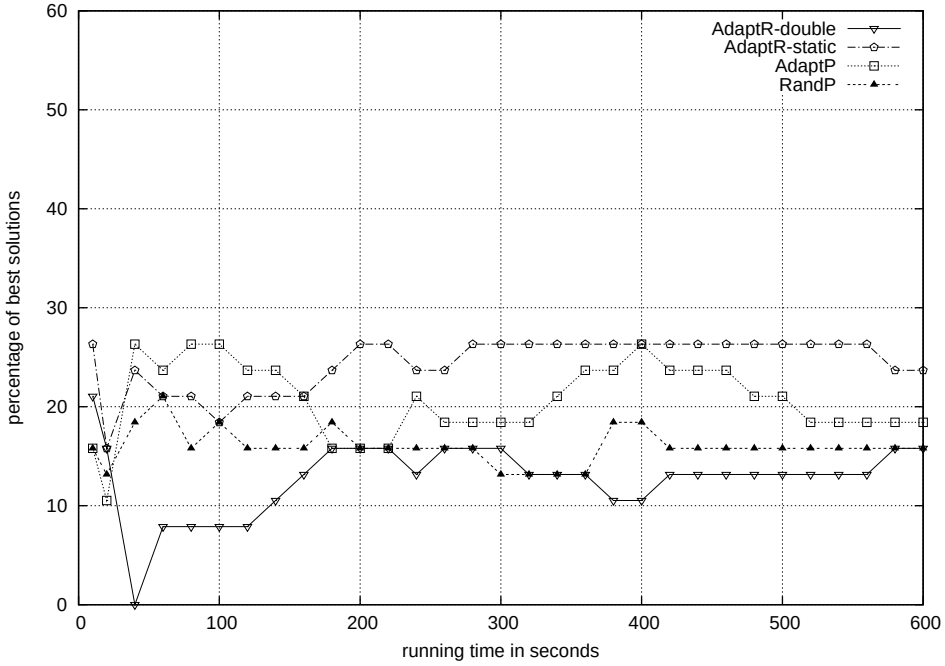


(a) Mean relative error against the best solution over all time limits.

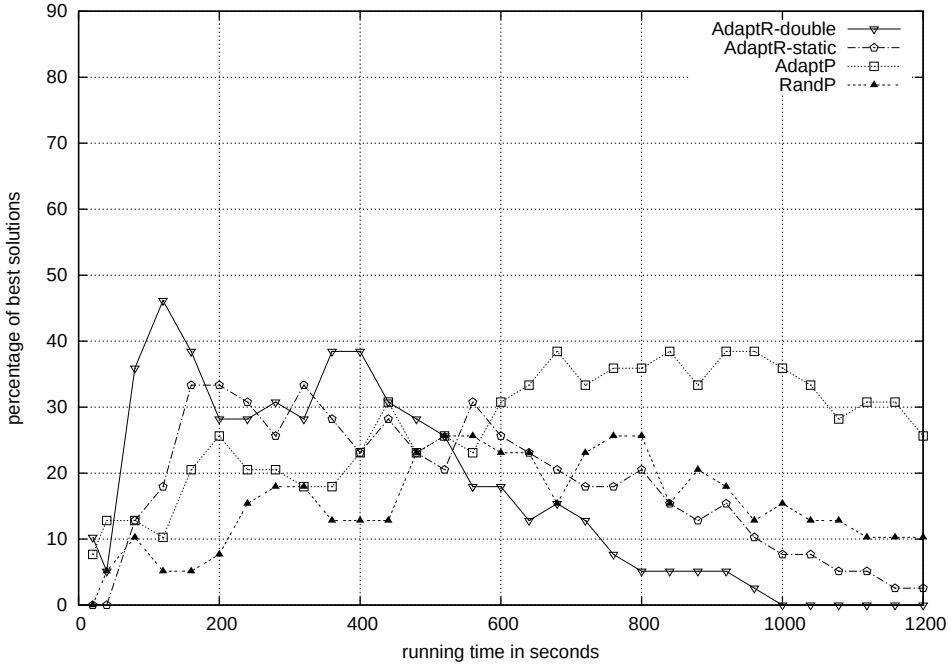


(b) Mean relative error (MRE(t)) against the best solution found at time t.

Figure 6.8: Combined neighbourhood heuristics run on the 40x40 problem set.



(a) 20x20 problem set.



(b) 40x40 problem set.

Figure 6.9: Best solutions found by combined neighbourhood heuristics.

the best performance that can be achieved by applying a single pure method.

On the 20x20 problem set all of the combined methods perform well. As Figure 6.7(b) shows, after a minute of run time, all of the combined methods (except *AdaptR-double*) find solutions within 1% of the best known solution at each time point t. *AdaptR-double* appears slower to converge but eventually joins the other combined methods after two minutes. Although the gap is slight, *AdaptP* appears to perform better than the other methods at all time limits and outperforms *BestMRE(P,t)* after only a minute. *AdaptR-static* also slightly outperforms *BestMRE(P,t)* but does not perform as well as *AdaptP*. *RandP* appears to perform as well as *AdaptR-double* and *BestMRE(P,t)* for times after 200 seconds. Recall that *BestMRE(P,t)* is the best pure algorithm at each time point. In this problem set, *BestMRE(P,t)* is *Time Window* until 300 seconds, after which it is *Resource Load*.

On the 40x40 problem set the gap between *BestMRE(P,t)* and the combined neighbourhoods grows. From 200 seconds to 700 seconds, all combined neighbourhoods perform better than *BestMRE(P,t)*. *BestMRE(P,t)* performs better at the very start and end of the runs. On this set, *BestMRE(P,t)* is *Cost-Based* from 0 to 260, then *Random* until 780 seconds, then *Time Window* for the remaining time points. For the most part there is little difference between the combined methods, until after 500 to 600 seconds, after which the gap between them starts to grow. *AdaptR-double* performs the worst at this point, with *AdaptR-static* following closely. *RandP* maintains performance that is only slightly worse than *AdaptP* across most of the run. However, *AdaptP* appears to perform better than the other combined methods except for at the very start, where *AdaptR-double* performs best.

Figure 6.9 presents the number of best solutions found by the combined methods. On the 20x20 problem set, after a short startup time, all of the combined methods find a reasonable number of the best solutions, ranging 15 to 25 percent. *AdaptR-static* and *AdaptP* find slightly more better solutions than the other methods. On the 40x40 problem set, *AdaptR-double* starts off finding nearly 50% of the best solutions, but performs worse later in the run. All of the combined methods do well until 600 seconds, after which all of them, with the exception of *AdaptP*, find fewer of the best solutions. *AdaptP*

continues to find over 25% of the best solutions at the end of the run.

To analyse the differences in performance over time, we repeat the statistical analysis described in Section 6.5.5 with the combined methods. We performed this analysis procedure twice: once to compare performance against the pure methods and once to compare performance among the combined methods. Recall that a comparison is made for each of the 30 time points. Each comparison has $n = 40$ samples, the result of running each algorithm on 40 problem instances. We refer to this as the *30 time point analysis*. We also perform an *all points analysis* where we combine the results from each of the 30 time points into a single data set where each method has $n = 40 \times 30 = 1200$ samples. When comparing against the pure methods P there are 7 factors: $|C| = 4$ combined neighbourhood heuristics, *BestMRE(P,T)*, *BestMRE(P,t)* and *BestInstance(P,t)*. When comparing among combined methods C there are 5 factors: $|C| = 4$ combined neighbourhood heuristics and *BestInstance(C,t)*. Although the comparison among the combined methods includes *BestMRE(C,T)* and *BestMRE(C,t)*, this does not introduce 2 additional factors since these results are derived from the analysis of the 4 combined methods.

First we look at the difference over the 30 time points. In Table 6.7(a) we show the difference between the combined methods and *BestMRE(P,T)*, the pure method with the best MRE at the last time point. We see that, on all 30 time points on the 20x20 problem set, the combined methods never perform worse and sometimes perform better than *BestMRE(P,T)*. The best performing method is *AdaptP* which outperforms *BestMRE(P,T)* on 10 time points, while the other methods outperform on only a few time points. On the 40x40 set, three of the combined methods perform worse at some time points, however *AdaptP* does not perform significantly worse at any time point. All of the combined methods perform better on more than half of the time points in the 40x40 set. In Table 6.7(b) we compare the combined methods against *BestMRE(P,t)*, the pure method with the best MRE at each time point. On the 20x20 set, only *AdaptR-double* performs worse at two time points, but no combined method performs better than *BestMRE(P,t)*. On the 40x40 set, *AdaptR-static* and *AdaptR-double* perform worse on 8 time points, and

Table 6.7: Count of significant differences over *30 time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics.

(a) *BestMRE(P,T)* compared to combined neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
AdaptR-d	0	0	1	29	8	0	16	6
AdaptR-s	0	0	5	25	7	0	18	5
AdaptP	0	0	10	20	0	0	20	10
RandP	0	0	2	28	3	0	19	8

(b) *BestMRE(P,t)* compared to combined neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
AdaptR-d	2	0	0	28	8	0	4	18
AdaptR-s	0	0	0	30	8	0	15	7
AdaptP	0	0	0	30	0	0	14	16
RandP	0	0	0	30	4	0	10	16

(c) *BestInstance(P,t)* compared to combined neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
AdaptR-d	6	0	0	24	9	0	2	19
AdaptR-s	0	0	0	30	8	0	3	19
AdaptP	0	0	0	30	0	0	6	24
RandP	0	0	0	30	5	0	0	25

W/S/B/? denotes significantly Worse, Same, significantly Better and ? no evidence of a significant difference.

RandP performs worse on 4 time points. *AdaptP* does not perform worse on any time limit. All of the combined methods perform better at some time points, with *AdaptR-static* and *AdaptP* performing better on 15 and 14 time points respectively. Table 6.7(c) shows a comparison against the best solution found on each instance by any pure method, *BestInstance(P,t)*. On the 20x20 set, there is no evidence of a difference between the combined methods and *BestInstance(P,t)*, with the exception of *AdaptR-double* which performs worse at 6 time points. On the 40x40 set, the results are more varied. *AdaptR-double* and *AdaptR-static* perform worse on 9 and 8 points respectively, and better on 2 and 3 time points. *RandP* performs worse at 5 time points, and never better. Finally, *AdaptP* performs better on 6 time points and never performs worse than *BestInstance(P,t)*.

To evaluate the mean difference in performance over the entire run, we look at the all points comparison between the combined and pure methods. Table 6.8(a) shows the mean difference on the 20x20 problem set. We see that there is evidence that all combined methods perform better than *BestMRE(P,T)*. Comparing *BestMRE(P,t)*, there is evidence of only one combined method, *AdaptP*, outperforming *BestMRE(P,t)* and no evidence of a difference in performance against the other combined methods. Comparing against *BestInstance(P,t)*, we see that all combined methods perform worse with the exception of *AdaptP*, where there is no evidence of difference. Table 6.8(b) shows the all points comparison on the 40x40 problem set. Here we see an identical picture in terms of the significant differences when comparing both *BestMRE* measures, although the relative difference in MRE is larger. When we compare against *BestInstance(P,t)* we see that there is no evidence of a difference in performance against any of the combined methods on the 40x40 set. This concludes the comparison between the combined methods and the pure methods.

We now show a comparison of the combined methods against each other. The comparison between combined methods and *BestMRE(C,T)* is shown in Table 6.9(a). The method with the best MRE at time T was *AdaptP* on both problem sets. On the 20x20 set, *AdaptR-double* performs worse on 4 time points, while for all other time points and combined methods, there is no

Table 6.8: Mean difference over *all time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics.

(a) 20x20 problem set.

	BestMRE(P,T)	*BestMRE(P,t)*	*BestInstance(P,t)*
AdaptR-double	**-0.00210**	0.00170	**0.00571**
	(0.00051)	(0.01133)	(0.00000)
AdaptR-static	**-0.00428**	-0.00048	**0.00352**
	(0.00000)	(0.95999)	(0.00000)
AdaptP	**-0.00621**	**-0.00242**	0.00159
	(0.00000)	(0.00003)	(0.02354)
RandP	**-0.00285**	0.00095	**0.00496**
	(0.00000)	(0.47121)	(0.00000)

(b) 40x40 problem set.

	BestMRE(P,T)	*BestMRE(P,t)*	*BestInstance(P,t)*
AdaptR-double	**-0.00840**	0.00108	0.00300
	(0.00000)	(0.94447)	(0.06012)
AdaptR-static	**-0.01055**	-0.00107	0.00084
	(0.00000)	(0.94659)	(0.98399)
AdaptP	**-0.01438**	**-0.00490**	-0.00298
	(0.00000)	(0.00005)	(0.06269)
RandP	**-0.01179**	-0.00231	-0.00039
	(0.00000)	(0.28489)	(0.99978)

Significant differences ($p \leq 0.005$) are **shown in bold** and p values are shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

Table 6.9: Count of significant differences over *30 time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of combined neighbourhood heuristics.

(a) *BestMRE(C,T)* compared to combined neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
AdaptR-d	4	0	0	26	15	0	0	15
AdaptR-s	0	0	0	30	10	0	0	20
AdaptP	0	30	0	0	0	30	0	0
RandP	0	0	0	30	0	0	0	30

(b) *BestMRE(C,t)* compared to combined neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
AdaptR-d	4	0	0	26	15	4	0	11
AdaptR-s	0	0	0	30	10	7	0	13
AdaptP	0	29	0	1	0	19	0	11
RandP	0	1	0	29	0	0	0	30

(c) *BestInstance(C,t)* compared to combined neighborhood heuristics.

	20x20				40x40			
	W	S	B	?	W	S	B	?
AdaptR-d	30	0	0	0	22	0	0	8
AdaptR-s	29	0	0	1	17	0	0	13
AdaptP	25	0	0	5	7	0	0	23
RandP	29	0	0	1	17	0	0	13

W/S/B/? denotes significantly Worse, Same, significantly Better and ? no evidence of a significant difference.

evidence of a difference against *BestMRE(C,T)*. On the 40x40 set, *AdaptR-double* and *AdaptR-static* perform worse on 15 and 10 points respectively, while again we see no evidence of difference for any other time limits. In particular, we see that there is no evidence of a difference in performance between *BestMRE(C,T)* and *RandP* on either problem set. Table 6.9(b) shows the comparison against *BestMRE(C,t)*, the best combined method at each time point. In the 20x20 set, we see that *AdaptP* was the best method on 29 time points, while *RandP* was best on one time point. On the 40x40 set, *AdaptP* was best on 19 time points, with *AdaptR-double* and *AdaptR-static* performing best on 4 and 7 points respectively. In terms of combined methods performing worse, we see exactly the same results as shown in Table 6.9(a). Since we are comparing combined methods against the best combined method at each time point, no method can perform significantly better than *BestMRE(C,t)*. Table 6.9(c) compares the combined methods against *BestInstance(C,t)*, the best solution found by any combined method. On the 20x20 set, we see that all of the combined methods perform worse than *BestInstance(C,t)* at most of the time points. This indicates that the best solutions are not always found by the same combined method at each point. On the 40x40 set, the performance is not as varied. *AdaptR-double* performs worse at 22 time points, and *AdaptR-static* and *RandP* perform worse at 17 time points. *AdaptP* performs worse on only 7 time points. As with *BestMRE(C,t)*, it is impossible for a combined method to perform better than *BestInstance(C,t)*.

Tables 6.10(a) and 6.10(b) show the all time points comparison among the combined neighbourhoods. On the 20x20 set, *AdaptP* performs better than all other neighbourhood heuristics. *AdaptR-static* is second best, but only performs significantly better than *AdaptR-double*. *RandP* is not significantly better than any method, but not significantly worse than any method except *AdaptP*. Finally, *AdaptR-double* has the worst performance. On the 40x40 set *AdaptP* is again the best performer, however, it does not perform significantly better than *RandP*. *RandP* is the second best performer, but only significantly better than *AdaptR-double*. *AdaptR-static* is only significantly worse than *AdaptP*, but not better than any method. Once again,

Table 6.10: Mean difference over *all time points* for combined neighborhood heuristics.

(a) 20x20 problem set.

	AdaptR-double	AdaptR-static	AdaptP	RandP
AdaptR-double		**0.00218**	**0.00412**	0.00075
		(0.00002)	(0.00000)	(0.37833)
AdaptR-static	**-0.00218**		**0.00193**	-0.00144
	(0.00002)		(0.00022)	(0.01152)
AdaptP	**-0.00412**	**-0.00193**		**-0.00337**
	(0.00000)	(0.00022)		(0.00000)
RandP	-0.00075	0.00144	**0.00337**	
	(0.37833)	(0.01152)	(0.00000)	

(b) 40x40 problem set.

	AdaptR-double	AdaptR-static	AdaptP	RandP
AdaptR-double		0.00216	**0.00598**	**0.00339**
		(0.10926)	(0.00000)	(0.00225)
AdaptR-static	-0.00216		**0.00383**	0.00123
	(0.10926)		(0.00038)	(0.56922)
AdaptP	**-0.00598**	**-0.00383**		-0.00259
	(0.00000)	(0.00038)		(0.03423)
RandP	**-0.00339**	-0.00123	0.00259	
	(0.00225)	(0.56922)	(0.03423)	

Significant differences ($p \leq 0.005$) are **shown in bold** and p values are shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

AdaptR-double performs worst. We note that while there is evidence of a significant difference, the observed difference is less than 0.5% in all cases.

We show an all points comparison of combined methods against the *BestMRE* and *BestInstance(C,t)* in Tables 6.11(a) and 6.11(b). On the 20x20 set, all methods except for *AdaptP* perform worse than *BestMRE*. In the 30 point comparison against *BestMRE*, we did not see evidence that these methods performed worse. This is perhaps due to the increase in sample size in the all points comparison.[14] However, we note that despite the evidence of a significant difference, the observed difference in MRE is relatively small, less than 0.5%. On the 20x20 set, all methods perform worse than *BestInstance(C,t)*, confirming again the variance of best performance on the 20x20 set. On the 40x40 set, we see a similar picture comparing against *BestMRE*, although this time *RandP* is not significantly worse than *BestMRE(C,T)* and while it is significantly worse than *BestMRE(C,t)*, there is only barely evidence of this difference ($p = 0.004 < 0.005$). We see that all methods perform worse than *BestInstance(C,t)* when the performance is compared over all time points.

6.8.3 Discussion

In combining different neighbourhood heuristics we hope to achieve two things: robustness and synergy. Robustness refers to the ability to achieve strong performance, no matter what problem set or processing time is available. Robust solving is a shortcoming of many optimization techniques, and to achieve robust solving implies that we can reduce the requirement for human effort and expertise in applying these methods. For instance, with a number of pure neighbourhood heuristics to choose from, each of which performs differently depending on the time available, the problem set and often the problem instance, we would need human expertise and experimentation to find the best one. A robust technique obtains good performance across all problem sets and time limits. Synergy refers to a combination of neighbour-

[14]Recall that for each method there are 40 samples at each point in the 30 point comparison, while there are $30 \times 40 = 1200$ samples in the all points comparison.

Table 6.11: Mean difference over *all time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of combined neighbourhood heuristics.

(a) 20x20 problem set.

	BestMRE(C,T)	*BestMRE(C,t)*	*BestInstance(C,t)*
AdaptR-double	**0.00412**	**0.00419**	**0.01081**
	(0.00000)	(0.00000)	(0.00000)
AdaptR-static	**0.00193**	**0.00201**	**0.00863**
	(0.00029)	(0.00013)	(0.00000)
AdaptP	0.00000	0.00008	**0.00670**
	(1.00000)	(1.00000)	(0.00000)
RandP	**0.00337**	**0.00345**	**0.01006**
	(0.00000)	(0.00000)	(0.00000)

(b) 40x40 problem set.

	BestMRE(C,T)	*BestMRE(C,t)*	*BestInstance(C,t)*
AdaptR-double	**0.00598**	**0.00697**	**0.01073**
	(0.00000)	(0.00000)	(0.00000)
AdaptR-static	**0.00383**	**0.00481**	**0.00858**
	(0.00165)	(0.00002)	(0.00000)
AdaptP	-0.00000	0.00098	**0.00475**
	(1.00000)	(0.95173)	(0.00002)
RandP	0.00259	**0.00358**	**0.00734**
	(0.10790)	(0.00450)	(0.00000)

hood heuristics outperforming a single neighbourhood heuristic. In Chapter 5 we observed a synergistic effect when combining different search algorithms.

On the both problem sets, we clearly achieve robustness. The distance of each combined method to the best known solution at each time, MRE(t) in Figures 6.7(b) and 6.8(b), is relatively similar across all times unlike the MRE(t) values of the pure methods, shown in 6.3(b) and 6.4(b). While some pure methods perform better at times, they perform worse at others and the variance in MRE(t) is high. The combined methods with the best performance do not exhibit this variance in behaviour and stay within 1% of the best known solution at time t for all time points after 60 seconds.

In terms of synergy, the combined methods produce MRE values lower than the pure methods. While this difference is not significant on the 20x20 set, it is significant on the 40x40 problem set, where combined methods outperform the best pure method at many time points. The best performing combined method *AdaptP* has performance better or equal to the best pure method at all time points. The version without learning, *RandP*, also performs well, but lags behind *AdaptP* in solution quality. The adaptive runtime methods, *AdaptR-static* and *AdaptR-double*, perform well but start to degrade over time on the large problem instances. Either the policy of committing to run each neighbourhood every iteration or the weight learning scheme is to blame.

We believe that as the variance in performance of the available neighbourhoods grows, we will see the gap between the learning and non-learning methods grow. Since the four neighbourhood heuristics that were applied all perform relatively well, good performance can be achieved simply by alternating between them. In the next section, we investigate the effects of unknown problem instances that vary in size and apply neighbourhood heuristics without performing parameter tuning.

6.9 Evaluation on Taillard Benchmarks

We performed experiments on a subset of Taillard's job shop benchmarks [104]. The problem instances selected are known as TA-11 to TA-80 and

range in size from 20 to 100 jobs and 15 to 20 machines. In this experiment, we did not tune the neighbourhood heuristics for this problem set. Instead we used the same parameter settings that were applied to the 20x20 problem set. This provides insight on the performance of the control methods when applied to problems from an unknown problem domain.

For those interested in the specific results on each instance, for comparison against the state-of-the-art, we attach detailed results on these benchmarks in Appendix A.

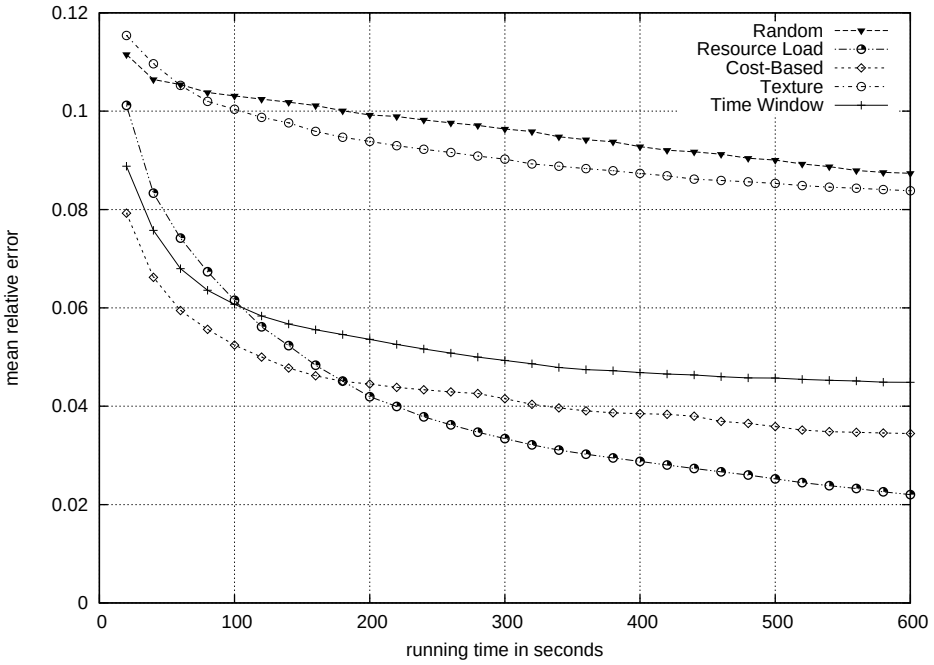
6.9.1 Experiments

In Figure 6.10, we show the results, in terms of MRE,[15] of running the pure neighbourhood heuristics and the *Texture* algorithm. The difference in performance between the heuristics is greater here than in the previous experiments. In particular, the *Random* heuristic performs far worse than the other heuristics, even worse than *Texture*. For the other pure heuristics the MRE(t) performance, as shown in Figure 6.10(b) varies between 1% and 4%. In terms of best solutions found, *Cost-Based* and *Resource Load* appear to be the strongest performers with a crossover at around 200 seconds, as shown in Figure 6.11(a).

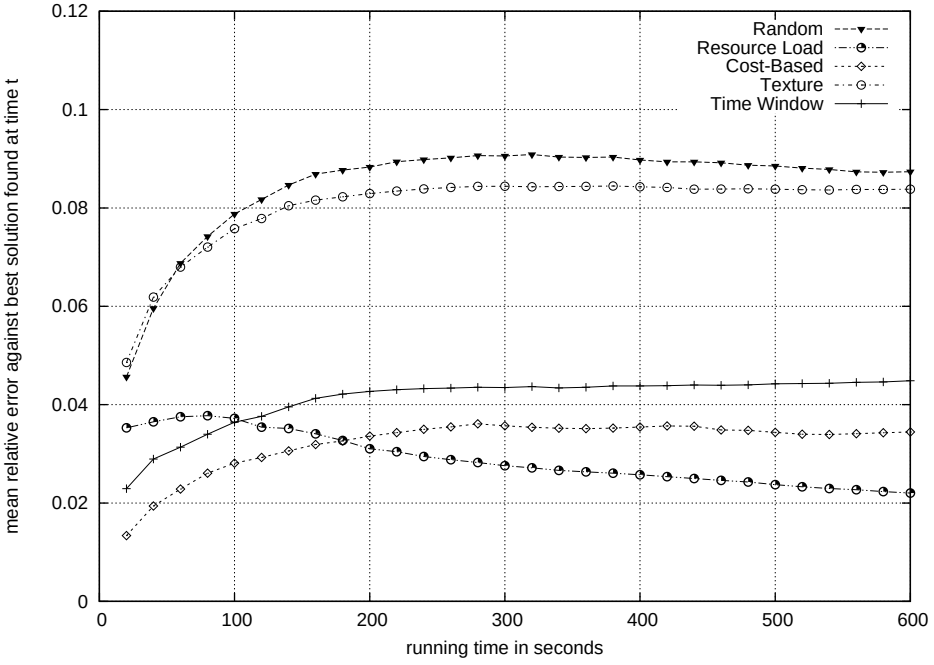
The combined neighbourhood heuristics perform well on this problem set, as shown in Figure 6.12. In terms of MRE, every combined method performs better than the pure neighbourhoods, except at the very start of the runs. *AdaptP* and *AdaptR-static* are the best performers of the combined neighbourhood heuristics, producing the best solutions across all time limits. *RandP* starts off lagging behind the other neighbourhood heuristics, although *AdaptR-double* produces a comparable MRE result after 300 seconds.

Three statistical analyses were performed using the procedure from Section 6.5.5 using the following factors. The analysis of the set of pure neighbourhoods involved 5 factors: the $|P| = 4$ pure neighbourhoods and *BestIn-*

[15] In this section we present MRE against the best solution found during our experiments rather than the best known upper bounds [105]. Our best upper bounds are on average 3% worse. A detailed comparison against the best known solutions is shown in Appendix A.

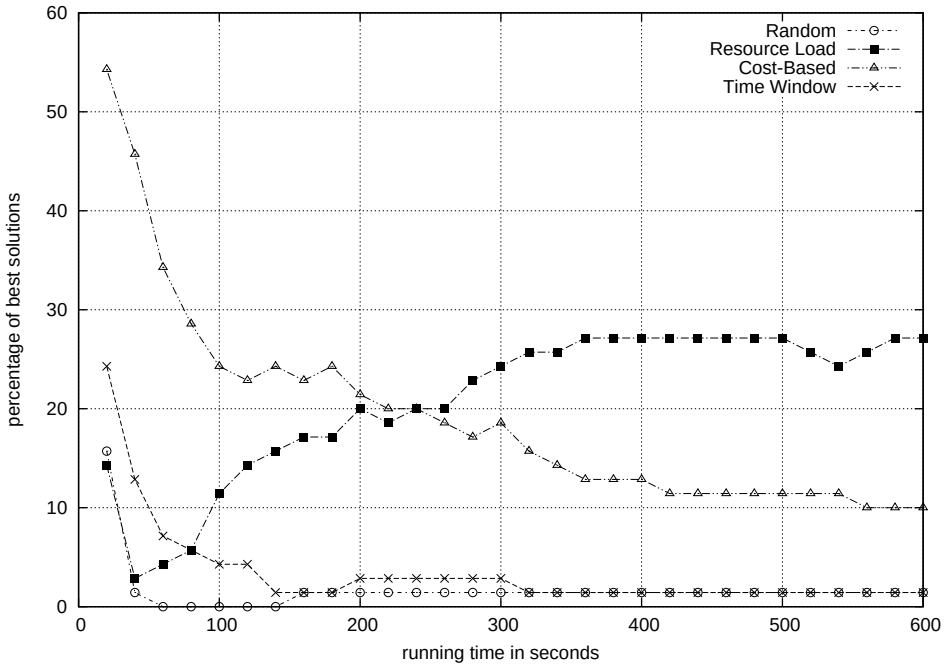


(a) Mean relative error against the best solution over all time limits.

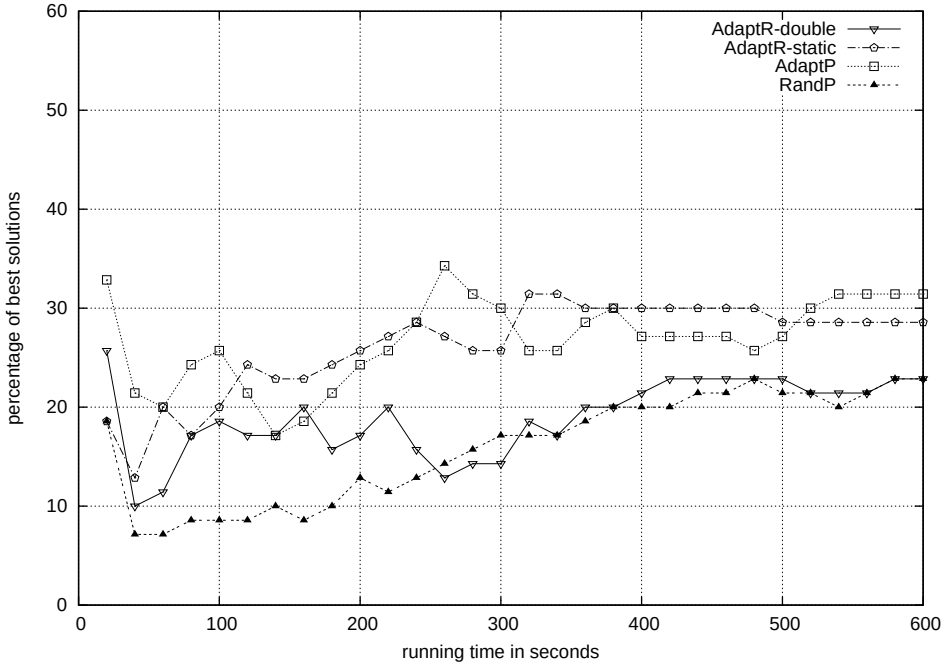


(b) Mean relative error (MRE(t)) against the best solution found at time t.

Figure 6.10: Pure neighbourhood heuristics run on Taillard's problem set.

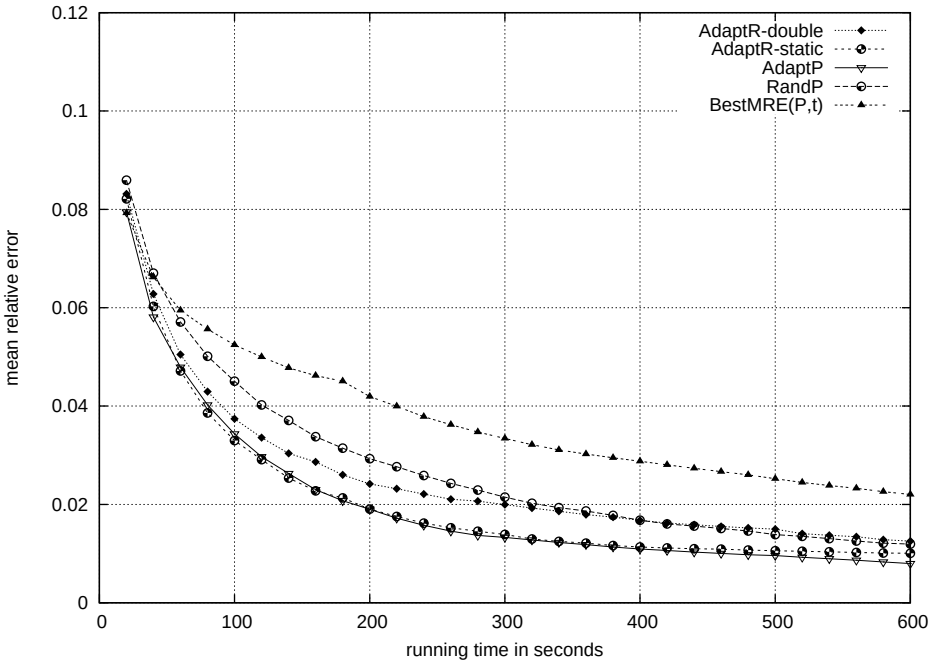


(a) Pure NHs.

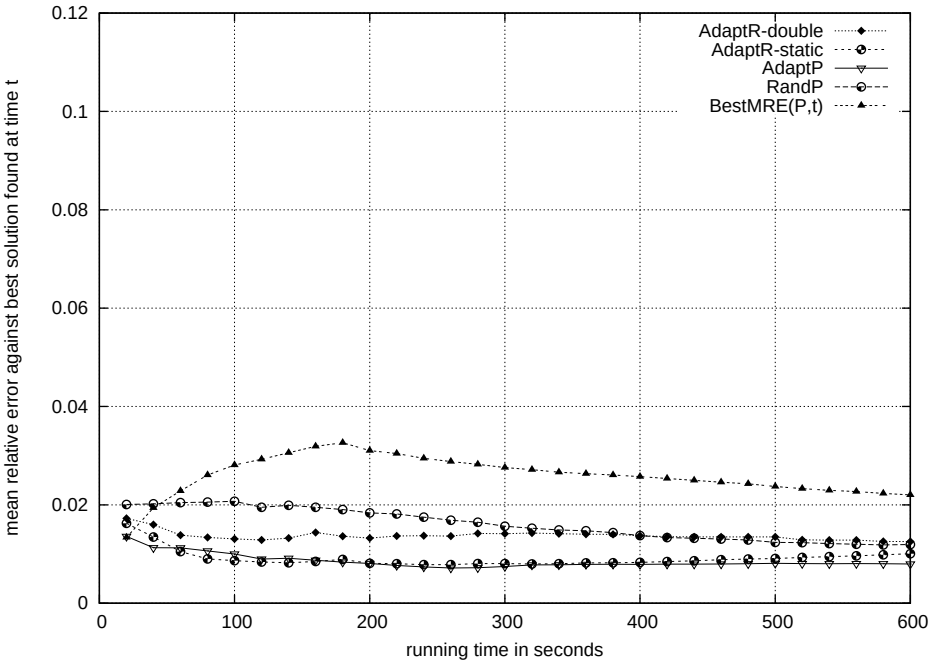


(b) Combined NHs.

Figure 6.11: Best solutions found on Taillard's problem set.



(a) Mean relative error against the best solution over all time limits.



(b) Mean relative error (MRE(t)) against the best solution found at time t.

Figure 6.12: Combined neighbourhood heuristics run on Taillard's problem set.

stance(P,t). The analysis comparing pure and combined neighbourhoods had 7 factors: $|C| = 4$ combined methods and 3 factors for *BestMRE(P,T)*, *BestMRE(P,t)*, and *BestInstance(P,t)*. The analysis comparing the combined methods had 5 factors: $|C| = 4$ and *BestInstance(C,t)*.

In Table 6.12(a) we show the difference between each pure neighbourhood against *BestMRE(P,T)*, the best pure neighbourhood at the last time point T. On Taillard's problem set, *BestMRE(P,T)* was *Resource Load*. We see that *Random* performed worse at 28 points and *Time Window* performed worse at 1 point. No evidence of a difference was found for any other points or methods. Table 6.12(b) shows the difference between each pure neighbourhood against *BestMRE(P,t)*, the best pure neighbourhood at each time point. The best pure neighbourhoods are *Resource Load* and *Cost-Based* which are *BestMRE(P,t)* at 21 and 9 points respectively. *Time Window* remains worse at only 1 time point and *Random* performs worse on all time points. Table 6.12(c) compares against *BestInstance(P,t)*, the best solution found by any pure method. Here we see *Resource Load* is the strongest performer with worse performance at 7 time points. In a distant second place is *Cost-Based*, which is worse at 21 points, followed by *Time Window* at 26 points and then *Random*, which is worse at all points. Table 6.13(a) shows the all points analysis between the pure methods. *Resource Load* and *Cost-Based* are the best performers and are significantly better than the other two pure neighbourhoods. *Time Window* performs significantly better than *Random*, and *Random* performs worse than all other heuristics. Table 6.13(b) shows the comparison against *BestMRE* and *BestInstance(P,t)*. Here we see that even though *Resource Load* and *Cost-Based* perform well against *BestMRE*, they are more than 2% worse than *BestInstance(P,t)*, which is nearly twice the difference in performance that was observed in Section 6.7 where the pure neighbourhood heuristics were tuned for the problem set.

We now turn to the analysis of combined methods compared to the pure methods. Table 6.14(a) shows the difference between combined neighbourhoods and *BestMRE(P,T)*. We see that all combined methods perform better than the best pure neighbourhood at many time points. The best performance is seen by *AdaptP* at 29 time points, followed by *AdaptR-double*

Table 6.12: Count of significant differences over *30 time points* for pure neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics.

(a) *BestMRE(P,T)* compared to pure neighborhood heuristics.

	Taillard			
	W	S	B	?
Random	28	0	0	2
Resource Load	0	30	0	0
Cost-Based	0	0	0	30
Time Window	1	0	0	29

(b) *BestMRE(P,t)* compared to pure neighborhood heuristics.

	Taillard			
	W	S	B	?
Random	30	0	0	0
Resource Load	0	21	0	9
Cost-Based	0	9	0	21
Time Window	1	0	0	29

(c) *BestInstance(P,t)* compared to pure neighborhood heuristics.

	Taillard			
	W	S	B	?
Random	30	0	0	0
Resource Load	7	0	0	23
Cost-Based	21	0	0	9
Time Window	26	0	0	4

W/S/B/? denotes significantly Worse, Same, significantly Better and ? no evidence of a significant difference.

Table 6.13: Mean difference over *all time points* for pure neighborhood heuristics on Taillard's benchmarks.

(a) Among pure neighborhood heuristics

	Random	Resource Load	Cost-Based	Time Window
Random		**0.05587**	**0.05249**	**0.04357**
		(0.00000)	(0.00000)	(0.00000)
Resource Load	**-0.05587**		-0.00338	**-0.01231**
	(0.00000)		(0.07452)	(0.00000)
Cost-Based	**-0.05249**	0.00338		**-0.00893**
	(0.00000)	(0.07452)		(0.00000)
Time Window	**-0.04357**	**0.01231**	**0.00893**	
	(0.00000)	(0.00000)	(0.00000)	

(b) *BestMRE* and *BestInstance* of pure neighborhood heuristics

	BestMRE(P,T)	*BestMRE(P,t)*	*BestInstance(P,t)*
Random	**0.05587**	**0.05879**	**0.07655**
	(0.00000)	(0.00000)	(0.00000)
Resource Load	0.00000	0.00292	**0.02067**
	(1.00000)	(0.22643)	(0.00000)
Cost-Based	0.00338	**0.00630**	**0.02405**
	(0.09619)	(0.00001)	(0.00000)
Time Window	**0.01231**	**0.01522**	**0.03298**
	(0.00000)	(0.00000)	(0.00000)

Significant differences ($p \leq 0.005$) are **shown in bold** and p values are shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

Table 6.14: Count of significant differences over *30 time points* for combined neighborhood heuristics against *BestMRE* and *BestInstance* of pure neighborhood heuristics.

(a) *BestMRE(P,T)* compared to combined neighborhood heuristics.

	Taillard			
	W	S	B	?
AdaptR-d	0	0	28	2
AdaptR-s	0	0	28	2
AdaptP	0	0	29	1
RandP	0	0	16	14

(b) *BestMRE(P,t)* compared to combined neighborhood heuristics.

	Taillard			
	W	S	B	?
AdaptR-d	0	0	25	5
AdaptR-s	0	0	27	3
AdaptP	0	0	26	4
RandP	0	0	17	13

(c) *BestInstance(P,t)* compared to combined neighborhood heuristics.

	Taillard			
	W	S	B	?
AdaptR-d	0	0	0	30
AdaptR-s	0	0	0	30
AdaptP	0	0	0	30
RandP	2	0	0	28

W/S/B/? denotes significantly Worse, Same, significantly Better and ? no evidence of a significant difference.

Table 6.15: Mean difference over *all time points* for combined neighborhood heuristics. on the Taillard problem set.

	BestMRE(P,T)	*BestMRE(P,t)*	*BestInstance(P,t)*
AdaptR-double	**-0.01513**	**-0.01222**	**0.00554**
	(0.00000)	(0.00000)	(0.00000)
AdaptR-static	**-0.01975**	**-0.01684**	0.00092
	(0.00000)	(0.00000)	(0.94687)
AdaptP	**-0.02030**	**-0.01738**	0.00038
	(0.00000)	(0.00000)	(0.99957)
RandP	**-0.01281**	**-0.00990**	**0.00786**
	(0.00000)	(0.00000)	(0.00000)

Significant differences ($p \leq 0.005$) are marked in bold and p value is shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

and *AdaptR-static* at 28 time points. *RandP* only performs better on 16 time points. There is no evidence that combined methods perform worse than *BestMRE(P,T)* at any time point. Table 6.14(a) comparing against *BestMRE(P,t)* shows a similar picture, with the best performance seen by *AdaptR-static* at 27 time points, followed by *AdaptP* at 26 time points, and then *AdaptR-double* at 25 time points. *RandP* only performs better on 17 time points. Table 6.14(c) shows a the comparison against *BestInstance(P,t)*. We no longer see evidence of the combined methods outperform the pure methods, in fact the only evidence of a significant difference is *RandP* performing worse on 2 time points. For the rest of the time points, there is no evidence of a difference between the combined methods and *BestInstance(P,t)*, the best solution found by any pure method. Table 6.15 shows the results of the all points analysis. We see that all combined methods are better than *BestMRE(P,T)* and *BestMRE(P,t)*, while *AdaptR-double* and *RandP* are significantly worse than *BestInstance(P,t)*.

We now turn to the analysis of difference among the combined heuristics. Table 6.16(a) shows the comparison against *BestMRE(C,T)*, which is *AdaptP* on this problem set. We see that the only evidence of a difference is *RandP* which performs worse on 6 time points. Table 6.16(a) shows the

Table 6.16: Count of significant differences over *30 time points* for combined neighborhood heuristicsagainst *BestMRE* and *BestInstance* of combined neighbourhood heuristics.

(a) *BestMRE(C,T)* compared to combined neighborhood heuristics.

	Taillard			
	W	S	B	?
AdaptR-d	0	0	0	30
AdaptR-s	0	0	0	30
AdaptP	0	30	0	0
RandP	6	0	0	24

(b) *BestMRE(C,t)* compared to combined neighborhood heuristics.

	Taillard			
	W	S	B	?
AdaptR-d	0	0	0	30
AdaptR-s	0	6	0	24
AdaptP	0	24	0	6
RandP	6	0	0	24

(c) *BestInstance(C,t)* compared to combined neighborhood heuristics.

	Taillard			
	W	S	B	?
AdaptR-d	23	0	0	7
AdaptR-s	9	0	0	21
AdaptP	4	0	0	26
RandP	27	0	0	3

W/S/B/? denotes significantly Worse, Same, significantly Better and ? no evidence of a significant difference.

Table 6.17: Mean difference over *all time points* for combined neighborhood heuristicson Taillard's benchmarks.

(a) Among combined neighborhood heuristics

	AdaptR-double	AdaptR-static	AdaptP	RandP
AdaptR-double		**0.00462**	**0.00517**	-0.00232
		(0.00000)	(0.00000)	(0.01819)
AdaptR-static	**-0.00462**		0.00054	**-0.00694**
	(0.00000)		(0.90240)	(0.00000)
AdaptP	**-0.00517**	-0.00054		**-0.00749**
	(0.00000)	(0.90240)		(0.00000)
RandP	0.00232	**0.00694**	**0.00749**	
	(0.01819)	(0.00000)	(0.00000)	

(b) *BestMRE* and *BestInstance* of combined neighbourhood heuristics

	BestMRE(C,T)	*BestMRE(C,t)*	*BestInstance(C,t)*
AdaptR-double	**0.00517**	**0.00535**	**0.01143**
	(0.00000)	(0.00000)	(0.00000)
AdaptR-static	0.00054	0.00073	**0.00681**
	(0.99194)	(0.96380)	(0.00000)
AdaptP	0.00000	0.00018	**0.00627**
	(1.00000)	(0.99998)	(0.00000)
RandP	**0.00749**	**0.00767**	**0.01375**
	(0.00000)	(0.00000)	(0.00000)

Significant differences ($p \leq 0.005$) are **shown in bold** and p values are shown in (). Negative values indicate the algorithm in the row performs better than the algorithm in the column.

comparison against *BestMRE(C,t)*, and we see the same results for *AdaptR-double* and *RandP*. *AdaptR-static* was better at time 6 points and *AdaptP* was better for the other 25 time points. Table 6.16(c) shows the comparison against *BestInstance(C,t)*, the best solution found on each instance by a combined method. Here we see that the best performance is *AdaptP* which is only worse on 4 time points followed by *AdaptR-static*, which is worse on 9 only points. The other methods, *AdaptR-double* and *RandP*, perform far worse, having significantly worse performance on 23 and 27 time points respectively. Table 6.17(a) shows the results of the all points analysis. Again, we see the strongest performers are *AdaptR-static* and *AdaptP*, which have no evidence difference in performance between each other but are better than the *AdaptR-double* and *RandP*. The worst performers, *AdaptR-double* and *RandP* also have no evidence of a difference in performance between each other. Table 6.17(b) shows a comparison of combined methods against the best performing combined methods across all time points. We see that although there is evidence of a difference between *BestInstance(C,t)*, the best performing methods *AdaptR-static* and *AdaptP* are less than 0.7% worse. The worst performing combined method, *RandP* has performance that is 1.4% worse, twice that of the best performers.

6.9.2 Discussion

The results on Taillard's problem instances are interesting from several perspectives. We used these instances to validate our approach and the results show that the combined methods with learning are able to outperform the other methods, in terms of MRE, for all time limits. Comparing the results of the combined learning methods, we see a performance increase over the individual neighbourhood heuristics. Comparing the combined methods to each other, we see that the best learning schemes provide a significant advantage to randomly alternating between neighbourhoods.

Recall that we did not tune the pure neighbourhood heuristics to solve Taillard's benchmarks. The greater disparity in performance of the pure heuristics against the combined methods shows that learning may provide a

greater benefit when it is used in situations where the pure methods cannot be tuned beforehand, or if the problem set has significant variation in difficult or problem size. These results provide evidence that combining neighbourhood heuristics with learning provides a greater benefit when applied to unseen problem domains.

Although not shown in our presentation of results, we compared our approach to the best known upper bounds [105]. We are on average no worse than 3% from the best known solutions despite only giving our method 10 minutes to solve these instances.

6.10 Adaptive Large Neighbourhood Search

We now discuss the approaches called adaptive large neighbourhood search (ALNS) [92] and self-adaptive LNS (SA-LNS) [59] which have a strong similarity to the work presented in this chapter. Both of these approaches are instantiations of the adaptive probability (*AdaptP*) algorithm with the adaptive runtime (*AdaptR*) weight updating scheme. That is, heuristics are selected with a roulette wheel approach and weights are updated using Equation 6.10 from Section 6.8.1.1.

The authors of ALNS present a thorough study of their approach on many classes of vehicle routing problems. An interesting variation of the experiments presented in the ALNS work is the use of simulated annealing as the acceptance criteria in the master LNS algorithm. In ALNS, weights are updated after 100 iterations (similar to a time slice). During each iteration, the current weights are used to repeatedly select and then apply a neighbourhood. After a M iterations, weights are reset to their initial values. The performance measure is simply the improvement in solution quality. No normalization of performance values is applied, although the authors mention this may be useful if some algorithms are more expensive than others.

Laboire & Godard present the SA-LNS algorithm [59] with results on a wide range of single mode scheduling problems. As in our experiments, SA-LNS applies weight learning after each application of a neighbourhood heuristic, using the same performance measure as *AdaptR* (improvement /

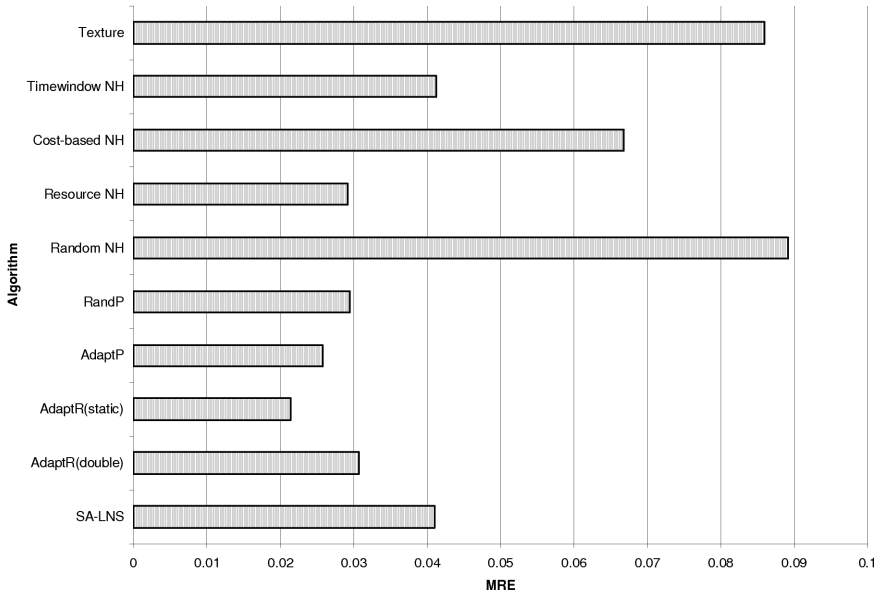


Figure 6.13: Comparison of SA-LNS to our methods.

time spent) but without normalization. The master LNS algorithm uses hill climbing as the acceptance criteria. A novel feature of SA-LNS work is applying the same weight scheme to parameters. A discrete set of parameter values are provided and weights are learnt for each parameter value. In this manner, they repeat a process of selecting a heuristic followed by selecting a parameter setting. Unfortunately, no details on the number of parameter combinations used are given in the paper.

The results of the ALNS and SA-LNS experiments are very interesting. Robust and competitive performance on a diverse range of benchmarks is observed. Indeed, in many cases the best known bound on benchmark instances are improved. It is worth noting that the latest version of ILOG's commercial optimization product OPL now recommend SA-LNS as a robust search method for optimizing scheduling problems [55].

We now present an brief comparison of our work to SA-LNS on a subset of problems that were solved by both approaches. This subset consists of 12 of

Taillard's problem instances[16] and a comparison of MRE[17] is shown in Figure 6.13. As can be seen, our method produced better quality solutions than SA-LNS. While it is difficult to compare CPU times on different hardware,[18] our results were obtained using 600 seconds, whereas theirs report a CPU time of 1320 to 1680 seconds. To draw stronger conclusions on the strengths of each approach, a more detailed study is required.

We note the timing of the development of these similar approaches. The experiments in this chapter were completed in December 2005 culminating in a submission to the Journal of Mathematical Modelling and Algorithms on 15 January 2006 and published on-line on 7 January 2009. SA-LNS appeared in a paper at the MISTA conference on 28 August 2007. We became aware of ALNS through a reference in the SA-LNS paper. Although the ALNS paper was published in 2007, the article states that a technical report of the work was available on the authors website in October 2005.

6.11 Conclusion

In this chapter, we have presented several low knowledge techniques to apply large neighbourhood search to optimization problems. Each of the methods presented can be applied without knowledge of the underlying algorithms or the problem domain. The inputs required are a set of algorithms, a space of parameters, and if available, a sample of problem instances. While the expertise to develop algorithms remains, we believe that algorithm development is not the critical issue since commercial software exists that provide such algorithms in the form of toolkits. The critical issue is, given such a toolkit, how does a non-expert use it. The methods presented in this chapter deal with the effective utilization of such a toolkit. We believe these control methods are a significant step towards producing a system that will consis-

[16]SA-LNS results are from http://wikix.ilog.fr/wiki/bin/view/Scheduler/SA-LNS.

[17]MRE is computed here using the best known upper bounds shown in Appendix A.

[18]SA-LNS experiments were run on a Dell Latitude D620 laptop, 2 GHz CPU with 2GB RAM, with an implementation on top of ILOG CP 1.1. This is in comparison to our implementation on a Pentium IV 1.8 Ghz CPU with 512MB of RAM using the older version of ILOG Scheduler 6.0.

tently produce high quality solutions and reduce the expertise required to apply optimization technology.

Chapter 7

Conclusions and Future Work

7.1 Contributions

The primary contributions of this dissertation are methods to make optimization technology easier to use. Our central thesis is that a practical and effective way to achieve ease-of-use is through the application of low knowledge algorithm control methods. We have presented several such control methods, applied them in different contexts and shown strong performance. The performance is not the primary achievement; it is that this performance is possible using simple and general control methods. These mechanisms can be applied to many different types of optimization algorithms and can therefore make optimization technology more accessible to people without the technical expertise that is currently required.

In summary, the contributions of this dissertation are:

- The introduction of simple and general low knowledge algorithm control methods and the demonstration that they perform well on scheduling problems.
- A framework for algorithm control and an analysis of the literature on algorithm control methods.
- The extension of the algorithm selection problem to the more general problem of algorithm control.

- A low knowledge control system applied to large neighbourhood search and the application of this system to very large scheduling problems.

7.1.1 Low Knowledge Algorithm Control

The central thesis of this dissertation is that a low knowledge approach to algorithm control supports the goal of reducing the required expertise to effectively apply optimization technology. We have challenged the belief that more information will lead to a better reasoning system and proposed that low knowledge control systems are both easier to implement and more robust to change. We highlighted the challenges and limitations of using a high knowledge approach to create a model of the world in order to reason about decisions. The engineering of high knowledge models is in itself a task that requires significant expertise and effort and therefore does not address the ease-of-use goal. Worse still, high knowledge models are specific to the problem domains for which they are developed. This specificity reduces the ability to reuse a model and carries a risk of over-fitting when problem instances vary from those considered during model building.

We demonstrated that low knowledge control methods perform well even though they do not have high knowledge models of the world. Several control methods were presented and evaluated against optimal high knowledge selection methods. The high knowledge methods used were purely theoretical in that they never make errors and take no CPU time to compute. Despite the unfair advantage we gave the high knowledge methods, our empirical results suggest that low knowledge approaches perform well: the low knowledge selection methods performed as well as a high knowledge classifier that could determine the best algorithm for a problem set. When we applied low knowledge control methods that alternated between algorithms, performance was at least as good as the high knowledge selection method that could identify the best algorithm for each problem instance, and in many cases, the low knowledge switching control method actually outperformed the high knowledge selection method. These results provide evidence that despite being easier to implement, low knowledge approaches can perform as well as high

knowledge approaches.

Our empirical investigation applied control methods to state-of-the-art scheduling algorithms for minimizing makespan on the job shop scheduling problem. The problem instances ranged in size from 20x20 (400 activities) to 40x40 (1600 activities). The larger problem instances extend the limits of the size of problems that are typically considered in academic research and are more representative of the size seen in industrial applications. We applied control methods in two contexts: choosing among solvers and choosing among large neighbourhood heuristics.

7.1.2 Analysis of Control Algorithms

We developed a framework to define the structure of control methods through knowledge capture and control decisions. The framework provided a tool to categorize and understand the literature, resulting in an analysis of the benefits and shortcomings of on-line and off-line control approaches. From this analysis, we gained the insight that a control method that makes on-line control decisions, during the course of search, has several benefits. First, it is not committed to making only a single decision. Repeated decision making allows the control mechanism to be more robust to mistakes; control decisions can changed if they do not perform well. Second, it reduces the need to capture all of the information to reason about performance before we make our decision. The information gathered while executing a decision is extremely valuable to making future decisions.

7.1.3 Algorithm Control Problem

We extended the definition of the algorithm selection problem [94] to the more general problem of algorithm control, where algorithm selection is repeated during execution. The algorithm control approach allows the interleaving of knowledge capture and decision making. We showed that low knowledge control methods are able to perform as well as high knowledge selection methods when applied to scheduling algorithms.

7.1.4 Control Applied to Large Neighbourhood Search

We presented a scheduling system to solve industrial sized scheduling problems that is based on low knowledge control and large neighbourhood search. Several mechanisms allow this system to automatically configure itself to achieve strong performance. This configuration takes place in two phases: during an off-line training phase and during an on-line execution phase. The system performed strongly on all problems we tested, including problems with 1600 activities and some challenging academic benchmarks.

In the off-line training phase, a tuning procedure determines the best parameter settings for the components of the system. The tuning procedure executes many control decisions at each point of search to determine the best performing components and parameters. We applied this procedure to the neighbourhood heuristics, although other aspects of the system can also be tuned, such as search algorithms. A novel feature of our tuning method is that the evaluation mechanism executes all components together, rather than tuning each heuristic in isolation.

When the system is applied to new problem instances in execution mode, control methods determine the best performing heuristics, allowing the system to adapt to each problem instance and exploit the heuristics that perform best at each stage of search. The best performing control method was based on reinforcement learning and roulette wheel selection.

The resulting system represents a 'black-box' scheduling system that provides a framework to automatically exploit any new neighbourhood heuristics or search algorithms that are added. We believe this system framework can be easily applied to other problem domains. The algorithm designer simply produces components and measurements of performance. The user of such a system then produces problem instances and the system will automatically adapt to achieve the best performance.

7.2 Future Work

The work in this dissertation represents an advance in applying machine learning methods to optimization technology. Our approach has been guided by the goal of making optimization technology easier to use. In doing so, we found that many of the existing approaches in machine learning are, quite simply, inappropriate since they are very difficult to apply, and once developed, become very specific to the problem they solve. We have identified some simple guidelines for machine learning methods to support the goal of generality and ease of use. In this section, we describe these guidelines and point to possible future research directions and application areas.

Our control mechanisms operate with a simple, low knowledge, view of the world that is based only on observations of performance. By taking this perspective, we avoid the problem of producing a complex model for reasoning about the world and replace it with an approach that reasons based on the observation of actions.

The broad areas of future research, then, are:

- To understand the problem areas where control can be applied (or not) and the types of algorithms that are suitable for control.

- To discover effective mechanisms of control. This research area includes investigating types of control decisions, learning mechanisms, search space exploration, parallel execution, action sequencing, and execution monitoring.

- To understand how to configure control systems. This area includes choosing the algorithms, tuning algorithms for combination, and determining the best control method.

- To develop new component algorithms. This area is concerned with developing algorithms that are explicitly designed to be combined with other algorithms.

7.2.1 Application Areas

The low knowledge control approach is very general: it can be applied to many types of algorithms and problems. To apply the approach requires a measure of performance, a collection of algorithms to choose among, and the ability to execute these algorithms through control decisions. In this section, we describe these requirements and then follow with some suggestions on interesting application areas.

To apply the algorithm control methods described in this dissertation, there are three requirements: a measure of algorithm performance, anytime behavior, and the ability to share solutions among other algorithms. The first two requirements are mandatory but the last requirement, of sharing solutions, is only required if algorithm switching methods are used.

1. A **measure of performance** is required for the control methods to make decisions based on how well each algorithm is performing. The performance measure should have a direct impact on the aim of the control system.

2. The **anytime behavior** requirement is that, at any time, an algorithm can be halted and will return the best solution it has found so far.

3. The ability of an algorithm to **improve a solution** found by another algorithm is a requirement of the most successful control methods we discovered.

In considering whether a class of algorithms can meet these requirements, it is interesting to consider the level at which algorithm control is applied. It can be applied at a high level, as a wrapper around an entire search procedure, or at a low level, on the heuristics and other components of a single search procedure.

We are interested in the application of algorithm control to constraint satisfaction and other areas where there are many algorithms and no clear way to identify the best algorithm to apply. A challenge with constraint satisfaction is the difficulty in identifying a good measure of performance before

the problem is solved [4, 40, 68, 86, 117]. A second challenge is the ability to share (partial) solutions among algorithms. Sharing solutions appears easier with local search based methods or when control methods are applied inside a constructive search algorithm (e.g., to choice point heuristics or propagation algorithms). Other areas of interest are mixed integer programming [28] and other types of heuristic search algorithms, such as those used in vehicle routing [92].

7.2.2 Control Mechanisms

Algorithm control is an instance of the general problem of action selection, which has long been one of fundamental challenges in AI. Put simply, action selection is the process of choosing what to do, given some knowledge and observations regarding the current state of the world, in order to achieve a goal (or set of goals). The primary driver for research in this area has been robotics, and more recently, AI for characters in video games. Both of these domains require decisions to be made rapidly and in environments with a large degree of uncertainty and change, not unlike the domain of algorithm control.

The AI planning community has typically addressed the action selection problem by generating a sequence of actions (a plan) to achieve goals, and in an execution context, replanning by updating the sequence of actions based on observations. This type of reasoning is fundamentally high knowledge, and the planning community continues to struggle with the ability to not only represent the world, but to be able to reason as more and more complex models are developed. A very different approach is the subsumption architecture [19] which is a low knowledge approach to control. Rather than creating complex models of the world and the way it operates, a subsumption architecture creates simple layers that are concerned with a single goal, such as avoiding obstacles, and uses sensors to observe the state of the world. These simple layers are then assembled in a hierarchy, which produces a system to achieve the overall system goals. From a practical standpoint, the subsumption architecture approach has been far more successful than AI planning;

more than three million floor-vacuuming robots based on this design have been sold.

The control methods in this dissertation have been developed with the single goal of maximizing algorithm performance. The sensors in our systems have been measures of algorithm performance and the actions are the application of algorithms. In this dissertation, we have only explored a limited number of possible configurations for algorithm control. The following lists some interesting directions to extend our work:

Exploration of Search Space In our work, each algorithm is always asked to improve the best known solution, a strategy known as the hill-climbing metaheuristic. Other metaheuristics, such as simulated annealing and tabu search, have been applied in an algorithm control context [92, 103] but no study has been performed to compare metaheuristics in such a context. Solution pools are another interesting idea that could be used in the context of algorithm control [112].

Parallel Processing Our control methods so far have operated in a serial fashion and have not considered parallel processing. When multiple control decisions are executed at the same time, the risk of committing to the wrong decision decreases [50] and the knowledge gained regarding algorithm performance increases, since more algorithms are evaluated. It is interesting to consider policies for parallel execution and for sharing information among algorithms.

Ordering of Algorithms We have only attempted to learn how to combine effective algorithms rather than considering the order in which algorithms are applied. Investigating the impact of algorithm ordering and developing efficient mechanisms to learn and store this information could lead to the discovery of complementary interactions between algorithms.

Execution Monitoring The investigation of mechanisms that monitor algorithm performance during execution may provide better control as compared to running each algorithm for a fixed time. It is interesting to

consider work such as deliberation control [62] and other mechanisms to determine when to stop an algorithm.

Learning Mechanism We have explored the application of two different weight learning mechanisms, but there are many areas that warrant further investigation such as the learning rate, exploration/exploitation mechanisms, and learning different weights for each stage of search. Of interest is the application of other learning mechanisms, aside from reinforcement learning, to low knowledge algorithm control.

7.2.3 Configuring Control Systems

The aim of configuring an algorithm control system is to produce an on-line control system which not only maximizes algorithm performance, but produces a robust system that is able to perform very well on all instances and time limits. The configuration process can be computationally expensive, within reason, since it is performed off-line. However, a requirement of the configuration process is that it is easy to apply without expert knowledge, since the goal of our work is to make algorithms easier to use.

Consider the desirable properties of an optimization system that is comprised of many algorithms. The system should provide robust solving performance, performing well on every problem instance it is applied to and across all time limits. Each of the algorithms that makes up the system should assist in this performance, either by directly improving the quality of the solution, or assisting other algorithms in improving the quality of the solution (for example, through diversification). Each of these algorithms should have a unique contribution towards this performance, in that it excels on a particular type of problem or stage of search.

At first, it may appear as if this configuration process is very similar to portfolio optimization. Each algorithm has a risk/reward ratio that is an indicator of the algorithm's computational cost/performance. The optimal algorithm portfolio, then, is one that maximizes the expected value, by picking the right algorithms. Unfortunately, portfolio optimization considers each algorithm to be independent which is clearly not the case when solu-

tions are shared among algorithms. If one algorithm produces solutions that are very difficult for another algorithm to improve, then the combined system will perform poorly. Without a deep analytical and empirical analysis of these interactions it is difficult to determine this *a priori*. As more algorithms are added, the number of possible interactions increases quadratically. To perform a complete analysis requires inspecting every interaction, at each control point, on every problem instance, which is clearly impractical.

Despite these challenges, automatic configuration approaches, such as parameter tuning [15, 52] can be applied to determine a configuration of algorithm components. The parameters to be tuned are the parameters of the control system and the set of algorithms to be used, and each individual algorithm's parameters. While these tuning methods are ideal from the perspective of being low knowledge, they are extremely computationally expensive. The evaluation mechanism requires that each configuration is applied to sample problem instances. The work of Hutter et al. [52] attempts to limit evaluations of unpromising configurations, however the time taken to search the parameter space can clearly become very expensive.

The simple tuning procedure we presented in Chapter 6 is a first step towards building a mechanism to configure algorithm control systems. The procedure applies each possible algorithm[19] at every control decision. One of the next steps in this research direction is to discover more effective ways of choosing algorithms; we simply took the best parameter configuration for each distinct heuristic, based on a measure of utility. It is interesting to examine algorithm performance over time, by evaluating performing over the entire run, thus learning the best configuration for different stages of search. Another direction is to apply the ideas of FocusedParamILS [52] to limit the evaluation of unpromising parameter configurations, which could speed up this procedure.

[19]Note that in our experiment each algorithm had a range of parameter settings, thus each parameter configuration represents an algorithm, for a total of 21 such algorithms.

7.2.4 Developing Algorithms for Algorithm Control

In this section, we discuss designing algorithms that are specifically for use in an algorithm control system. Algorithm design for control systems is an interesting research area as it is a departure from the winner-takes-all school of algorithm development. In the thinking of algorithm control, we want to develop specialized algorithms that excel in one area, or complementary algorithms which interact well with each other, rather than a single algorithm that dominates all others. A common pattern in computer science research is to create hybrids from combinations of existing algorithms to produce an algorithm that performs better than its components. This is the synergy benefit that we hope to gain when combining algorithms with algorithm control, albeit, we wish each hybrid to be realized automatically and that a 'unique' hybrid is learned for each problem instance as we solve it.

An interesting area then, is to take successful algorithms and deconstruct them into their components [113]. Given a set of building block algorithms, the algorithms can be combined again in different ways to produce new algorithms. Collections of such algorithmic building blocks are offered in commercial toolkits such as Comet [108]. Decomposing an algorithm into parts is similar to the suggestion of exposing parameters for algorithm tuning [52, 76]. The notable difference in our suggestion is that, unlike tuning which produces a single static configuration before execution, we suggest producing a collection of building blocks that can be dynamically applied during execution using on-line algorithm control.

The benefit of applying on-line algorithm control is that the configuration process is concerned only with which algorithm to include and the control policy, rather than exploring rigid parameter combinations of building block configurations. When the control system is applied to problem instances, it will apply the available algorithms which we believe will produce a more robust system that performs better across all instances and time limits.

7.3 Conclusion

The central thesis of this dissertation is that *low knowledge* control methods for optimization algorithms allow non-experts to achieve high quality results from optimization technology. The primary motivation for our research is to extend the reach of optimization technology by making it more accessible. To this end, we have presented methods that not only provide high quality solutions, but do so without the requirement for significant effort and expertise by a practitioner who wants to use off-the-shelf methods to solve a problem. It is of note that our control methods provide robust solving performance, both across problem instances and across time limits.

In particular, in this dissertation:

- We created and investigated mechanisms for algorithm control that do not require detailed knowledge of the problem domain or algorithm behaviour. These low knowledge control methods make decisions based only on observations of algorithm performance. This approach lowers the expertise required to employ these control mechanisms since no knowledge engineering effort is required to apply control methods to new algorithms or problem domains.

- We compared low knowledge control approaches to idealized high knowledge approaches in the domain of scheduling. We presented an analysis against the best possible high knowledge approaches and observed strong performance. Although high knowledge methods have been shown to provide good performance, they fail to make these algorithms easier to use since they shift expertise from analysis of algorithm performance to an analysis of high knowledge models.

- We applied low knowledge control methods to a large neighbourhood search configuration for solving industrial-sized scheduling problems. The control mechanisms were applied to the selection of neighbourhood heuristics during search. In addition, a tuning procedure for neighbourhood heuristics was presented that considers interactions between

neighborhoods. Strong performance was observed across all problem sets and time limits.

Appendix A

Detailed Results on Taillard's JSP Instances

Abbreviation	Description
TA	Problem Instance
UB	Best known upper-bound [105]
ARd	AdaptR-double (Combined NH)
ARs	AdaptR-static (Combined NH)
AP	AdaptP (Combined NH)
RP	RandP (Combined NH)
RN	Random NH
RS	Resource NH
CB	Cost-Based NH
TW	Time Window NH
BT	Texture with bounded backtracking
	Bold indicates the best found solution by our methods

Table A.1: Legend for Results in Appendix A.

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
11	1359	1398	1403	**1387**	1391	1434	1396	1439	1408	1430
12	1367	1386	1377	**1367**	1371	1409	1377	1412	1387	1430
13	1342	1365	1359	1359	1363	1412	**1353**	1451	1384	1406
14	1345	**1345**	1347	**1345**	**1345**	**1345**	**1345**	1380	1386	1396
15	1339	1385	**1369**	1399	1377	1411	1394	1422	1389	1468
16	1360	1377	1381	**1370**	1386	1403	1377	1409	1403	1457
17	1462	**1493**	1511	1523	1532	1516	1506	1521	1530	1530
18	1396	1438	1426	1428	1423	1470	**1415**	1483	1457	1491
19	1335	1383	**1373**	1377	1377	1388	1383	1420	1387	1418
20	1348	1363	**1361**	1380	1373	1364	1362	1463	1387	1411

Table A.2: Taillard 20x15

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
21	1644	1683	**1681**	1692	1713	1721	1722	1777	1691	1736
22	1600	1644	1642	1664	**1633**	1677	1656	1671	1651	1683
23	1557	1592	**1576**	1597	1593	1645	1618	1638	1607	1647
24	1646	1681	1676	1693	**1674**	1710	1688	1751	1680	1712
25	1595	1635	1643	1653	1632	1650	**1605**	1681	1654	1716
26	1645	1684	1676	**1655**	1690	1729	1678	1750	1692	1743
27	1680	1724	1732	**1703**	1713	1783	1727	1801	1753	1793
28	1603	1626	1631	1654	1637	1674	**1617**	1701	1629	1698
29	1625	1645	1651	1652	1648	1659	**1639**	1678	1676	1699
30	1584	1642	1631	1628	1627	1669	1619	1682	1641	1683

Table A.3: Taillard 20x20

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
31	1764	1826	1780	1775	1788	2112	**1768**	1945	1816	1963
32	1795	1853	1855	1855	1884	2060	**1833**	1905	1883	2068
33	1791	1921	1853	1860	**1843**	2126	1876	1923	1906	2098
34	1829	1897	1900	1881	1879	2052	**1877**	1982	1918	2047
35	2007	**2007**	**2007**	2010	**2007**	2080	**2007**	2047	**2007**	2044
36	1819	1847	**1828**	1842	1851	2190	1856	1918	1870	2034
37	1771	1831	1834	1823	1834	2018	**1799**	1865	1887	1995
38	1673	1715	1712	**1702**	1743	1823	1705	1762	1827	1893
39	1795	1837	**1814**	1833	**1814**	1956	1822	1957	1896	1990
40	1674	1759	**1718**	1720	1734	2141	1755	1828	1736	1918

Table A.4: Taillard 30x15

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
41	2018	2139	2120	**2072**	2082	2484	2077	2323	2165	2364
42	1949	2012	2047	2025	2029	2228	**2005**	2136	2081	2185
43	1858	1972	1923	**1908**	1944	2210	1962	2002	1970	2175
44	1983	2116	2082	**2061**	2080	2373	2075	2139	2114	2274
45	2000	2094	2063	2083	2054	2399	**2037**	2151	2113	2263
46	2015	2131	**2080**	2108	2120	2411	2094	2292	2156	2359
47	1903	1982	1957	**1944**	1950	2013	1967	2060	2053	2174
48	1949	2034	2034	2057	2007	2103	**2005**	2127	2085	2217
49	1967	2107	2096	2035	**2034**	2297	2051	2112	2129	2249
50	1926	2031	2058	2014	2014	2078	**2001**	2069	2104	2214

Table A.5: Taillard 30x20

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
51	2760	2901	2795	**2760**	2765	3420	2851	2928	3120	3138
52	2756	2771	**2756**	**2756**	2776	3278	2929	2821	2886	3122
53	2717	2740	**2717**	2717	2717	3034	2717	**2717**	2843	3012
54	2839	**2839**	**2839**	**2839**	**2839**	3135	**2839**	2869	3135	3106
55	2679	2730	2701	**2679**	2707	3119	2828	2711	3119	3114
56	2781	**2781**	**2781**	**2781**	**2781**	3038	**2781**	**2781**	2959	3038
57	2943	**2943**	**2943**	**2943**	**2943**	3232	2953	3018	3023	3231
58	2885	**2885**	**2885**	**2885**	**2885**	3314	2973	**2885**	2991	3286
59	2655	2745	**2655**	2659	2669	3083	2789	2697	2965	3065
60	2723	2800	**2723**	**2723**	2743	2978	2733	2739	2975	2978

Table A.6: Taillard 50x15

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
61	2868	2984	2987	2995	3008	3335	3188	**2970**	3132	3321
62	2869	3074	**3045**	3076	3053	3341	3155	3063	3324	3341
63	2755	2850	**2838**	2845	2876	3172	2997	2878	3118	3170
64	2702	**2755**	2811	2784	2811	3172	2959	2831	2997	3171
65	2725	2974	2970	**2845**	2890	3365	3030	3035	3075	3313
66	2845	2968	2989	**2966**	**2966**	3240	3104	3073	3208	3238
67	2825	**2906**	2933	2920	2941	3247	3051	2963	3123	3242
68	2784	**2878**	2904	2857	**2818**	3052	2934	2885	2994	3052
69	3071	3218	3233	3169	**3136**	3481	3336	3145	3481	3432
70	2995	3225	3207	3150	**3132**	3443	3309	3163	3285	3440

Table A.7: Taillard 50x20

TA	UB	ARd	ARs	AP	RP	RN	RS	CB	TW	BT
71	5464	**5650**	5702	5709	5796	5817	5817	5687	5817	5817
72	5181	**5204**	5277	5312	5403	5414	5412	5247	5414	5414
73	5568	**5707**	5869	5752	6020	6058	6053	5756	6058	6058
74	5339	**5339**	5442	5389	5518	5568	5563	5373	5568	5568
75	5392	5748	5806	5812	5921	5957	5952	**5714**	5957	5957
76	5342	5535	5579	5761	5809	5963	5958	**5524**	5763	5963
77	5436	**5461**	5585	5517	5697	5767	5765	5496	5767	5767
78	5394	5553	5592	5641	5669	5733	5729	**5509**	5733	5733
79	5358	**5374**	5415	5401	5513	5563	5563	5382	5563	5563
80	5183	**5271**	5328	5310	5399	5434	5431	5303	5434	5434

Table A.8: Taillard 100x20

Bibliography

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.

[2] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[3] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[4] A. Arbelaez, Y. Hamadi, and M. Sebag. Online heuristic selection in constraint programming. In *Proceedings of the Second International Symposium on Combinatorial Search*, 2009.

[5] S. Bain, J. Thornton, and A. Sattar. Evolving algorithms for constraint satisfaction. In *Proceedings of the Congress on Evolutionary Computation*, pages 265–272, Portland, Oregon, 20-23 June 2004. IEEE Press.

[6] P. Baptiste, C. Le Pape, and W. Nuijten. Constraint-based optimization and approximation for job-shop scheduling. In *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95*, 1995.

[7] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based Scheduling*. Kluwer Academic Publishers, 2001.

[8] J. C. Beck. *Texture measurements as a basis for heuristic commitment techniques in constraint-directed scheduling*. PhD thesis, University of Toronto, 1999.

[9] J. C. Beck and M. S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1):31–81, 2000.

[10] J. C. Beck and E. C. Freuder. Simple rules for low-knowledge algorithm selection. In *Proceedings of the First International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 50–64, 2004.

[11] J. C. Beck and L. Perron. Discrepancy-bounded depth first search. In *Proceedings of the Second International Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems*, 2000.

[12] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):22–62, 2007.

[13] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[14] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. The tractability of global constraints. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, 2004.

[15] M. Birattari. *The Problem of Tuning Metaheuristics as seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, 2004.

[16] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, 2002.

[17] J. Blazewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1):1–33, 1996.

[18] J. Borrett, E. P. K. Tsang, and N. R. Walsh. Adaptive constraint satisfaction: the quickest first principle. In *Proceedings of the Twelfth European Conference on Artificial Intelligence*, pages 160–164, 1996.

[19] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1&2):3–15, June 1990.

[20] E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, December 2003.

[21] E. K. Burke, S. Petrovic, and R. Qu. Case-based heuristic selection for timetabling problems. *Journal of Scheduling*, 9(2):115–132, 2006.

[22] Y. Caseau, F. Laburthe, C. Le Pape, and B. Rottembourg. Combining local and global search in a constraint programming environment. *Knowledge Engineering Review*, 16(1):41–68, 2001.

[23] A. Cesta and A. Oddi. Gaining efficiency and flexibility in the simple temporal problem. In L. Chittaro, S. Goodwin, H. Hamilton, and A. Montanari, editors, *Proceedings of the Third International Workshop on Temporal Representation and Reasoning*, Los Alamitos, CA, May 1996. IEEE Computer Society Press.

[24] A. Cesta, A. Oddi, and S. F. Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 742–747. AAAI Press / The MIT Press, 2000.

[25] A. Chabrier, E. Danna, C. Le Pape, and L. Perron. Solving a network design problem. *Annals of Operations Research*, 130:217–239, 2004.

[26] D. M. Chickering. The WinMine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA, 2002.

[27] P. R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, Cambridge, Mass., 1995.

[28] E. Danna. Integrating local search techniques into mixed integer programming. *4OR: A Quarterly Journal of Operations Research*, 2(4):321–324, 2004.

[29] E. Danna and L. Perron. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 817–821, 2003.

[30] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[31] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems; a survey. *Constraints*, 1998.

[32] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[33] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1990.

[34] S. L. Epstein, E. C. Freuder, R. Wallace, A. Morozov, and B. Samuels. The adaptive constraint engine. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 525–540, 2002.

[35] A. Flaxman. *Average-case analysis for combinatorial problems*. PhD thesis, Dept. of Mathematical Sciences, Carnegie Mellon University, 2006.

[36] E. C. Freuder. Synthesizing constraint expressions. *Communications of the Association for Computing Machinery*, 21(11):958–966, November 1978.

[37] E. C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997.

[38] E. C. Freuder. Holy grail redux. *Constraint Programming Letters*, 1:3–5, 2007.

[39] Y. Freund. Boosting a weak learning algorithm by majority. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, 1990.

[40] M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Special Issue of the Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2006.

[41] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[42] C. Gebruers, B. Hnich, D. G. Bridge, and E. C. Freuder. Using CBR to select solution strategies in constraint programming. In *Proceedings of the Sixth International Conference on Case-Based Reasoning*, pages 222–236, 2005.

[43] D. Godard, P. Laborie, and W. Nuijten. Randomized large neighborhood search for cumulative scheduling. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, pages 81–89, 2005.

[44] C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.

[45] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431–437, 1998.

[46] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the Sixteenth Eureopean Conference on Artificial Intelligence*, pages 475–479. IOS Press, 2004.

[47] P. Hansen and N. Mladenovic. *Handbook of Metaheuristics*, chapter 6: Variable Neighborhood Search, pages 145–184. Springer, 2003.

[48] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier / Morgan Kaufmann, 2004.

[49] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A Bayesian approach to tacking hard computational problems. In *Proceedings of the Seventeenth Conference on Uncertainty and Artificial Intelligence*, pages 235–244, 2001.

[50] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, January 3 1997.

[51] F. Hutter, Y. Hamadi, K. Leyton-Brown, and H. H. Hoos. Performance prediction and automated tuning of randomized and parametric algorithms. In *Twelfth International Conference on Principles and Practice of Constraint Programming*, pages 213–228, 2006.

[52] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 2009. under review.

[53] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second Conference on Artifical Intelligence*, pages 1152–1157, 2007.

[54] ILOG. *ILOG CONCERT 2.0 User's Manual*. ILOG, S.A., 2003.

[55] ILOG. ILOG CP Optimizer 2.0. *The Right Hand Side - ILOG's optimization eNewsletter*, 1(2), October 2008. http://www.ilog.com/ optimization/ the-right-hand-side/ 2/ PU_CP_Optimizer.html.

[56] I. Katriel and P. Van Hentenryck. Maintaining longest paths in cyclic graphs. In *Proceedings of Eleventh International Conference on Principles and Practice of Constraint Programming*. Springer-Verlag, 2005.

[57] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[58] P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143:151–188, January 2003.

[59] P. Laborie and D. Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proceedings of the Third Multidisciplinary International Scheduling Conference*, 2007.

[60] M. Lagoudakis and M. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *Proceedings of the LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, 2001.

[61] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518. Morgan Kaufmann, San Francisco, CA, 2000.

[62] K. Larson and T. Sandholm. Using performance profile trees to improve deliberation control. In *Nineteenth National Conference on Artificial Intelligence*, July 2004.

[63] C. Le Pape. Implementation of resource constraints in ILOG Schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.

[64] C. Le Pape, P. Couronné, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the UK Planning Special Interest Group*, 1994.

[65] C. Le Pape, L. Perron, J. C. Régin, and P. Shaw. Robust and parallel solving of a network design problem. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 633–648, 2002.

[66] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proceed-*

ings of the Ninth International Conference on Principles and Practice of Constraint Programming, 2003.

[67] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 556–572, 2002.

[68] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. Under Review, http://www.cs.ubc.ca/ kevinlb/papers/EmpiricalHardness.pdf, 2008.

[69] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[70] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[71] T. Mautor and P. Michelon. Mimausa: An application of referent domain optimization. Technical report, Laboratoire d'Informatique d'Avignon, 2001.

[72] L. Mercier and P. Van Hentenryck. Edge finding for cumulative scheduling. *Informs Journal on Computing*, 20(1):143–153, 2008.

[73] L. Michel and P. Van Hentenryck. Iterative relaxations for iterative flattening in cumulative scheduling. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 200–208, 2004.

[74] S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1,2):7–43, 1996.

[75] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the Eighth European Conference on Artificial Intelligence*, pages 651–656, 1988.

[76] J. N. Monette, Y. Deville, and P. Van Hentenryck. Aeon: Synthesizing Scheduling Algorithms from High-Level Models. In J. W. Chinneck, B. Kristjansson, and M. J. Saltzman, editors, *Operations Research and Cyber-Infrastructure*, 2009.

[77] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[78] A. Nareyek. Choosing search heuristics by non-stationary reinforcement learning. *Metaheuristics: Computer Decision-Making*, pages 523–544, 2003.

[79] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[80] E. Nudelman. *Empirical Approach To The Complexity Of Hard Problems*. PhD thesis, Stanford University, October 2005.

[81] E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. H. Hoos. SATzilla: An algorithm portfolio for SAT. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004 Competition: Solver Descriptions*, pages 13–14, 2004.

[82] E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. H. Hoos. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Tenth International Conference on Principles and Practice of Constraint Programming*, pages 438–452, 2004.

[83] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 438–452, 2004.

[84] W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ILOG scheduler. *Journal of Heuristics*, 3:271–286, 1998.

[85] W. P. M. Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1994.

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

[87] T.V. Perneger. What's wrong with bonferroni adjustments. *British Medical Journal*, 316(7139):1236–1238, 1998.

[88] L. Perron. Fast restart policies and large neighborhood search. In *Proceedings of the Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2003.

[89] L. Perron and P. Shaw. Combining forces to solve the car sequencing problem. In *Proceedings of the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 225–239, 2004.

[90] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighborhood search. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 468–481, 2004.

[91] M. Petrik. Learning parallel portfolios of algorithms. Master's thesis, Comenius University, 2005.

[92] D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34(8):2403–2435, 2007.

[93] J. F. Puget. Constraint programming next challenge: Simplicity of use. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 5–8, 2004.

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

[95] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[96] R. Schapire. Strength of weak learnability. *Journal of Machine Learning*, 5:197–227, 1990.

[97] Scheduler. *ILOG Scheduler 5.2 User's Manual and Reference Manual*. ILOG, S.A., 2001.

[98] Scheduler. *ILOG Scheduler 5.3 User's Manual and Reference Manual*. ILOG, S.A., 2002.

[99] P. Shaw. Proceedings of the 4th international conference on principles and practice of constraint programming. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, volume 1520, page 417, 1998.

[100] H. A. Simon. The structure of ill-structured problems. *Artificial Intelligence*, 4:181–200, 1973.

[101] B.M. Smith. *Handbook of Constraint Programming*, chapter 11: Modelling, pages 377–406. Elsevier, 2006.

[102] E. D. Sontag. *Mathematical Control Theory. Deterministic Finite-Dimensional Systems*, volume 6. Springer-Verlag, second edition edition, 1998.

[103] E. Soubeiga. *Development and application of hyperheuristics to personnel scheduling*. PhD thesis, University of Nottingham, 2003.

[104] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[105] E. Taillard. Best lower and upper bounds known, from or-lib. Website, 2005. http://mistic.heig-vd.ch/ taillard/ problemes.dir/ ordonnancement.dir/ jobshop.dir/ best_lb_up.txt.

[106] E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press: London and San Diego (see http://cswww.essex.ac.uk/Research/CSP/edward/FCS.html for availability), 1993.

[107] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[108] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[109] P. Vilím, R.Barták, and O. Čepek. Extension of o(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 10(4):403–425, 2005.

[110] D. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[111] J. P. Watson, L. Barbulescu, L. D. Whitley, and A. E. Howe. Contrasting structured and random permutation flow-shop scheduling problems: search-space topology and algorithm performance. *INFORMS Journal on Computing*, 14(2):98–123, 2002.

[112] J. P. Watson and J. C. Beck. A hybrid constraint programming / local search approach to the job-shop scheduling problem. In *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2008.

[113] J. P. Watson, A. E. Howe, and L. D. Whitley. Deconstructing Nowicki and Smutnicki's i-TSAB tabu search algorithm for the job-shop scheduling problem. *Computers and Operations Research*, 33(9):2623–2644, 2006.

[114] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition edition, 2005.

[115] D.H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82, 1997.

[116] H. Wu and P. van Beek. Restart strategies: Analysis and simulation. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, page 1001, 2003.

[117] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.