Learning during search

Alejandro Arbelaez Rodriguez

UNIVERSITÉ PARIS-SUD 11

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

THÈSE

DE L'UNIVERSITÉ PARIS–SUD

presentée en vue de l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ PARIS–SUD

Specialité: INFORMATIQUE

par

ALEJANDRO ARBELAEZ RODRIGUEZ

LEARNING DURING SEARCH

Soutenue le 31 Mai 2011 devant la commission d'examen :

Susan L. EPSTEIN,	Hunter College of The City University of New York	(Rapporteur)
Youssef HAMADI,	Microsoft Research	(Directeur de thèse)
Abdel LISSER,	LRI	(Examinateur)
Eric MONFROY,	Université de Nantes / Universidad de Valparaiso	(Invité)
Barry O'SULLIVAN,	University College Cork	(Rapporteur)
Frédéric SAUBION,	Université d'Angers	(Examinateur)
Thomas SCHIEX,	INRA	(Examinateur)
Michèle SEBAG,	CNRS / LRI	(Directeur de thèse)

Microsoft Research - INRIA Joint Centre
28 rue Jean Rostand 91893 Orsay Cedex, France

UNIVERSITÉ PARIS-SUD 11

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

PHD THESIS

by

ALEJANDRO ARBELAEZ RODRIGUEZ

LEARNING DURING SEARCH

Microsoft Research - INRIA Joint Centre
28 rue Jean Rostand 91893 Orsay Cedex, France

Acknowledgements

First, I would like to express my gratitude to my supervisors Youssef Hamadi and Michèle Sebag for their continuous guidance and advice during this time. I would also like to thank Marc Schoenauer for giving me the opportunity of being part of the TAO research group.

Special thanks also go to Camilo Rueda and Nestor Cataño who supported my application to the PhD at the Microsoft Research – INRIA joint centre. My gratitude also goes to Susan L. Epstein, Abdel Lisser, Eric Monfroy, Barry O'Sullivan, Frédéric Saubion, and Thomas Schiex, for being part of the jury committee.

I would like to thank my friends who made my stay in France more enjoyable, in particular, at the university: Alvaro Fialho, Romaric Gaudel, Raymond Roos, Luis Da Costa, Ibrahim Abdoulahi, Daniel Ricketts, and Said Jabbour. Also outside the research lab: Carlos Olarte, Andres Aristizabal, Luis Pino, Frank Valencia, Hélène Collado, Silway Maliszewska, Tatiana Hernandez, and Carlos Acuña. And all the people that used to play table-tennis with me here in France: Pierre Monceaux, Julien Cledera, Albin Andrieux, Sylvain Meurgue, Thomas Treal, Sébastien Gadelle, and Sébastien Barthelemi just to name a few. I have got to say that I will miss playing the French competition on Saturdays.

I also want to express my gratitude to the members of the AVISPA research group in Cali-Colombia, as well as, many thanks to Alberto Delgado, Julian Gutierrez, Jorge Pérez, Daniel Ricketts, and Carlos Olarte for proofreading some chapters of this thesis.

Last but not least, I would like to thank from the bottom of my heart my family, my parents Juan Bautista and Yolanda, my brothers Gilma and Juan Carlos, and my little nephew Stephan for all their support and encouragement in so many ways during all my life.

Alejandro Arbelaez Rodriguez,
Paris, May 2011

Abstract

Autonomous Search is a new emerging area in Constraint Programming, motivated by the demonstrated importance of the application of Machine Learning techniques to the Algorithm Selection Problem, and with potential applications ranging from planning and configuring to scheduling. This area aims at developing automatic tools to improve the performance of search algorithms to solve combinatorial problems, e.g., selecting the best parameter settings for a constraint solver to solve a particular problem instance. In this thesis, we study three different points of view to automatically solve combinatorial problems; in particular Constraint Satisfaction, Constraint Optimization, and SAT problems.

First, we present domFD, a new Variable Selection Heuristic whose objective is to heuristically compute a simplified form of functional dependencies called weak dependencies. These weak dependencies are then used to guide the search at each decision point. Second, we study the Algorithm Selection Problem from two different angles. On the one hand, we review a traditional portfolio algorithm to learn offline a heuristics model for the Protein Structure Prediction Problem. On the other hand, we present the Continuous Search paradigm, whose objective is to allow any user to eventually get his constraint solver to achieve a top performance on their problems. Continuous Search comes in two modes: the functioning mode solves the user's problem instances using the current heuristics model; the exploration mode reuses these instances to training and improve the heuristics model through Machine Learning during the computer idle time. Finally, the last part of the thesis, considers the question of adding a knowledge-sharing layer to current portfolio-based parallel local search solvers for SAT. We show that by sharing the best configuration of each algorithm in the parallel portfolio on regular basis and aggregating this information in special ways, the overall performance can be greatly improved.

Contents

1 Introduction **1**

1.1 Motivation and Context 1

1.2 Constraint Programming 2

1.3 Contributions of this thesis 3

1.4 Thesis outline 4

1.5 List of Publications 6

2 Formalism **9**

2.1 Constraint Satisfaction Problems 9

2.2 Complete Search 10

2.2.1 Variable and Value ordering 15

2.3 Incomplete Search 16

2.4 The Propositional Satisfiability Problem 19

2.4.1 Variable Selection 19

2.5 Constraint Optimization Problems 22

2.6 Supervised Machine Learning 24

2.6.1 Support Vector Machines 24

2.6.2 Decision Trees 26

3 Related work **29**

3.1 The Algorithm Selection Problem 29

3.1.1 Portfolios for SAT 30

3.1.2 Portfolios for QBF 34

3.1.3 Portfolios for CSP . . . 35
3.2 Portfolios for Optimization problems . . . 37
3.3 Per class learning . . . 39
3.4 Adaptive Control . . . 41
3.5 Other work . . . 43

4 Exploiting Weak Dependencies in Tree-based Search 47
4.1 Introduction . . . 47
4.2 Constraint propagation . . . 48
4.3 Exploiting weak dependencies in tree-based search . . . 50
4.3.1 Weak dependencies . . . 50
4.3.2 Example . . . 50
4.3.3 Computing weak dependencies . . . 52
4.3.4 The domFD dynamic variable ordering . . . 53
4.3.5 Complexities of domFD . . . 54
4.4 Experiments . . . 55
4.4.1 The problems . . . 56
4.4.2 Searching for all solutions or for an optimal solution . . . 58
4.4.3 Searching for a solution with a classical branch-and-prune strategy . 58
4.4.4 Searching for a solution with a restart-based branch-and-prune strategy . . . 59
4.4.5 Synthesis . . . 62
4.5 Previous work . . . 63
4.6 Summary . . . 64

5 Building Portfolios for the Protein Structure Prediction Problem 65
5.1 Introduction . . . 65
5.2 The protein structure prediction problem . . . 66
5.3 Features . . . 68
5.3.1 Problem features . . . 68
5.3.2 CP features . . . 69
5.4 Algorithm portfolios . . . 70

5.4.1 Algorithm subset selection . . . 72
5.5 Experiments . . . 73
5.6 Summary . . . 80

6 Continuous Search in Constraint Programming 83
6.1 Introduction . . . 83
6.2 Continuous Search in Constraint Programming . . . 85
6.3 Dynamic Continuous Search . . . 86
6.3.1 Representing instances: feature definition . . . 87
6.3.1.1 Static features . . . 88
6.3.1.2 Dynamic features . . . 88
6.3.2 Feature pre-processing . . . 89
6.3.3 Learning and using the heuristics model . . . 89
6.3.4 Generating examples in Exploration mode . . . 90
6.3.5 Imbalanced examples . . . 91
6.4 Experimental validation . . . 92
6.4.1 Experimental setting . . . 92
6.4.2 Practical performances . . . 94
6.4.3 Exploration time . . . 99
6.4.4 The power of adaptation . . . 99
6.5 Previous Work . . . 101
6.6 Summary . . . 102

7 Efficient Parallel Local Search for SAT 105
7.1 Introduction . . . 105
7.2 Knowledge Sharing in Parallel Local Search for SAT . . . 106
7.2.1 Using Best Known Configurations . . . 107
7.2.2 Weighting Best Known Configurations . . . 107
7.2.2.1 Ranking . . . 108
7.2.2.2 Normalized Performance . . . 108
7.2.3 Restart Policy . . . 108
7.3 Experiments . . . 109

7.3.1 Experimental Settings 109
7.3.2 Practical Performances with 4 Cores 110
7.3.3 Practical Performances with 8 Cores 114
7.3.4 Analysis of the Diversification/Intensification Trade off 116
7.3.5 Analysis of the Limitations of the Hardware 118
7.4 Previous Work 123
7.4.1 Complete Methods for Parallel SAT 123
7.4.2 Incomplete Methods for Parallel SAT 124
7.4.3 Cooperative Algorithms 125
7.5 Summary 126

8 Conclusions and Perspectives **127**
8.1 Overview of the main contributions 127
8.2 Perspectives 128

Bibliography **131**

List of Figures

2.1 Sudoku 11
2.2 Resulting search tree for the sudoku example 13
2.3 Sudoku resolution step-by-step (Backtracking) 14
2.4 Sudoku resolution step-by-step (Local Search) 18
2.5 SAT example 19
2.6 Linear Support Vector Machine. The optimal hyperplane is the one maximizing the minimal distance to the examples. 25
2.7 Decision tree example 26

3.1 Rice's abstract model for the algorithm selection problem 30

4.1 Weak dependencies 51
4.2 Variables and propagators 52

5.1 3D conformation of the 3SDHA protein 67
5.2 Traditional algorithms portfolio framework 71
5.3 Experimental validation using 10-fold cross-validation and an inner forward selection 74
5.4 *wdeg*Vs $wdeg^{+}$ 75
5.5 Experimental evaluation using all available heuristics 78
5.6 Experimental evaluation using forward heuristic selection 79
5.7 $\langle$ALL, cp+bio$\rangle$ Vs $\langle$FS, cp+bio$\rangle$ 80

6.1 Continuous Search scenario 85

6.2 *dyn-CS*: selecting the best heuristic at each restart point 86
6.3 Continuous Search in Constraint Programming 87
6.4 Langford-number (lfn) 94
6.5 Geometric (geom) 95
6.6 Balance incomplete block designs (bibd) 95
6.7 Job Shop (js) 96
6.8 Nurse Scheduling (nsp) 96

7.1 Performance using 4 cores in a given amount of time 112
7.2 Runtime comparison, each point indicates the runtime to solve a given instance using *4cores-Prob* (y-axis) and *4cores-No Sharing* (x-axis) 113
7.3 Best configuration cost comparison on unsolved instances. Each point indicates the best configuration (median) cost of a given instance using *4cores-Prob* (y-axis) and *4cores-No Sharing* (x-axis) 113
7.4 Performance using 8 cores in a given amount of time 115
7.5 Pairwise average Hamming distance (x-axis) vs Number of flips every 10^6 steps (y-axis) to solve the *unif-k3-r4.2-v16000-c67200-S2082290699-014.cnf* instance 119
7.6 Individual algorithms performance to solve the *unif-k3-r4.2-v16000-c67200-S2082290699-014.cnf* instance 121
7.7 Runtime comparison using parallel local search portfolios made of respectively 1, 4, and 8 identical copies of PAWS 122

List of Tables

4.1 All solutions and optimal solution 58
4.2 First solution, branch-and-prune strategy 60
4.3 First solution, restart-based strategy 61
4.4 Synthesis of the experiments 62

5.1 Amino-acid feature's group 69
5.2 Overall strategies solution cost with a 5-minute timeout 76

6.1 Total solved instances (5 Minutes) 97
6.2 Total solved instances (3 Minutes) 98
6.3 Predictive Accuracy of the heuristics model (10-fold Cross Validation) . . . 98
6.4 Exploration time in Hours (time-out 3 Minutes) 99
6.5 Exploration time in Hours (time-out 5 Minutes) 99
6.6 Total solved instances (5 Minutes) 100

7.1 Overall evaluation using 4 cores 114
7.2 Overall evaluation using 8 cores 116
7.3 *Diversification-Intensification* analysis using 8 cores 120

Chapter 1

Introduction

1.1 Motivation and Context

Nowadays, it is widely recognized that selecting the right algorithm for a given problem might considerably increase the overall performance to solve complex combinatorial problems. This is because, in general no algorithm outperforms all others on all possible problems. In order to understand this more precisely, we recall the *No Free Lunch Theorem* (NFL) [WM97] which states that without particular knowledge about a given class (or distribution) of problems, it is not possible to establish that a given algorithm is on average better than any other.

> *if an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems.*
>
> –Wolpert & Macready [WM97].

Interestingly, in the 70's (nearly 20 years ahead of the NFL) Rice [Ric76] introduced a framework for the *Algorithm Selection Problem*. Broadly speaking, this framework attempts to use Machine Learning to build a heuristics model. Such a model is mainly a function $f(x)$ which maps a given instance x into an algorithm $h_i \in \{h_1, \ldots, h_n\}$, where h_i is the most suitable algorithm to solve x, based on some performance criteria (e.g.,

runtime, solution cost, etc). In fact, in general the application of Machine Learning to efficiently solve combinatorial problems is part of an emerging area called *Autonomous Search* [HMS11, O'S10] whose objective is to automatically tune the parameters of a given algorithm in two directions: offline and self-adaptive tuning. The former is based on extensive preliminary experimentation to identify promising parameters, while in the latter case (also called *reactive search* [BB05]) the solver maintains an ongoing interaction with its environment to adapt on-the-fly the parameters of the search algorithm.

1.2 Constraint Programming

Constraint Programming (CP) is a powerful technique which allows the resolution of many combinatorial problems and is usually used as black-box for problem solving. That is, a user might only need to write down a model for his problem and push the "go" button to find a feasible solution. However, constraint solvers have been open since the beginning, and expose their parameters to a properly trained 'Constraint Programmer'. What seemed a correct standpoint in the 90's, at a time where the number of applications was pretty small, is seen today as a major weakness [Pug04].

In CP, properly crafting a constraint model capturing all constraints of a particular hard problem is often not enough to ensure an acceptable runtime performance. One way to improve performance is to use well-known techniques such as redundant and channeling constraints or to be aware that your constraint solver has a particular global constraint which can do part of the job more efficiently. The problem with these techniques (or tricks) is that they are far from obvious. Indeed, they do not change the solution space of the original modeling, and for a normal user (with a classical mathematical background), it might not be easy to understand why adding redundancy helps.

For this reason, *Autonomous* (or *Automated*) search has recently been established as

one of the main challenging areas in CP. This area consists in developing automatic techniques to tune the parameters of constraint solvers[1]. Broadly speaking, current work extends Rice's framework and consists of the following three-step procedure: (1) definition of a proper set of features (or problem descriptors), (2) using Machine Learning to train and learn a heuristics model based on a set of representative training instances, and (3) testing the accuracy of the learnt model on a set of unseen instances.

In addition, current work (see [SM08] and more in Chapter 3) can also be seen from two different perspectives. On the one hand, static portfolios (e.g., [PT07, LBNS02, GJK+10, GHBF05, HHHLB06]) use the *winner-takes-all* approach by selecting a single (the best) algorithm to solve a given instance. On the other hand, dynamic portfolios (e.g., [OHH+08, SM07, LL01, PT09, CB04]) refer to a current technology where the system maintains an ongoing interaction with its environment to identify on-the-fly the best algorithm for a given instance.

As pointed out earlier, there exist several approaches which have contributed to the current success of *Autonomous Search*. We would like to highlight SATzilla [XHHLB07] and CPHYDRA [OHH+08] which are pioneer portfolios in SAT and CP. In addition, the Quickest First Principle [BTW96] is a reference framework for developing dynamic portfolios, that is, where a set of pre-determined algorithms interleaves their execution in order to speed up the search.

1.3 Contributions of this thesis

In this thesis we study three different viewpoints to automatically solve combinatorial problems, in particular Constraint Satisfaction, Constraints Optimization, and SAT problems.

The first contribution of this thesis concerns the development of *domFD* (see Chapter 4) a novel dynamic variable selection heuristic which learns *Weak Dependencies* during the course of the search. *Weak Dependencies* are actually a simplified form of functional dependencies which represent relations between the variables of the problem and are used

[1] In this thesis, we assume that a constraint solver is a black-box technology whose parameters need to be carefully tuned to efficiently solve combinatorial problems. For instance, selecting the right search heuristic, constraint pruning level, restart strategy, etc.

to identify the most suitable variable at each choice point of the tree-based search algorithm.

The second contribution (see Chapter 5) refers to the *Algorithm Selection Problem*. Here, we explore the application of two different feature sets to build a portfolio algorithm for the Protein Structure Prediction Problem. One feature set is from the application domain and the other is from the CP abstraction of the problem. Moreover, in this chapter we also propose the use of forward selection to identify a proper subset of heuristics candidates to build the final heuristics model.

Furthermore, we propose the Continuous Search paradigm (see Chapter 6) which extends the typical point of view of the *Algorithm Selection Problem*. This new paradigm considers real-life situations where the user does not necessary dispose of a large number of training instances to train and learn a heuristics model. Instead, the heuristics model is set to its default parameter settings and it is enriched along a lifelong learning approach, exploiting the problem instances submitted by the user to the constraint solver.

Finally, our last contribution is devoted to a new parallel algorithm for the SAT problem (see Chapter 7). This parallel algorithm uses the well-known concept of parallel portfolio of algorithms, where several algorithms compete and cooperate to solve a given instance. In this context, each algorithm in the parallel portfolio exchanges the best configuration found so far, in order to carefully craft a new starting point.

1.4 Thesis outline

In this section, we describe the structure of the thesis, as well as the connection between the publications associated to the thesis with each chapter.

Chapter 2 introduces the basic concepts and terminology used in this thesis. We briefly describe Constraint Satisfaction Problems, Constraint Optimization Problems, the Satisfiability Problem, and the main algorithms to solve these kinds of problems. Including complete and incomplete search. Additionally, this chapter presents important concepts about Machine Learning.

Chapter 3 presents an extensive literature review on the *Algorithm Selection Problem* in the context of Constraint Satisfaction Problems, Constraint Optimization Problems, the Satisfiability Problem, and Quantified Boolean Formulas. This way, this chapter describes the *Algorithm Selection Problem* using four different abstractions: traditional portfolio algorithms, per class learning, adaptive algorithms and other work in the area.

Chapter 4 presents our first contribution *domFD*, a new variable selection heuristic, which aims to compute a simplified form of functional dependencies, so-called weak dependencies. These weak dependencies are used to rank decision variables and guide the search procedure. The main contribution of this chapter was published as [AH09b].

Chapter 5 investigates the use of Machine Learning techniques to build a portfolio algorithm by taking into account two different feature sets: features extracted directly from problem domain and features extracted from the CP abstraction of the problem. Moreover, forward selection is used to automatically select the best subset of algorithms to build the final heuristics model. Finally, the selection of the best algorithm is based on the solution cost of each heuristic in the portfolio after a given amount of time. The main contribution of this chapter was published as [AHS10a].

Chapter 6 introduces the Continuous Search paradigm whose objective is to exploit computer's idle time to incrementally build a heuristic model for the current distribution of problems. Unlike other work related to the algorithm selection problem (e.g., [SM08, XHHLB07, PT07, SM07, HW09b]) Continuous Search does not require a large number of representative training examples to train and build a heuristics model. The main contributions of this chapter are described as follows: [AHS09] presents the application of Support Vector Machines to select the best CSP heuristic at different states of the search, [AH09a] briefly introduces the Continuous Search paradigm, and [AHS10b, AHS11] fully details this new paradigm.

Chapter 7 describes a new parallel local search solver for SAT which extends the traditional parallel portfolio algorithm by adding a knowledge-sharing framework.

This way, each algorithm candidate exchanges its best configuration found so far, in order to carefully design a new starting point. The main contributions of this chapter were published as [AH11b, AH11a]

Chapter 8 presents general conclusions of this thesis and gives some directions for future work.

1.5 List of Publications

Most of the material of this thesis has been previously reported in the following peer-reviewed publications:

- Alejandro Arbelaez and Youssef Hamadi. Exploiting Weak Dependencies in Tree-Based Search. In *ACM Symposium on Applied Computing (SAC)*, pages 1385–1391, Honolulu, Hawaii, USA, March 2009. ACM. [AH09b]

- Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Building Portfolios for the Protein Structure Prediction Problem. In *Workshop on Constraint Based Methods for Bioinformatics (WCB)*, Edinburgh, UK, July 2010. [AHS10a]

- Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Online Heuristic Selection in Constraint Programming. In *International Symposium on Combinatorial Search*, Lake Arrowhead, USA, July 2009. [AHS09]

- Alejandro Arbelaez and Youssef Hamadi. Continuous Search in Constraint Programming: An Initial Investigation. In Karen Petrie and Olivia Smith, editors, *Constraint Programming Doctoral Program*, pages 7–12, Lisbon, Portugal, September 2009. [AH09a]

- Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous Search in Constraint Programming. In Eric Gregoire, editor, *22th International Conference on Tools With Artificial Intelligence (ICTAI)*, volume 1, pages 53–60, Arras, France, October 2010. IEEE. [AHS10b]

- Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous Search in Constraint Programming. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*. Springer-Verlag, 2011. [AHS11]
- Alejandro Arbelaez and Youssef Hamadi. Improving parallel local search for SAT. In *Learning and Intelligent Optimization, Fifth International Conference, LION 2011 (to appear)*, 2011. [AH11b]
- Alejandro Arbelaez and Youssef Hamadi. Efficient Parallel Local Search for SAT. *Submitted to JAIR*, 2011. [AH11a]

Chapter 2

Formalism

In the previous chapter, we have introduced and motivated the general objectives of this thesis. Now, in this chapter, we describe some basic concepts, such as Constraint Satisfaction Problems (CSPs), the Propositional Satisfiability Problem (SAT), and Constraint Optimization Problems (COPs). Additionally, we review notions on backtracking and local search, which are the most commonly used techniques to solve this kind of problems. The chapter concludes by presenting Supervised Machine Learning.

2.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a triple $\langle X, D, C \rangle$ where

- $X = \{X_1, X_2, \ldots, X_n\}$ represents a set of n variables.
- $D = \{D_1, D_2, \ldots D_n\}$ represents a set of associated domains (i.e., possible values for the variables).
- $C = \{C_1, C_2, \ldots, C_m\}$ represents a finite set of constraints.

Each constraint C_i involves a set of variables in X and is used to restrict the combinations of values between these variables. Constraints are often expressed as mathematical expressions such as $X_1 = X_2 + X_{10}$ or particular declarations such as `alldifferent(`

$[X_1, X_2, \ldots, X_n]$) indicating that variables X_1 to X_n must have different values. The degree of a variable $deg(X_i)$ indicates the number of constraints involving X_i and $dom(X_i)$ ($dom(X_i) = D_i$) denotes the current domain of a given variable X_i. In theory, the domain of a variable can initially take an infinite set of values, however in practice it is usually restricted to a finite set of numbers (e.g., $dom(X_i) \in [1..100]$)

Solving a CSP involves finding a solution, i.e., an assignment of values to variables such as all constraint are satisfied. If a solution exits the problem is stated as satisfiable and unsatisfiable otherwise. Currently, there are two well established techniques for solving CSPs, complete and incomplete techniques [BHZ06, RvBW06], the former is developed on top of a backtracking algorithm which combines a tree-based search with constraint propagation, while the latter is based on local search algorithms to quickly find an assignment for each variable that satisfies all constraints.

A well known CSP example is the sudoku problem [Sim05]. This problem consists in completing a pre-filled 9×9 matrix with numbers from $[1..9]$ such that every column, row, and 3×3 sub-matrix (see Figure 2.1) contain different values. A formal CSP definition of this problem is presented as follows:

$$\textit{Variables} = \begin{cases} X_{11} & X_{21} & \ldots & X_{91} \\ X_{12} & X_{22} & \ldots & X_{92} \\ \vdots & \vdots & \vdots & \vdots \\ X_{19} & X_{29} & \ldots & X_{99} \end{cases}$$

$$\textit{Constraints} = \begin{cases} \forall_{i \in [1..9]}\ alldifferent([X_{i1} \ldots X_{i9}]) & \textit{Columns} \\ \forall_{i \in [1..9]}\ alldifferent([X_{1i} \ldots X_{9i}]) & \textit{Rows} \\ \forall_{i,j \in [0..2]}\ alldifferent([X_{3i+1,3j+1} \ldots X_{3i+3,3j+3}]) & \textit{Sub-matrices} \end{cases}$$

$$\textit{Domains} = \quad \forall_{i,j \in [1..9]}\ D_{ij} \in [1..9]$$

2.2 Complete Search

Algorithm 2.1 depicts a depth-first search backtracking algorithm [Van06] widely used to tackle CSPs. The algorithm starts with the problem definition s and at each node of the

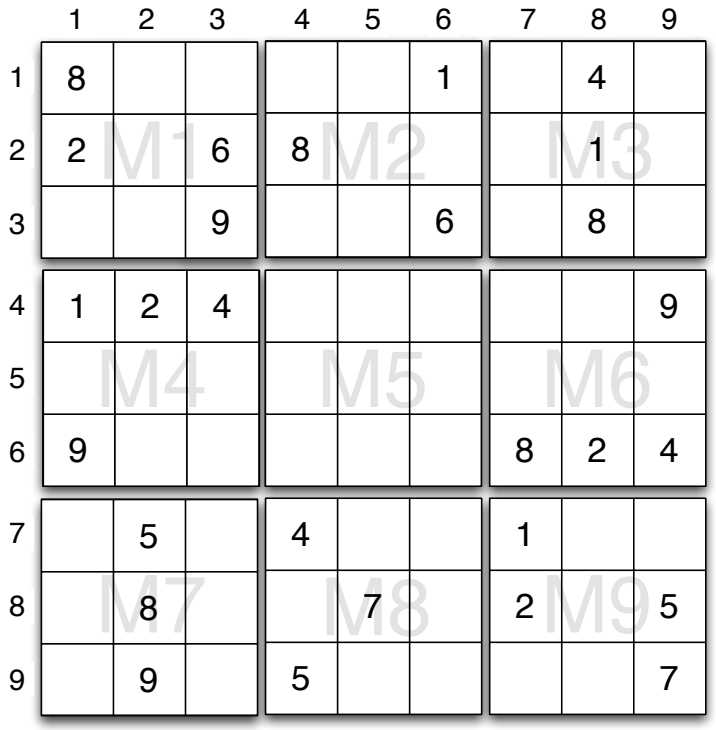


Figure 2.1: Sudoku

search *select-variable* (line 4) selects an unassigned variable x and *select-value* (line 5) selects a valid value for x, following that a constraint $x = v$ is added to the search process. In case of unfeasibility, the backtracking search can undo previous decisions (line 9) and add new constraints such as $x \neq v$ (line 10). The search thus explores a so-called search tree (i.e., binary search tree), where each leaf-node corresponds to a solution or an inconsistent assignment of values for the variables (i.e., Failure).

Clearly, in the worst-case scenario the search process requires to explore an exponential space. Therefore, it is necessary to combine the exploration of variable/value candidates with a look-ahead strategy to narrow the domains of the variables and reduce the remaining search space through constraint propagation.

Algorithm 2.1 backtracking(Problem s)

```
1: if s = SOLUTION or s = FAILURE then
2:    return s
3: end if
4: x ← select-variable(s)
5: v ← select-value(x)
6: add-constraint-and-propagate(x = v) to s
7: result ←backtracking(s)
8: if result = FAILURE then
9:    remove-constraint(x = v) from s
10:   add-constraint-and-propagate(x ≠ v) to s
11:   return backtracking(s)
12: end if
13: return result
```

In order to illustrate how the backtracking algorithm solves a CSP, let us consider the sudoku example presented in Figure 2.1[1]. In this example, we assume that the *alldifferent* constraint implements domain consistency (so-called generalized arc consistency) which means that for each value for all variables there exists an assignment of variables that satisfies the constraint. Additionally, let us assume that the variable selection function returns the first none assigned variable in the list of candidates and the value selection returns the minimum value in the domain.

[1] Instance 12 of the sudoku example in Gecode [Gec06]

Figure 2.2 shows a step by step execution of the algorithm, it starts by performing constraint propagation to remove inconsistent values for the variables (Figure 2.3(a), then the search starts and the constraint $X_{12} = 3$ is added (Figure 2.3(b)), the resulting state of the search is still unknown, so that $X_{42} = 7$ is also added (Figure 2.3(c)). As a results of this X_{65} has no support because its remaining values are $\{1, 3\}$ and those values are already assigned to X_{64} and X_{75}. The search backtracks to the previous step and tries $X_{42} \neq 7$ (Figure 2.3(d)) where the variable X_{95} is inconsistent with X_{45} and X_{82}, at this point the search backtracks to the root node to post $X_{12} \neq 3$ (Figure 2.3(e)) and $X_{13} = 3$ (Figure 2.3(f)), after posting the last constraint the constraint propagation engine detects an inconsistency among X_{48}, X_{46}, X_{44} and X_{58}, therefore the algorithm backtracks to the previous state to post $X_{12} \neq 3$ (Figure 2.3(g)) where a solution is finally obtained.

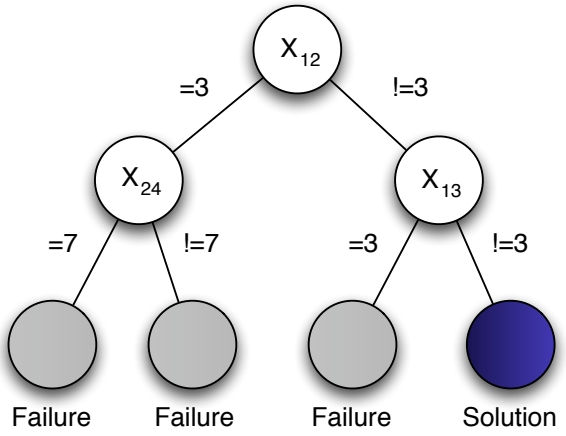


Figure 2.2: Resulting search tree for the sudoku example

The backtracking algorithm is usually equipped with a restart strategy which helps to reduce the effects of early mistakes in the search. Currently, in the literature there are several restart methodologies, among which the most important are: static restarts [GSK98] that implements a static policy by means of restart every c backtracks, geometric restarts [Wal99] that systematically increase the cutoff c by multiplying it by a constant factor (e.g., $\times 1.2$), Luby [LSZ93] defines the cutoff of each restart based on the sequence $Luby$={1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, ... }, which is formally defined as follows:

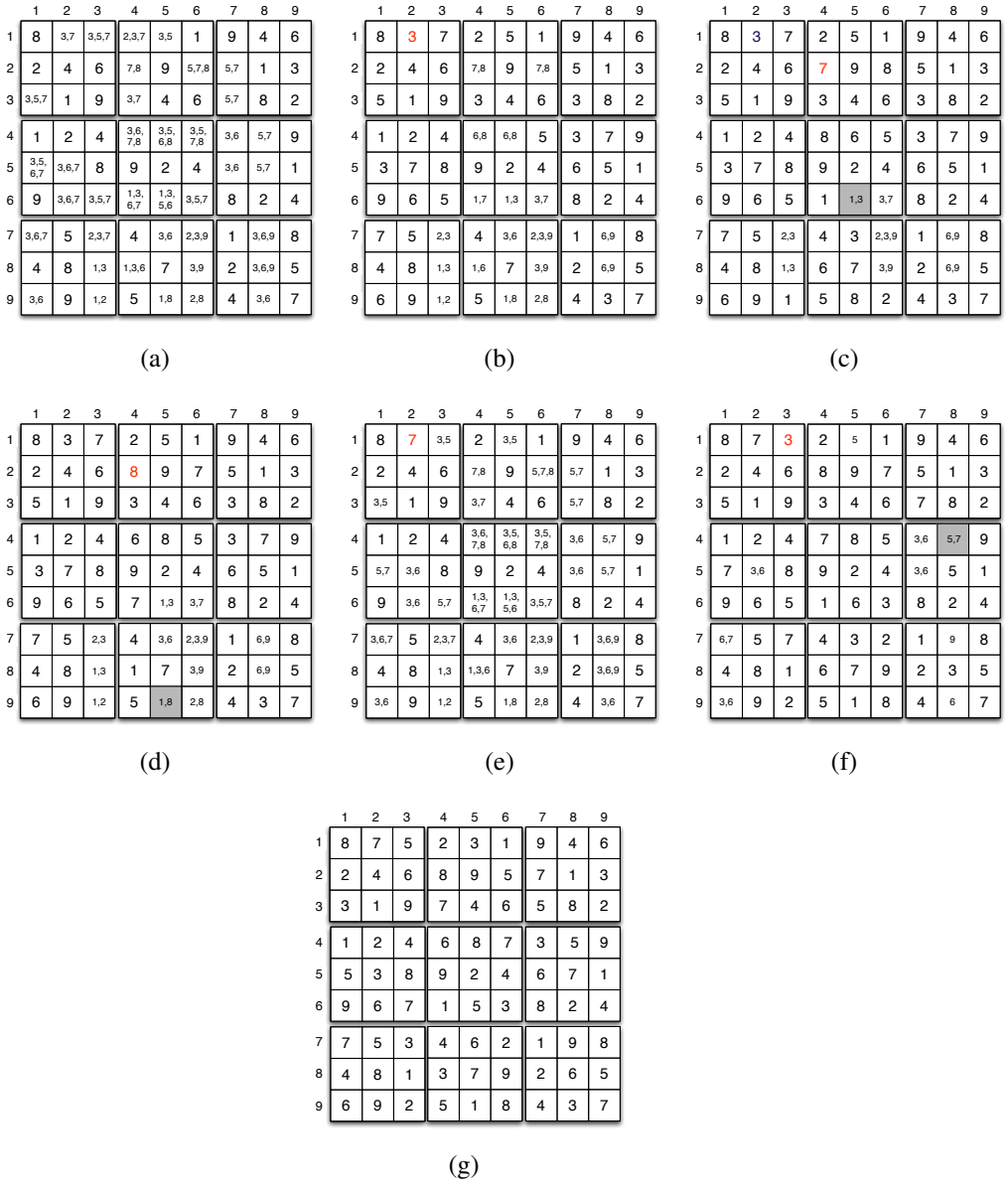


Figure 2.3: Sudoku resolution step-by-step (Backtracking)

$$Luby(i) = \begin{cases} 2^{k-1}, & \texttt{if } \exists k.i = 2^k - 1 \\ Luby(i - 2^{k-1} + 1), & \texttt{if } \exists k.2^{k-1} \leq i < 2^k - 1 \end{cases}$$

In this way, the cutoff of the ith restart is $p \times Luby(i)$, where p is a constant factor. Restarts are also an important component in nearly all complete SAT solvers as they have shown tremendous impact on the performance of the solvers [Hua07].

2.2.1 Variable and Value ordering

This section briefly reviews the basic ideas and principles behind the last generation of CSP heuristics. As pointed out above, variable/value selection heuristics are critical when solving CSPs and selecting the right combination of heuristics for a given problem-instance might considerably improve the overall performance. Classical value ordering strategies can be summarized as follows: *min-value* selects the minimum value, *max-value* selects the maximum value, *mid-value* selects the median value and *random-value* selects a random value from the remaining domain of a given variable.

On the other hand, variable selection heuristics are more important and comprehend more sophisticated algorithms. *lexico* is one of the simplest heuristics for variable selection, selecting the first unassigned variable in the list of decision variables. *random* selects an unassigned variable with a uniform distribution. *mindom* [HE79] is a well established CSP heuristic based on the "First-Fail Principle: try first where you are more likely to fail", this strategy chooses the variable with minimum size domain. *mindom* is usually used to complement more sophisticated heuristics such as *dom-deg*, which selects the variable that minimizes the ration $\frac{dom}{degree}$, where dom and $degree$ denote the size of the domain of a given variable and its respectively dynamic degree.

In [BHLS04], Boussemart et al., proposed *wdeg* and *dom-wdeg* to focus the search on difficult constraints. The former selects the variable that is involved in most failed constraints. A weight is associated to each constraint and incremented each time the constraint fails. Using this information *wdeg* selects the variable whose weight is maximal. The latter *dom-wdeg*, is a mixture of the current domain and the weighted degree of a variable,

choosing the variable that minimizes the ratio $\frac{dom}{wdeg}$.

In [Ref04], Refalo proposed the *impacts* dynamic variable-value selection heuristic. The rationale of impacts is to measure the size of the search space given by the Cartesian product of the domain of the variables (i.e., $|dom(x_1)| \times |dom(X_2)| \times \cdots \times |dom(x_n)|$). Using this information the impact of a variable is averaged over all previous decisions in the search tree and the variable with highest impact is selected.

2.3 Incomplete Search

Unlike the previously mentioned search algorithm that combines the variable/value selection process with constraint propagation to solve CSPs, incomplete methods are mainly based on *local search* algorithms to explore the search space. Algorithm 2.2 depicts a traditional local search algorithm used to solve CSPs. The algorithm starts with a random value assignment for each variable in the problem (*initial-configuration* line 2), and iteratively selects the best move (variable and value selection) that will most likely increase the chances of solving the CSP. *min-conflict* [MJPL92] is a well-known variable selection strategy in the context of local search, it firstly selects a random variable from an unsatisfied constraints and from that variable it chooses the value that minimize the number of failed constraints.

Algorithm 2.2 local search(Problem s)

```
1:  for try := 1 to Max-Tries do
2:      A := initial-configuration(s)
3:      for ite := 1 to Max-Iter do
4:          if A satisfies s then
5:              return s
6:          end if
7:          var := select-variable(A)
8:          val := select-value(var)
9:          A[var]:=val
10:     end for
11: end for
12: return 'No solution found'
```

One of the main drawbacks of local search is that it can quickly reach a local minimum and at this point no improvement can be easily achieved. To overcome this limitation, the algorithm is usually equipped with a *tabu list* [Gen03] that prevents the search of visiting previous observed states. Another strategy to avoid local minimum is the *random-walk* [SK93] method which adds noise to the variable/value selection process. This algorithm selects a random value from the selected variable with probability p and with probability $1 - p$ selects the value that minimizes the number of failed constraints. In addition, it is also a common practice to restart the local search algorithm with a new (fresh) random configuration after a given number of iterations (i.e., *Max-Iter*).

In order to illustrate the local search algorithm, let us consider again the sudoku instance described in Figure 2.1. In this example, we will assume that the variable selection process selects an unsatisfiable constraint and from this constraint it selects the best action (i.e., variable and value).

Let us assume that after completing with random values the pre-filled matrix we obtain the configuration observed in Figure 2.4(a), from this configuration a conflicting constraint is selected (blue constraint in Figure 2.4(a)). In this constraint it is observed that variables X_{41} and X_{52} are in conflict, and the best action would be replacing $X_{41} = 3$ since this move satisfies 3 constraints while changing X_{52} would only satisfy one. A similar behavior is obtained at the next iteration of the algorithm presented in Figure 2.4(b) by selecting $X_{34} = 3$. Finally at the last iteration of the algorithm the only two unsatisfied constraints are observed in Figure 2.4(c) and here the best action is to assign $X_{77} = 1$ to finally reach the solution depicted in Figure 2.4(d).

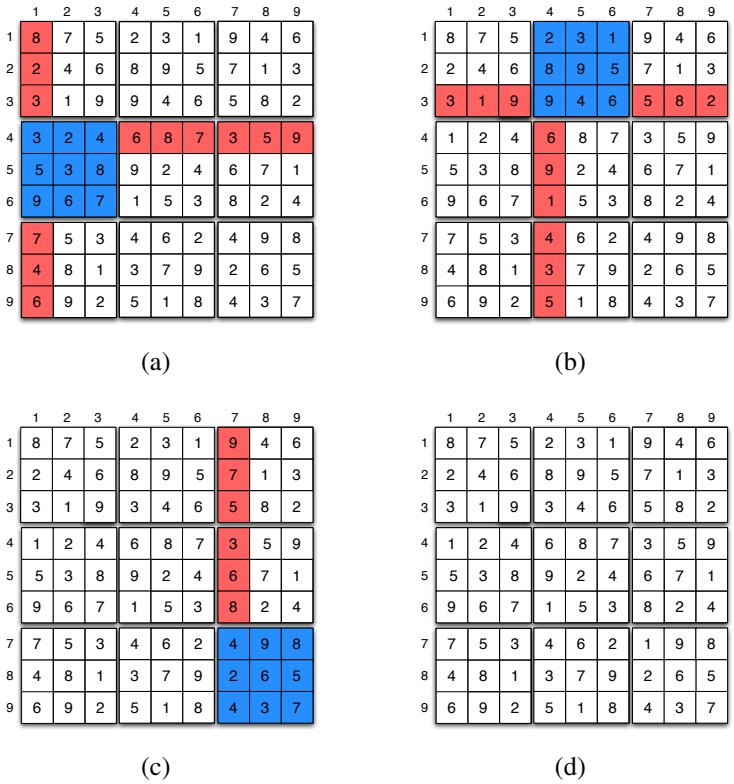


Figure 2.4: Sudoku resolution step-by-step (Local Search)

2.4 The Propositional Satisfiability Problem

The propositional Satisfiability Problem (SAT) is the first known NP-complete problem [Coo71]. It can be seen as a particular class of CSP represented by a pair $\langle \mathcal{V}, \mathcal{C} \rangle$, where $\mathcal{V}$ indicates a set of boolean variables and $\mathcal{C}$ a set of clauses representing a propositional *conjunctive-normal form* (CNF) formula. Figure 2.5 shows a SAT example described by means of the CNF formula. This problem is represented by a conjunction of clauses while each clause is a disjunction of literals (a variable or its negation), and the problem consists in finding a truth assignment for each variable such that all clauses are satisfied, or demonstrating that no such assignment can be found.

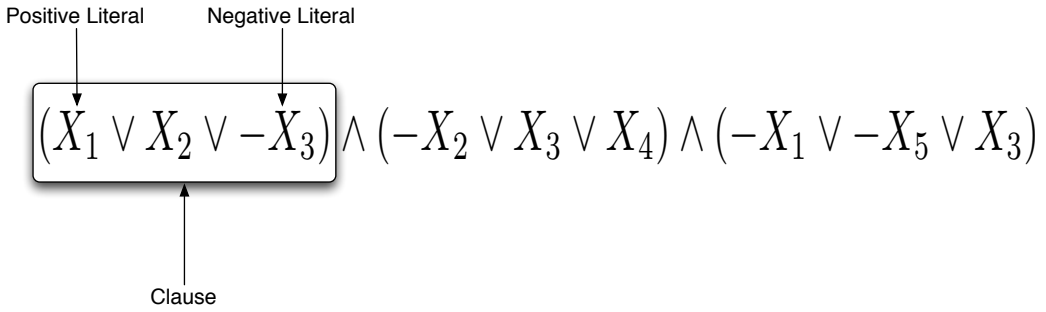


Figure 2.5: SAT example

Complete and Incomplete algorithms are also widely used to tackle SAT instances. The former is developed on top of the DPLL [DP60] algorithm and combines tree-based search with constraint propagation, conflict-clause learning, and intelligent backtracking. The latter is mainly developed on top of the local search algorithm described in Section 2.3, however, in this case at each iteration the main component is to flip the truth value of the selected variable. In the remaining of this section, we will focus on the main variable selection methods in the context of local search for SAT.

2.4.1 Variable Selection

This section briefly reviews the main characteristics of state-of-the-art local search solvers for SAT solving. As pointed out above these algorithms are developed to deal with the variable selection function (*select-variable* in Algorithm 2.2) which indicates the next variable

to be flipped in the current iteration of the algorithm. Broadly speaking, there are two main categories of variable selection functions, the first one motivated by the GSAT algorithm [SLM92] is based on the following score function:

$$score(x) = make(x) - break(x)$$

Intuitively $make(x)$ indicates the number of clauses that are currently satisfied but flipping x become unsatisfied, and $break(x)$ indicates the number of clauses that are unsatisfied but flipping x become satisfied. In this way, local search algorithms select the variable with minimal score value (preferably with negative value), because flipping this variable would most likely increase the chances of solving the instance. It is also important to notice that GSAT-like algorithms have been previously used in [MSG98] to guide the variable selection process of complete SAT solvers

The second category of variable selection functions is the WalkSAT-based one [SKC94] which includes a diversification strategy in order to avoid local minimums, this extension selects, at random, an unsatisfied clause and then picks a variable from that clause. The variable that is generally picked will result in the fewest previously satisfied clauses becoming unsatisfied, with some probability of picking one of the variables at random.

- *TSAT* [MSG97] extends the *GSAT* algorithm by proposing the use of a tabu list which contains a set of recently flipped variables in order to avoid flipping the same variables for a given number of flips. This way, the tabu list helps to scape from local minimums.

- *Novelty* [MSK97] firstly selects an unsatisfied clause c and from c selects the best v_{best} and second best v_{2best} variable candidates, if v_{best} is not the latest flipped variable in c then *Novelty* flips this variable, otherwise v_{2best} is flipped with a given probability p and v_{best} with probability *1-p*. Important extensions to this algorithm can be found in *Novelty+*, *Novelty++* and *Novelty+p*.

- *Novelty+* [Hoo99] with a given probability wp (random walk probability) randomly selects a variable from an unsatisfied clause and with probability *1-wp* uses *Novelty* to select the variable.

- *Novelty++* [LH05] with a given probability dp (diversification probability) flips the latest flipped variable from the selected unsatisfied clause and with probability *1-dp* uses *Novelty* to select the variable.

- *Novelty+p* [LWZ07] introduces the concept of promising score (*pscore*) for a given variable as follows:

$$pscore(x) = score_A(x) + score_B(x')$$

  where A is the current problem configuration (assignment for the variables), B is the configuration after flipping x, and x' the best promising decreasing variable with respect to B. Similarly to *Novelty*, *Novelty+p* starts by selecting v_{best} and v_{2best} from an unsatisfied clause c. Afterwards, if v_{best} is the latest flipped variable in c, then with a probability p selects v_{2best} and with probability *1-p* uses the promising score to select the next variable. Finally, if v_{best} is not the latest flipped variable in c but was flipped after v_{2best}, then v_{best} is selected, otherwise the promising score is used to select the best variable.

- *G*^{2}*WSAT* [LH05] (G2) maintains a list of promising decreasing variables to determine the most suitable variable to be flipped at each iteration of Algorithm 2.2, where a variable is decreasing if $score(x) > 0$. This way, *G*^{2}*WSAT* selects the best variable from the list, breaking ties using the flip history. If the list of decreasing variables is empty the algorithm uses *Novely++* as a backup heuristic. *G*^{2}*WSAT+p* (G2+p) uses a similar strategy that *G*^{2}*WSAT* however in this case the backup solver is *Novelty+p*.

- *Adaptive Novelty+* (AN+) [Hoo02] uses an adaptive mechanism to properly tune the noise parameter (wp) of walksat-like algorithms (e.g, *Novelty+*). This way, wp is initialized to 0 and if search stagnation is observed (i.e., no improvement has been observed for a while), then wp is incremented, i.e., wp=$wp + (1 + wp) \times \phi$. On the other hand, whenever an improvement is observed wp is decreased, i.e., wp=$wp - wp \times \phi/2$. This adaptive mechanism has shown impressive results, and is used to improve the performance of other local search algorithms in the context of SAT solving.

- *Scaling and Probabilistic Smoothing* (SAPS) [HTH02] adds a weight penalty to each

clause. These weights are initialized to 1 and updated during the search process. More precisely, as soon as a local minimum is reached SAPS implements a multiplicative increase rule to dynamically update the weight for unsatisfied clauses and at each step of the algorithm with a given probability P_{smooth} weights are adjusted according to a given smoothing factor ρ. Additionally, a local minimum is assumed when no improvement has been observed for a while.

- *Pure Additive Weighting Scheme* (PAWS) [TPBaFJ04] similarly to SAPS, each clause is associated with a weight penalty. However, in this case the authors implement an additive increase rule to dynamically modify the penalty for unsatisfied clauses and if a given clause penalty has been changed a given number of times this penalty value is adjusted.

- *Reactive SAPS* (RSAPS) [HTH02] extends SAPS by adding an automatic tuning mechanism to identify suitable values for the smoothing factor ρ.

- *Adaptive G^{2}WSAT* (AG2) [LWZ07] aims to integrate an adaptive noise mechanism into the *G^{2}WSAT* algorithm. Similarly, *Adaptive G^{2}WSAT+p* (AG2+p) also uses an adaptive noise mechanism into the *G^{2}WSAT+p* algorithm.

2.5 Constraint Optimization Problems

A *Constraint Optimization Problem* (COP) is basically a CSP with an objective function $f(X)$ to optimize. Unlike CSPs where we only need to explore the search space until a solution is found, solving a COP involves finding a solution (i.e., a valid value for all variables) and proving that the solution is optimal.

Algorithms 2.3 and 2.4 describe a well-known method used to solve COPs, here we assume without loss of generality a minimization problem. The algorithm starts with the *Search Optimization* method which sets the best solution found so far to $NULL$ and then executes a depth-first search branch and bound algorithm (*Search-BB*). This algorithm prunes (if possible) sub-optimal portions of the search by comparing the current state of the search with the best solution found so far (line 1). This pruning mechanic ensures

that every new solution is better than the previous best known one. After this point, the B&B algorithm behaves similar to the backtracking algorithm described above, selecting the most suitable variable/value pair (lines 8-9). However, in this case to prove optimality, the algorithm must explore the right and the left branches of the tree. Finally, it is worth mentioning that in the general case the variable/value selection heuristics used to solve COPs are mainly the same ones used for CSPs (see Section 2.2.1).

Algorithm 2.3 Search Optimization(Problem s, Objective f)

```
1: best ← NULL
2: Search-BB(s, f)
3: if best = NULL then
4:     return 'No Solution Found'
5: end if
6: return best
```

Algorithm 2.4 Search-BB(Problem s, Objective f)

```
1: if s = FAILURE or f(s) > f(best) then
2:     return
3: end if
4: if s = SOLUTION then
5:     best ← s
6:     return
7: end if
8: x ← select-variable(s)
9: v ← select-value(x)
10: add-constraint-and-propagate(x = v) to s
11: Search-BB(s,f)
12: remove-constraint(x = v) from s
13: add-constraint-and-propagate(x ≠ v) to s
14: Search-BB(s,f)
```

2.6 Supervised Machine Learning

Supervised Machine Learning exploits data labelled by the expert to automatically build hypotheses emulating the expert's decisions [Vap95]. Formally, a learning algorithm processes a training set $\mathcal{E} = \{(x_i, y_i), \ldots (x_n, y_n)\}$ where x_i is the example description (i.e., vector of features, $\Omega = \mathbb{R}^d$) and y_i is the associated output. The output can be a numerical value (i.e., regression) or a class label (i.e., classification). In this chapter, we limit our study to the classification case and each feature can be a categorical, continuous or discrete value.

The learning algorithm outputs a hypothesis $f : \Omega \mapsto Y$ associating to each example description a desirable output y. Among machine learning applications are pattern recognition, ranging from computer vision to fraud detection [LB08], predicting protein function [ASBG07], game playing [GS07], or autonomic computing [RBM+05]. In the following, we will describe Support Vector Machines and Decision Trees, two of the most popular machine learning algorithms for classification.

A common technique to evaluate the performance of a machine learning system is to use k-fold cross-validation [Koh95]. In k-fold cross-validation the entire dataset D is divided into k disjoint sets $(D_1, D_2, \ldots, D_k)$. For each subset $D_{i \in k}$, the hypothesis model is learn with $D - D_i$ and tested on D_i.

2.6.1 Support Vector Machines

Support Vector Machines (SVM) is a well-known machine learning technique for binary classification, that is, the associated label y_i is limited to two categories $\{-1, 1\}$. Linear SVM considers real-valued positive and negative instances ($\Omega = \mathbb{R}^d$) and constructs the separating hyperplane which maximizes the margin (Figure. 2.6), i.e. the minimal distance between the examples and the separating hyperplane. The margin maximization principle provides good guarantees about the stability of the solution and its convergence towards the optimal solution when the number of examples increases.

The linear SVM hypothesis $f(x)$ can be described from the sum of the scalar products between the current instance x and some of the training instances x_i, called support vectors:

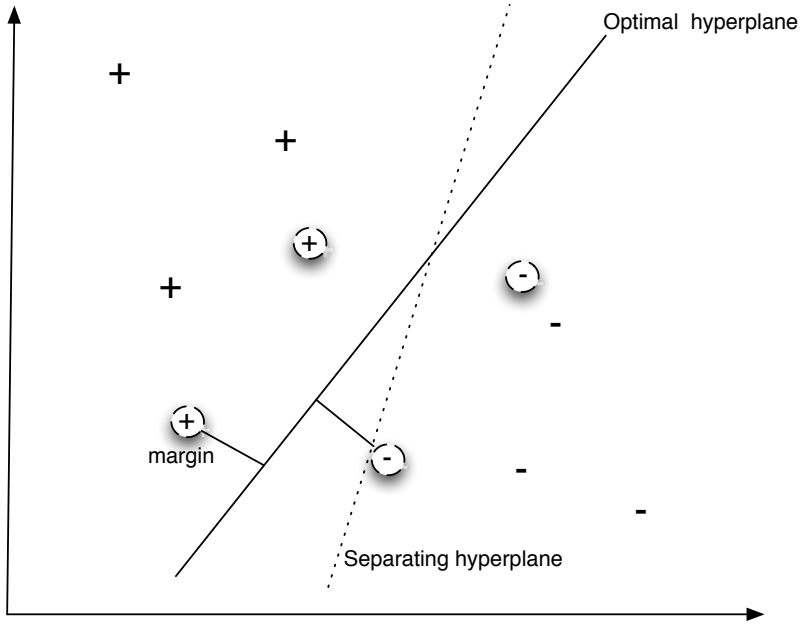


Figure 2.6: Linear Support Vector Machine. The optimal hyperplane is the one maximizing the minimal distance to the examples.

$$f(x) = < w, x > +b = \sum \alpha_i < x_i, x > +b$$

The SVM approach can be extended to non-linear spaces, by mapping the instance space Ω into a more expressive feature space $\Phi(\Omega)$. This mapping is made implicit through the so-called *kernel trick*, by defining $K(x, x') = < \Phi(x), \Phi(x') >$; it preserves all good SVM properties provided the kernel be positive definite. Among the most widely used kernels are the Gaussian kernel ($K(x, x')$ = $exp\{-\frac{||x-x'||^2}{\sigma^2}\}$) and the polynomial kernel ($K(x, x') = (< x, x' > +c)^d$). More complex separating hypotheses can be built on such kernels,

$$f(x) = \sum \alpha_i K(x_i, x) + b$$

using the same learning algorithm core as in the linear case. In all cases, a new instance x is classified as positive (respectively negative) if $f(x)$ is positive (resp. negative).

Although SVMs are developed for binary classification, this machine learning technique can also be used in the context of multi-class classification using two main strategies:

one-vs-all and *one-vs-one* [RK04]. The former creates N binary classifiers (one for each class) and to label a new example the classifier with largest score is selected. The latter creates all possible combinations of binary classifiers pairs, and to label a new example, all classifiers vote for a winning class.

2.6.2 Decision Trees

In decision tree learning the hypothesis model is represented in terms of a decision tree (see Figure 2.7) where each non-leaf node represents a given attribute (or feature), each branch represents the value of the node indicating a decision, and each leaf node represents a class label. A widely used algorithm to build the hypothesis model is the ID3 algorithm [Qui86] which uses a greedy technique to recursively select the best feature for each node in the tree.

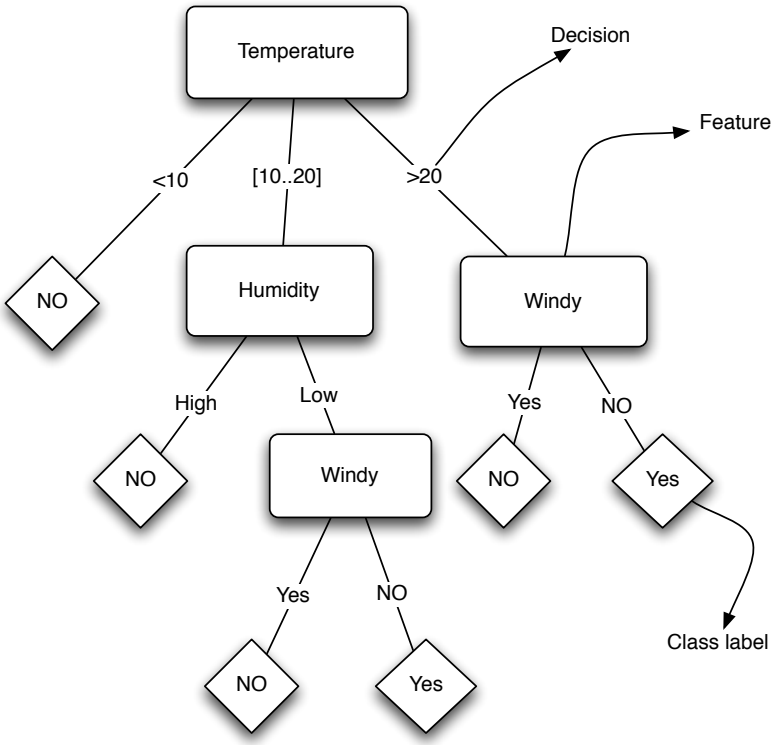


Figure 2.7: Decision tree example

Finally, Figure 2.7 illustrates a resulting decision tree example for the well-known tennis problem [Mit97], which consists in defining whether to play tennis based on the weather conditions. An instance is classified by looking the attribute indicated by the root node and moving down in the tree using the branching decisions. It is also worth mentioning that unlike SVMs, decision tree learning supports multi-class classification without using the composition of several independent models.

Chapter 3

Related work

In the previous chapter, we have presented general concepts about Constraint Programming and Machine Learning. Now, in this chapter, we review a wide variety of methods devoted to the Algorithm Selection Problem in the context of Constraint Satisfaction Problems, and related areas such as the Satisfiability Problem and Quantified Boolean Formulas. In particular, we discuss how up to now this problem has been explored from different perspectives, including: *portfolio algorithms*, which select the most appropriate algorithm for a given problem instance; *per-class learning*, which aim to select the most appropriate parameter configuration for a given algorithm on a given class of problems; *adaptive algorithms*, which internally adapt the parameters of a given algorithm based on problem changing conditions. We also review some other work developed in the area.

3.1 The Algorithm Selection Problem

The study of the algorithm selection problem goes back to the seminal work of Rice in [Ric76] who proposed an abstract model to select the most suitable algorithm for a given instance taking into account some performance criteria. Figure 3.1 depicts the general framework of the proposed model. The basic idea behind the scheme is that each problem instance x in the problem space P is represented by a set of problem features $f(x) \in F$, where f is a function intended to extract the feature vector from x and F represents the feature space.

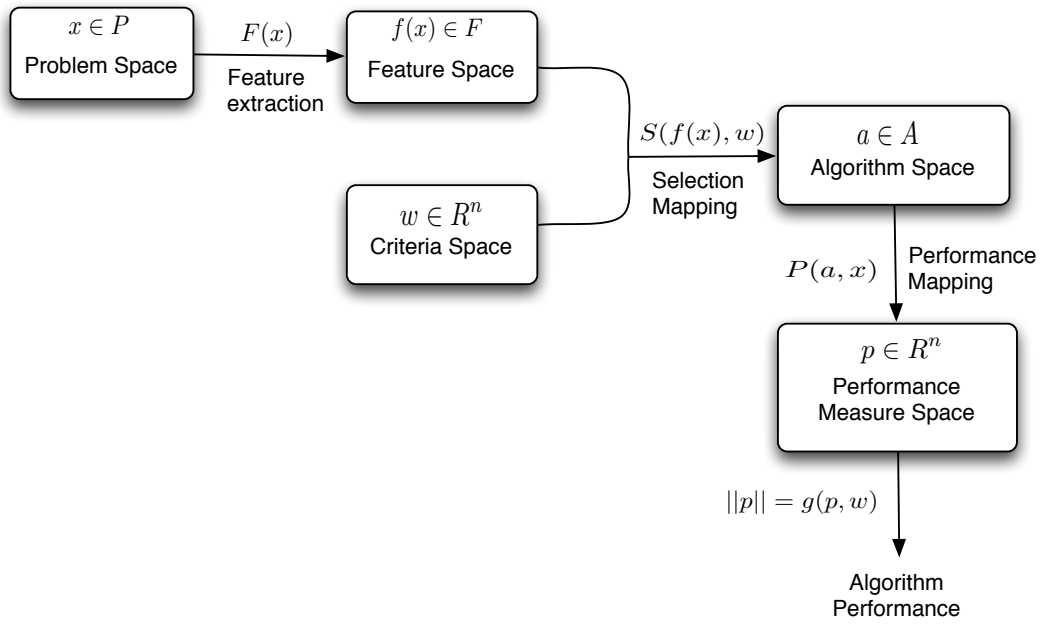


Figure 3.1: Rice's abstract model for the algorithm selection problem

Additionally, the selection function $S(f(x), w)$ takes as an input the feature vector f and a set of user defined performance criteria (or criteria space) w which indicates a family of performance metrics (e.g., algorithm's runtime, solution quality, etc), and returns the most suitable algorithm a whose expected performance is indicated by p. Notice that p is a n-dimensional vector, where each element in the vector indicates a given performance criterion. Finally in order to identify a single performance measure for a given algorithm a the norm value $||p|| = g(p, w)$ is calculated such that $p = P(a, w)$.

3.1.1 Portfolios for SAT

Over the last 15 years, the study of the application of Machine Learning to build a portfolio algorithm for the satisfiability problem has become a hot topic in the AI community. Early studies were devoted to dividing the expected runtime of a given algorithm into different categories. However, the current state-of-the-art portfolios focus on predicting the actual runtime of each algorithm candidate.

In [HRG+01] Horvitz et al., proposed a supervised machine learning approach to characterize the runtime of a given algorithm on a set *Quasigroup* instances. In an offline training procedure each training instance is described by means of a set of features or attributes such as: "ratio between unassigned variables and the size of the problem", "variance of unassigned variables across all rows and columns", "average of explored nodes", "average depth of the search tree", etc. This feature set is used to train a bayesian learning algorithm that will classify the runtime needed to solve a new problem instance into two main classes: *short* and *long*. *short* (resp. *long*) states that the solving time of a given instance is less (resp. *greater*) than the median time required to solve the entire training set. This information can later be used to discard low quality heuristics when solving new instances.

Later in [NLBH+04, LBNS02, LBNS09] Nudelman et al. instead of characterizing the runtime to solve a SAT problem, faced the challenging task of predicting its runtime. To this end, the so-called *Empirical Hardness Model* methodology is proposed to estimate the runtime of a given algorithm to solve a given instance. *Empirical Hardness Model* consists of the following 5-steps procedure to build a linear regression model. This model will work as runtime predictor for a given algorithm based on a set of problem descriptors (so-called features)

1. Select a problem distribution.
2. Select a set of suitable features.
3. Compute the runtime and feature values for all training examples.
4. Perform feature selection in order to use the most informative subset of features, x
5. Use linear regression to learn $f(x)$ a runtime function prediction.

This methodology is general enough to build a runtime predictor for any algorithm. Among the most important steps are the identification of problem features and building the linear regression model. The feature extraction is an important step and it requires highly experimented users to come with an appropriate feature vector to fully describe the problem. For instance, in [LBNS02, LBNS09] features associated to the *winner determination*

problem are discussed and analyzed in the context of *Empirical hardness model*. Additionally, the learnt runtime function is defined as $f(x) = w^\top \phi$, where ϕ is the vector of features and w is a set of free variables which will be computed using ridge regression [Bis06].

Due to the high success of *Empirical Hardness model*, this methodology is also applied in SATzilla [XHHLB07, XHHLB08] state-of-the-art portfolio for SAT solving. SATzilla uses a set of features discussed in [NLBH$^+$04], those features comprehend general information about SAT problem such as: number of variables, clauses, fraction of Horn clauses, number of unit propagations, etc. Broadly speaking, the architecture of SATzilla is divided into two main phases: training and testing. During the training phase a set of representative training samples (or instances) are then required, and based on those instances a set of potential SAT solvers is identified as well as a set of *pre-solvers*. During the testing phase, *pre-solvers* are executed for a short pre-solving time and if no solution is obtained during the pre-solving time, the algorithm with minimal expected runtime is executed. It is also worth mentioning that SATzilla has shown impressive results during the two latest SAT competitions (2007[1] & 2009[2]) where this portfolio algorithm won 10 medals.

SATzilla has been extended in many directions. Firstly, incorporating a *Mixture-of-experts* model, where the machine learning model combines the decision of two or more learners to improve the overall accuracy of the learnt system. In this way, [XHLB07] uses a Sparse Multinomial Logistic technique to compute the probability that a given instance is SAT or UNSAT, this information is then used to weight the previously mentioned linear regression model. In the same direction, Haim and Walsh [HW09b] also combine the decision of a logistic and linear regression models to build a portfolio for SAT, however, in this case the portfolio takes into account several restarts policies for a set of well-known SAT solvers. At this point, it is also worth mentioning [DO08] where Devlin and O'Sullivan show that we can use traditional classification algorithms (e.g., Random Forest, Decision Trees, k-Nearest Neighbor) to predict with high level of confidence whether a given SAT formula is SAT or UNSAT.

Another important contribution to SATzilla is presented in [HHHLB06, HH05] where Hutter et al., extended SATzilla to deal with the parameter tuning problem. In this context,

[1] www.satcompetition.org/2007
[2] www.satcompetition.org/2009

the goal is to use the SATzilla framework to identify promising parameter configurations for local search algorithms for SAT. The learnt runtime predictor is slightly modified to include two inputs $f(x, y)$, where x is the vector of SAT features and y is a given parameter configuration. Thus, if the number of parameter configurations is long enough, once a new instance arrives one might select the configuration with minimal expected runtime, otherwise this approach can be used to identify a robust configuration during the training phase.

Most recently, Xu et al., proposed *hydra* [XHLB10] to build a portfolio for SAT considering highly parametrized algorithms. Thus, a solver candidate is a given algorithm running with a given parameter configuration. *hydra* is a robust methodology which begins with a given algorithm and then iteratively adds new solvers to the portfolio until a given timeout is reached or after a given number of iterations. More precisely, at every iteration a parameter tuning tool (e.g., [HHLBS09]) is used to identify a new solver candidate. Once the new solver is available, the portfolio is re-computed taking into account the latest obtained algorithm. Notice that the portfolio construction removes useless solvers that do not help to improve the performance, so that although new solvers are iteratively added, not all of them are necessarily considered in the portfolio.

Unlike previous methods that estimate the runtime of a given algorithm before actually solving the problem, Haim and Walsh studied a slightly different point of view in [HW08, HW09a], proposing the *Linear Model Prediction* (LMP) method. This method aims to build a runtime prediction function which determines the remaining time from the current state until the end of the search, this way, the estimation is available at any time. To this end, the feature vector includes information regarding past performance of the search. For instance, learnt clauses size, conflict clause size, average clause size, search depth, etc., and the construction of the runtime function is defined using linear regression as previously stated for *empirical hardness models*.

Finally, another interesting application of Machine Learning to the algorithm selection problem in the context of SAT is presented in [LL01] where Lagoudakis and Littman proposed the use of reinforcement learning [SB98] to select branching rules for SAT. Although this work presented important results, the learning function is defined by considering only the total number of variables at a given state of the search tree.

3.1.2 Portfolios for QBF

This section is devoted to the use of machine learning to deal with the algorithm selection problem in the context of *Quantified Boolean Formulas* (QBF) [BM08].

Unlike SATzilla, which selects the algorithm with minimal expected runtime before actually solving the problem, Samulowitz and Memisevic [SM07] proposed an adaptive solver, where at each node of the search tree a classification algorithm (i.e., multinomial logistic regression) is used to predict the best algorithm h_{best}. After h_{best} is obtained, it is used to choose the most promising variable. It is important to notice that although the best algorithm is dynamically selected while solving a given problem, the winner heuristic is determined by applying each single algorithm to solve the entire problem instance during the training phase. Therefore, this work requires an important number of training instances to build a model with good generalization properties.

Pulina and Tacchella in [PT07] studied the application of four well-known machine learning techniques (Decision trees, 1-nearest neighbor, decision rules and logistic regression) to implement *AQME* (Adaptive QBF multi-engine) a portfolio for QBF. As opposed to previous work on QBF, this approach uses *AQME* to select the most appropriate algorithm to solve a given QBF instance and this algorithm is used for the entire search process. Interestingly, the performance of *AQME* with all learning techniques was considerably better than individual QBF solvers. However, as one could have been expected none of the learning techniques is superior than the others for all the experimental scenarios.

Another important contribution in the context of QBF solving is presented in [PT09] where Pulina and Tacchella detailed *self-AQME*, which extends previous work on *AQME* with a new training algorithm to deal with several distributions of problems. *self-AQME* works in rounds, at each round an execution sequence $[(t_1, S_1), (t_2, S_2), \ldots, (t_n, S_n)]$ is obtained, where t_i indicates the time cutoff for the ith solver in the sequence. The predicted best solver is always placed first in the sequence and the remaining solvers candidates are sorted with a given criteria (e.g., performance in previous QBF competitions), notice that $(t_1,t_2, \ldots, t_n)$ values might change between rounds. This procedure is repeated until a solution is found or a given global timeout is reached, if a solution is obtained with a non-expected solver s (i.e., $s \neq$ predicted best solver) then a new training example $\langle I, s\rangle$ is

added to the classification model, where I represents the feature vector associated to the problem instance and s indicates the solver that found the solution.

self-AQME was experimented using the following three sorting criteria: *Trust the Predicted Engine* (TPE) grants a given L time cutoff for the first round by means of executing all available algorithms with a timeout of $\frac{L}{N}$, where N indicates the number of candidates. If no solution is found during the first round, the predicted best solver is executed with all the remaining time budget. *All engines are the same* (AES) equally divides the global time budget for each solver candidate, notice that AES will always solve the same number of instances no matter the learning technique (even with random guesses). *Increasing the time round-robin* (ITR), all solver candidates are executed with a given ρ time cutoff at each round, and ρ is exponentially increased after finishing a round. Interestingly, using QBF competitions settings[3], TPE exhibited the worst performance among the three methodologies, and AES showed the overall best performance in terms of number of solved instances. However, when drastically decreasing the overall global time budget, TPE becomes a very effective algorithm, followed by ITR and AES was the worst. On the other hand, more recently, Stern et al., [SSH+10] used a machine learning technique called MatchBox [SHG09] to improve the performance of the TPE strategy.

3.1.3 Portfolios for CSP

Nowadays, there is an important number of methodologies for building portfolios for SAT and QBF solving, however few efforts have been devoted to CSP. An alternative explanation lies in the fact that CSPs are more diverse than SAT instances; SAT instances only involve boolean variables and clauses, contrasting with CSPs using variables with large domains, and a variety of constraints and pruning rules [BCDP07, BHZ06, PBG05].

Gebruers, et al., in [GHBF05] studied the application of two well known classification algorithms such as: *decision trees* (C4.5) and *k-nearest neighbor* (3-NN) to select the most suitable strategy for the Social Golfer problem [AV06]. In this context a strategy is a tuple $\langle$*model*, *variable selection*, *value selection*$\rangle$, where *model* indicates the set of constraints used to codify the problem and *variable selection* (resp. *value selection*) chooses a variable

[3] http://www.qbflib.org/index_eval.php

(resp. value) during the tree search procedure. Overall the experiments, 3-NN exhibited the best performance by frequently selecting the most appropriate strategy. It is important to note that this portfolio uses features and heuristics only applicable to the social golfer problem.

On the other hand, to the best of our knowledge the most remarkable work in the context of CSPs is the CPHYDRA solver [OHH+08]. CPHYDRA is a portfolio-like algorithm that exhibited the overall best performance in the 2008 CSP competition[4]. Broadly speaking, CPHYDRA is a portfolio algorithm developed upon case-based reasoning; it maintains a database with all solved instances (so-called *cases*). Later on, once a new instance I arrives a set of similar cases C is computed and based on C it builds a switching policy to select (and execute) a set of black-box solvers that will maximize the possibilities of solving I within a given amount of time. Similar cases are retrieved using the *k-nearest neighbor* (knn) algorithm and a similarity metric (euclidean distance) which represents the distance between the feature vector of I and a training example.

CPHYDRA uses two set of features: *syntactic* and *semantic*. The *syntactic* feature set aims to capture general properties of the problem and is computed directly from the XCSP specification [RL09] of the instance, while the *semantic* features set aims to capture information about the structure of the problem and is computed by launching the *mistral* solver [Heb08] during a preliminary testing period of 2 secs and mainly involves general search statistics obtained during the preliminary testing period. After computing the features and obtaining the k most similar cases, the following problem formulated to define the order in which each individual CSP solver will be executed.

$$\begin{array}{ll} \text{maximize} & \bigcup_{s \in S} \frac{C(s,f(s))}{d(c)+1} \\ \text{subject to} & \sum_{s \in S} f(s) \leq 1800 \end{array}$$

Where $C(s,t)$ indicates that a given solver s is able to solve a similar instance with a time limit of t seconds, and $f(s) \leq 1800$ indicates that the overall time to solve a single instance is set to 1800 seconds, and $d(c)$ indicates the distance between the similar case and the new instance. Intuitively, nearest instances are more likely to be more informative. As

[4] http://www.cril.univ-artois.fr/CPAI08/

pointed out by the authors, solving this problem is NP-hard. However, as CPHYDRA uses a few number of solvers and k is not greater than 40, computing the schedule of solvers does not introduce any considerable overhead.

CPHYDRA was trained using instances from the 2007 CSP competition, this way before entering in the 2008 competition the database of stored cases (or instances) was completely filled with respectively features and runtimes for each particular training sample, and the information in the database remained constant during the competition (as incremental learning is not allowed). Finally, it is important to notice that CPHYDRA won 4 out of 5 categories in the competition and in the remaining category was placed 2nd.

Other researchers have also used machine learning to build portfolio algorithms in the context of constraint programming; in [XSS09] Xu et al., used Q-learning to identify branching rules and on the other hand, [KMN10, GKMN10] used a AdaBoost-like approach [Sch02] to automatically tune the minion solver [GJM06].

3.2 Portfolios for Optimization problems

So far, we have presented methodologies for solving satisfiability problems in the context of SAT, QBF and CSP. In this section, we switch our attention to optimization problems. Unlike satisfiability problems, solving an optimization problem involves finding the best solution and prove that the solution is the optimal. Unfortunately, in many cases this process cannot be completed within a reasonable amount of time and the system must provide to the user the best solution found so far.

Beck and Freuder in [BF04] proposed the low-knowledge approach to automatically select the most appropriate algorithm in the context of optimization problems. To this end, a given time limit of L secs is available to find the best solution for a given instance. In this way, during a prediction phase, each algorithm candidate A_i is executed with a timeout of L/N seconds. Every t secs the best solution found so far is stored in K_i=[b_1,b_2, ..., b_n] indicating the best solution for the ith algorithm up to each time interval (i.e., t, 2t, 3t, ..., nt seconds). Taking this into account, three methods to select the best algorithm are proposed: *pcost* selects the algorithm with final best solution cost, *pslope* selects the algorithm considering the best observed improvement in between stored solutions, and

pextrap selects the algorithm based on an extrapolation of the current solution at a time L/N to L. Overall the quality of the prediction technique depends on the characteristics of the problem and the amount of time given for the prediction phase.

In [CB05, CB04] Carchrae and Beck extended the low-knowledge approach including machine learning techniques to select the most appropriate algorithm. On the one hand, the low-knowledge strategy is used to identify a set of generic features common to all optimization problems (i.e., solution cost for each algorithm in the portfolio), these features are categorical values (i.e., 1/0 variables) obtained during the prediction time. This way, each algorithm is equipped with n features, one for each value in K_i, the algorithm with best performance at the ith interval gets value 1, and while the remaining algorithms get value 0. These features or attributes are a very elegant solution when there is absolutely no information about the distribution of problems. However, these descriptors did not include enough generalization to perform better than a simple strategy such as *pcost* which selects the best algorithm during the prediction phase

On the other hand, also in [CB05] a reinforcement learning method is used to interleave and assign computational runtime to all available algorithms according to the current performance of each one. In this context, algorithms are executed in rounds (or iterations), at each round the reinforcement learning method assigns computational runtime to each algorithm according to latest previous improvements. Intuitively algorithms with current best performance improvement would be assigned with more computational resources. This dynamic switching mechanism showed very good results as it was able to outperform the best pure single algorithm.

Beck and co-author's work in the low-knowledge framework is an interesting methodology, however it is important to note that this framework by itself can not be applied straightforward to satisfiability problems because it requires the algorithm to provide results of intermediate solutions (best solution found up to some time limit), and as pointed out above, the goal of a CSP is precisely to find a single solution.

On the other hand, the Bid Evaluation Problem (BEP) in Combinatorial auctions has been studied using empirical hardness models in [LBNS06] and decision trees in [GM04]. The former uses a SATzilla-like model, learning a runtime prediction function for each algorithm in the portfolio, while the latter uses decision trees to build a classification model

by considering the winner algorithm as the one with minimal runtime for each instance during the training phase. Notice that the main difference between these two approaches and Beck's work with low-knowledge is that low-knowledge is general enough to be applied to any optimization problems, while [LBNS06] and [GM04] need experimented users to define a suitable feature vector of the problem.

3.3 Per class learning

So far, we have presented preliminary work devoted to the use of Machine Learning to the algorithm selection problem in the context of Rice's framework (see Section 3.1), that is the selection of the best algorithm is based on some features or descriptors of the problem.

In this section, we would like to switch our attention to per-class methods for automatic parameter tuning. Contrasting with previous approaches which select the best algorithm for each particular instance, the following set of algorithms aim to select the best parameter configuration for a set of problems. In other words, The training data set is used to properly identify a single parameter configuration and it will be used to tackle all testing instances.

The automatic parameter problem was formally described in [BSPV02, Bir04] as an optimization problem where the search space is the space of all possible parameter configurations. In this context, Birattari et al., proposed the application of a racing algorithm called `F-RACE` to identify the most suitable configuration for a given algorithm. `F-RACE` aims to use a machine learning idea, typically used in feature selection [MM97] to avoid exhaustive search among all possible candidates. Broadly speaking, `F-RACE` iteratively executes each parameter configuration on a set of problem instances and as soon as enough statistical evidence (based on the Friedman's test) shows that a given configuration is inferior than the rest, this configuration is discarded and no longer considered as a candidate, `F-RACE` iterates until a single parameter configuration is found or a given timeout is reached.

Although `F-RACE` is a very effective technique, it involves the execution of each parameter configuration until enough statistical evidence is found to discard poor candidates. Therefore, `F-RACE` is limited to few parameter configurations. To overcome this limitation `paramILS` [HHLBS09, Hut09] and `GGA` [AST09] are developed to deal with large (order

of hundreds of thousands) number of parameters configurations. However, ironically these tools themselves require some parameters to be tuned.

paramILS is an iterative local search algorithm that executes the following two steps procedure until no improvement is found or a given time limit is reached.

1. Identification of the initial parameter configuration (usually algorithm's default configuration).

2. Using local search operators to explore the parameter configuration space, and the best known configuration found so far is updated according to some performance criteria.

paramILS is build on top of a restart-based search algorithm which involves three parameters (r,s,$p_{restart}$). $p_{restart}$ indicates the probability of restarting the search, r and s indicate respectively the degree of perturbation added to the initial configuration when restarting the search and at each iteration of the algorithm.

Recent work conducted in [AST09] uses genetic algorithms in combination with a gender separation strategy to focus the search on promising parameter configurations. In this context, one of the primary goals of GGA is to reduce the overall fitness evaluations, because computing the fitness value for a given population (or configuration) x involves running the solver several times with the same configuration x. The gender separation requires 5 parameters that were experimentally tuned in [AST09]. The main advantage of GGA compared to paramILS is that the genetic algorithm approach supports continuous values, so that no discretization step is required. However, GGA requires the user to define a variable tree structure which basically indicates dependencies between parameters.

Early work on the automatic configuration problem studied the CALIBRA system [ADL06]. CALIBRA initially performs a full factorial design to define the initial and worse values. Straight-after a local search algorithm is used until a local minimum is obtained. At this point a new configuration is carefully crafted using past local optima and worst solutions, and this procedure is repeated until a user defined stopping criterium is reached (e.g., max. number of experiments). The major limitation of CALIBRA is that it is limited to up to 5 parameters.

3.4 Adaptive Control

Methodologies described in this section differ from the above mentioned work as they do not require descriptors to characterize problem instances. Instead, algorithms automatically adapt their internal parameters when solving a problem instance. That is, the algorithm provides some feedback based on the current performance and the system decides the next action.

The quickest first principle (QFP) [BTW96] is a methodology for combining CSP heuristics. QFP relies on the fact that easy instances can frequently be solved by simple algorithms, while exceptionally difficult instances will require more complex heuristics. In this context, it is necessary to pre-define an execution order of heuristics and the switching mechanics is set according to the thrashing indicator (i.e., when the search seems to be stuck at a given portion of the tree), once the thrashing value of the current strategy reach some cutoff value, it becomes necessary to stop the current search procedure and try again with the next heuristic in the sequence.

The purpose in *The Adaptive Constraint Engine* (ACE) [EFW$^+$02] is to unify the decision of several heuristics in order to guide the search process. In this way, a voting mechanism selects a mixture of variable/value ordering heuristics by means using some offline learned weights associated to each heuristic candidate.

In [PE08] Petrovic and Epstein presented an extension of ACE. In this work the authors showed that a subset of powerful heuristics is more effective than using all available ones. Therefore, the objective of this method is to select the most suitable subset of heuristics in order to explore the search space with promising candidates. This method uses the weights learned during the training phase in order to discard heuristics whose weights are lower than their corresponding benchmark heuristics. Generally speaking, there are two benchmark heuristics, one for variable-ordering and one for value-ordering and these benchmark heuristics represents random behavior. Taking this into account, ACE learns a mixture of better-than-random heuristics.

It is also important to highlight that ACE also includes the *transfer learning* [RK07] concept. In particular, ACE learns on a class of problems, and then continue to learn on other classes, adapting weights as it goes. However, during the testing phase ACE

drops heuristics with poor weights values (below-benchmark heuristics) and weights of the remaining heuristics are not updated during the testing phase.

Another contribution to the ACE framework is presented in [PE06]. This work uses a restart-based learning method with two objectives: speed up the overall learning time and improve the quality of the learnt function. Thus, once ACE identifies that the current learnt information is no longer useful, it throws away the current learning model by re-initializing the associated weights to each heuristic. This way, two new parameters $\langle k, r \rangle$ are added to the ACE framework. In this context, a restart is activated if k consecutive unsuccessful runs have been observed during the last l training problems. Notice that the performance of this new approach is very sensitive to these two parameters. On the one hand, if k is too low the restart engine will be launched too often, while on the other hand, if k is too high it could be too late for restarting, because ACE could had already restored the learning values.

Battiti et al., in [BB05] deeply studied the reactive search framework to online tune the parameters of a given algorithm. In this way, the reactive search framework typically uses Machine Learning to on-the-fly adjust the parameters of a given algorithm in order to better react to fast changing conditions while solving a problem-instance. Taking this into account, reactive search requires to balance the *intensification-diversification* dilemma. That is, focussing the search on known good actions (*intensification*) and trying new actions to diversify the search (*diversification*). An interesting application of the reactive search framework is described and analyzed in [BT94] where Battiti and Tecchinolli proposed a mechanism for adapting the size of the tabu list by increasing (resp. decreasing) its size according to the recent progress of the search.

Another family of adaptive methods are developed in local search-based algorithms for SAT. Hoos [Hoo02] studied an adaptive noise (degree of randomization) mechanism for the *Novely+* algorithm by systematically increasing (resp. decreasing) the noise if no improvement has been observed for a while. A similar strategy has been adopted for other algorithms such as: *AdaptG^{2}WSAT* and *AdaptG^{2}WSAT+* in [LWZ07]. On the other hand, modern local search algorithms for SAT such as gNovelty+ [PG07, PTGS08] are developed by carefully selecting the main components of existing algorithms. Therefore according to the current conditions of the problem *gNovelty+* chooses the most appropriate component.

Finally, important efforts have been devoted to study the adaptive operator selection

(AOS) in the context of Genetic Algorithms in [DFSS08] and evolutionary computation in [FRSS10]. Broadly speaking, The main idea behind the AOS is to select the most appropriate operator based on the past history of each individual one by means of balancing the trade-off between exploration and exploitation[5]. Supporting this claim, Fialho and co-authors studied several methods for the AOS such as: Probabilistic Matching, Adaptive Pursuit and Multi-armed Bandit with some extensions developed for the AOS framework. An Extensive experimental validation described and analyzed in [FDSS10] indicates that multi-armed bandits methods were superior on artificial problems. Since these methods have been extensively verified, it would be interesting to study their performance in the low-knowledge framework described in Section 3.2

3.5 Other work

In order to conclude our description of previous work, we would like to highlight some extra strategies developed to combine different heuristics to solve combinatorial problems.

Back to 1996 in [Min96, Min93] Minton proposed the *Multi-Tactic Analytic Compiler* (MULTI-TAC) system to automatically configure LISP programs. Initially MULTI-TAC is equipped with a set of generic variable/value selection heuristics to automatically infer problem-specific algorithms which will effectively solve a family of problem instances. During a training phase MULTI-TAC analyzes the CSP specification of the problem to generate rules for variable/value selection, that are later compared against the generic variable/value heuristics to filter out low-quality ones. Finally, beam search [R.B92] is used to select the final subset of rules which will be then applied on a testing set of instances.

Although several methods for the variable selection problem have been thoroughly studied in the literature, less effort has been devoted to the value selection problem. Nevertheless, a remarkable work presented in [ZE08] aims to select the most suitable value for a given variable in a binary CSP. In this work, Zhang and Epstein keep track of the frequency in which the domain of a variable is modified through constraint propagation, and less frequently removed values are most likely to be used. Experimental results suggest that this simple strategy is an interesting alternative to traditional value ordering methods.

[5] exploration-exploration is also known as intensification-diversification

In [MCC06] Monfroy et al., proposed an adaptive enumeration of CSP strategies (or heuristics) in order to dynamically replace strategies exhibiting poor performances. The proposed engine maintains a priority value v for each strategy and v is updated using a given set of indicators that measure the progress of the search (e.g., thrashing, search deep, number of nodes, etc.). In this way, if the search observes some progress using a given strategy c, then c is rewarded, otherwise c is penalized. This work also proposes a *metabacktrack* engine which monitors the thrashing value in order to backtrack n steps or restarting the search from scratch. For instance, if the current thrashing is too high it might be better to jump to the root node, but if the thrashing is not critical it should backtracks few steps in the tree.

SATenstein [KXHLB09, Khu10] is motivated by the fact that current state-of-the-art dynamic local search algorithms for SAT are built by mixing the main properties of existing algorithms. For instance novelty [MSK97] as most WalkSAT-based [SKC94] algorithms firstly selects an unsatisfied clause and then uses the GSAT [SLM92] score function to select the next variable to flip. Supporting this claim, SATenstein is developed on top of a carefully selected set of components which consider the main characteristics (or properties) of a wide variety of local search algorithms and leave the selection of critical components to an automatic parameter tuning tool (e.g., `paramILS`, see Section 3.3). Finally, it is worth mentioning that SATenstein has been extensively tested in problems from a wide variety of domains.

In [GS01] Gomes and Selman studied the applicability of building an algorithm portfolio for combinatorial problems by launching several algorithms (or different copies of the same algorithm with different random seeds) in parallel settings or interleaving the execution of all candidates in a single machine. This work showed that when exploring the use of several independent randomized algorithms, one might rather explore the use of algorithms with large variance to obtain better performance improvements. However, there are still open questions in the use of a portfolio algorithm, for instance the definition of the number of processors or the amount of time to interleave the execution of each algorithm. Notice that the definition of a portfolio algorithm in this paper differs from the one used earlier on in this chapter (see Section 3.1), because here the portfolio executes several independent searches until at least one of them find a solution or a given time out is reached.

The estimation of the number of nodes in tree search algorithms goes back to the seminal work of Knuth in [Knu75] where the Knuth's method is proposed. The algorithm starts with the root node and moves down the tree by selecting random successors, once the algorithm reach a leaf-node the estimation is computed as $1 + d_1 + d_1 \times d_2 + d_1 \times d_2 \times d_3 + \ldots$, where d_i indicates the number of successors at the ith level of the tree. This procedure is repeated several times (each time with a different random seed) and the final estimation is the average across all samples.

Knuth's estimator is a very elegant algorithm but it might not work properly when the search tree is not known in advance, for instance in the case of branch and bound algorithms, where the search tree is systematically pruned to avoid non-optimal solutions. Therefore, Lobjois and Lemaître in [LL98] extended the knuth's estimator to deal with upper bound solutions which systematically help to prune useless portions of the search. This way, the so-called *selection by performance prediction* (SPP) method estimates the runtime of a given optimization algorithm. SPP estimates the total number of nodes of a branch and bound algorithm as well as the average time to explore a single node. Finally, once a new instance arrives the portfolio selects the algorithm with minimal overall runtime (number of times $\times$ average per node). In addition to this work, other extensions of the Knuth's estimator have been explored in [KSTW06] in the context of SAT and in [CKL06] in the context of Mixed Integer Programming.

In [SMJGT09] Smith-Miles et al., proposed the use of machine learning algorithms to identify the structure of a given problem. To this end, a portfolio algorithm is built on top of Supervised learning methods (Decision trees and Neural Networks) and an unsupervised learning method (Self-organized maps) to select the best between two heuristics. After extensive experiments with more than 70000 instances, all the three learning methods performed much better than independent heuristics. However, the self-organized map provides an interesting graphical representation which helps to understand the relationship between the features and the structure of the problem.

Another interesting approach was presented by Kautz et al., [KHR$^+$02] who proposed the use supervised machine learning techniques to build an optimal restart policy R=$\{t_1$,t_2, $\ldots$, $t_n\}$ for a given algorithm in order to quickly solve a distribution of problem instances. In this context t_i indicates the cutoff for the ith restart. In order to build R Kautz et al.,

assume independent restarts, where no information is shared between restarts. This work was extended by Ruan et al., [RHK02] by exchanging information between restarts (e.g., runtime from previous restarts) to on-the-fly update the cutoff for the upcoming restarts.

In [RAD10] Rachelson et al., used machine learning to predict values for a subset of variables in the context of Mixed Integer Programming (MIP). This way, the initial problem formulation is relaxed by instantiating some variables with the suggested values. This work has two main limitations. Firstly, the new solution is not necessary the optimal one. Secondly the learnt model is not generic enough to be applied to any MIP problem-instance since the input feature vector is restricted to a static number of variables and constraints.

Rather than using supervised learning techniques to select the most appropriate algorithm for a given instance, another option is to define a scheduler policy to interleave the execution of *black-box* solvers while solving a new problem instance. Roughly speaking, there are two main approaches for building the scheduler. On the one hand, Portfolios with deadlines [WB08] defines an execution sequence $[(t_1, A_1), (t_2, A_2), \ldots, (t_n, A_n)]$, where t_i indicates the time cutoff for the ith algorithm in the sequence. As soon as a given algorithm reachs its associated time limit, this algorithm is discarded and no longer considered for the current instance. On the other hand, Combining Multiples Heuristics Online [SGS07, SGS08] allows multiples executions of a given algorithm by using two functioning modes: *suspend-restart* and *stop-restart*. The former allows an algorithm to be suspended and then resumed at a later time, while the latter stops an algorithm and restart it at a later time. Additionally, [SS08] uses a set of boolean features to identify the most suitable scheduler for a given instance.

Chapter 4

Exploiting Weak Dependencies in Tree-based Search

So far in this thesis, we have presented background material in Chapter 2 and an extensive literature review in Chapter 3. From now on, we move our attention to the main contributions. This way, in this chapter, our objective is to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the variables. Our method shows that these relations can be detected with some acceptable overhead during constraint propagation. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic.

4.1 Introduction

The relationships between the variables of a combinatorial problem are key to its resolution. Among all the possible relations, explicit constraints are the most straightforward and were widely used. For instance, they are used to support classical look-ahead and look-back schemes. During look-ahead, they can limit the scope of the enforcement of some consistency level. During look-back, they can improve the backtracking by jumping to

related and/or guilty decisions. These relationships are also used in dynamic variable ordering (DVO) to relate the current variable selection to past decisions (e.g., [Bre79]), or to give preference to the most constrained parts of the problem, etc.

Recently, backdoors have been illustrated. A backdoor can be informally defined as a subset of the variables such that, once assigned values, the remaining instance simplifies to a computationally tractable class. Backdoors can be explained by the presence of a particular relation between variables, e.g., functional dependencies. Unfortunately, detecting backdoors can be computationally expensive [DGS07], and their exploitation is often restricted to restart-based strategies like in modern SAT solvers [WGS03].

In this chapter, our objective is to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the importance of each variable. Our method assumes that these relations can be detected with low overhead during constraint propagation. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic.

In the following section, we start with a general description of the constraint propagation algorithm. Section 4.3 describes our new heuristic. Section 4.4 presents experimental results. Finally, before summarizing the chapter, Section 4.5, presents related work.

4.2 Constraint propagation

Constraint propagation is usually based on some constraint network property which determines its locality and therefore its computational cost. Arc-consistency is widely used, and the results of its combination with backtrack search is called (MAC) for *Maintain Arc-consistency* [SF94]. Constraints are high-level abstractions implemented by propagators[1] [ST08, Tac09], these propagators help to remove invalid values from the variables through constraint propagation

Algorithm 4.1 describes a classic constraint propagation engine [SC06]. In this algorithm, constraints are managed as propagators in a propagation queue, Q. This structure

[1] In the following, we will use the term propagator as a synonym for constraint.

Algorithm 4.1 Classic propagation engine

```
1: Q = {p1, p2, ...}
2: while Q ≠ {} do
3:    p = choose(Q);
4:    run(p);
5:    for all X_i ∈ vars(p) s.t. D_i was narrowed do
6:       schedule(Q, p, X_i);
7:    end for
8: end while
```

represents the set of propagators that need to be revised. Revising a propagator corresponds to the enforcement of some consistency level on the domains of the associated variables.

Initially, Q is set to the entire set of constraints. This is used to enforce the arc-consistency property before the search process. During depth-first exploration, each decision is added to an empty queue, and propagated through this algorithm.

The function $choose$ removes a propagator $p \in Q$, run applies the filtering algorithm associated to p, and $schedule$ adds all propagators associated to X_i, i.e., $prop(X_i)$, to Q. The algorithm terminates when the queue is empty. A fix-point is reached and more propagations can only appear as the result of a tree-based decision.

Definition 4.1 *$f(X, y)$ is a functional dependency between the variables in the set X and the variable y, if and only if, for each combination of values in X there is precisely one value for y satisfying f.*

Many constraints of arity k can be seen as functional dependencies between a set of $k - 1$ variables and some remaining variable y. For instance, the arithmetic constraint $X + Y = Z$, gives the dependencies $f(\{X, Y\}, Z)$, $f(\{X, Z\}, Y)$, and $f(\{Y, Z\}, X)$. There are also many exceptions like the constraint $X \neq Y$, where in the general case, one variable is not functionally dependent of the other one.

4.3 Exploiting weak dependencies in tree-based search

4.3.1 Weak dependencies

In general, functional dependencies are difficult to obtain as they require to check the consequences of assigning each value for a given set of variables. Therefore, Our objective is to take advantage of functional dependencies during search. We propose to heuristically discover a weaker form of relation called *weak dependency* between pairs of variables. A weak dependency is observed when a variable gets instantiated as the result of another instantiation. Our new DVO heuristic records these weak dependencies and exploits them to prioritize the variables during the search process.

Definition 4.2 *During constraint propagation based on Algorithm 4.1, we call* (X, Y) *a weak dependency if the two following conditions hold:*

1. Y *is instantiated as the result of the execution of a propagator* p*.*
2. p *was inserted in* Q *as the result of the instantiation of* X*.*

Notice that the previous definition excludes intermediate constraint propagation narrowing the domain of other variables or narrowing the domain of Y. This excludes a situation where Y gets instantiated as a consequence of domain narrowing.

Property 4.1 *Weak dependency relations* (X, Y) *can be recorded as the result of the execution of a propagator* p iff $X \in vars(p)$ *and* $Y \in vars(p)$.

The proof is straightforward if we consider Algorithm 4.1.

4.3.2 Example

To illustrate our definition, we consider the following set of constraints:

- $p_1 \equiv X_1 + X_2 < X_3$
- $p_2 \equiv X_1 \neq X_4$
- $p_3 \equiv X_4 \neq X_5$

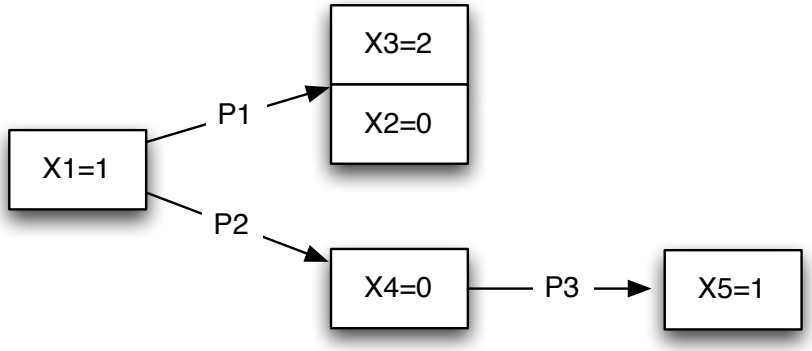


Figure 4.1: Weak dependencies

With the domains, $D_1 = D_2 = D_4 = D_5 = \{0, 1\}$ and $D_3 = \{1, 2\}$.

The initial filtering does not remove any value and the search process has to be started. Figure 4.1 shows the graph of weak dependencies assuming that the search is started on X_1 with value 1, the propagator $X_1 = 1$ is added to Q, and after its execution the domain D_1 has been narrowed, so that it is necessary to schedule p_1 and p_2.

Running p_1 sets X_2 to 0, and X_3 to 2, and gives the weak dependencies (X_1, X_2) and (X_1, X_3). Afterwards, p_2 sets X_4 to 0 which corresponds to (X_1, X_4). Finally, the narrowing of D_4 schedules p_3 which sets X_5 to 1, and gives the weak dependency (X_4, X_5).

In this example, it can also be observed that propagator p_1 assigns variables X_2 and X_3. However, weak dependencies (X_2, X_3) or (X_3, X_2) are not computed, since these variables are narrowed due to the nature of the propagator instead of a direct consequence between these two variables.

Weak dependencies are binary, therefore they only roughly approximate functional dependencies. For example, with the constraint $X + Y = Z$ they will never record $(\{X, Y\}, Z)$. On the other hand, weak dependencies exploit the current domains of the variables and can record relations which are not true in general but hold in particular cases. For instance, the propagator p_3 above creates (X_4, X_5). This represents a real functional dependency since the domains of the variables are binary and equal.

4.3.3 Computing weak dependencies

We can represent weak dependencies as a weighted digraph relation among the variables of the problem, where the nodes of the graph are the variables and the edges indicate weak dependencies relations between two variables, i.e, when there is an edge between two variables X and Y, the direction of the edge shows the relation and its weight indicates the number of observed occurrences of that relation (e.g., Figure 4.1, assuming that the weight for each edge is 1).

In a propagation centered approach [LS07] each variable has a list of dependent propagators and each propagator knows its variables (see Figure 4.2).

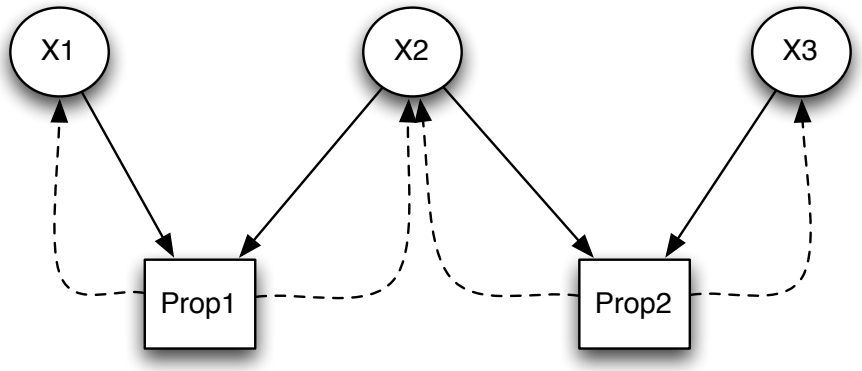


Figure 4.2: Variables and propagators

In this way, once the domain of a variable is narrowed it is necessary to schedule its associated propagators into the propagator pool. Since we are interested in capturing weak dependencies, we have to track the reasons for constraint propagation. More specifically, when a propagator gets activated as the result of the direct assignment of some variable, we need to keep a reference to that variable. Since the assignment of several variables can activate a propagator, we might have to keep several references.

A modified *schedule* procedure is shown in Algorithm 4.2. The algorithm starts by enqueueing all the propagators associated to a given variable X_i into the propagators pool. If the propagator p was called as the result of the assignment of X_i ($|D_i| = 1$), a weak dependency is created between each variable of the set *p.assigned* and X_i. Variables from

Algorithm 4.2 Schedule(Queue Q, Propagator p, Variable X_i)

```
1: enqueue(Q, prop(X_i));
2: if |D_i| = 1 then
3:    dependencies(p.assigned, X_i);
4:    for all p' in prop(X_i) do
5:       p'.assigned.add(X_i);
6:    end for
7: end if
```

this set are the ones whose assignment was the reason for propagating p. After that, a reference to X_i is added to its propagators $prop(X_i)$. This is done to ensure that if these propagators assign other variables, a subsequent call to the schedule procedure will be able to create dependencies between X_i and these variables.

4.3.4 The domFD dynamic variable ordering

In the previous section, we have seen that a generic constraint propagation algorithm can be modified to compute weak dependencies. As we pointed out above, weak dependencies can be seen as a weighted digraph relation among the variables. Using this graph, we propose to define a function $FD(X_i)$ which computes the out-degree weight of a variable X_i taking into account only uninstantiated variables.

$$FD(X_i) = \sum_{X_j \in \Gamma^+(X_i)} weight(X_i, X_j)$$

Where $\Gamma^+(x)$ (resp. $\Gamma^-(x)$) represents the set of outgoing (resp. ingoing) edges from (resp. to) x in the graph of dependencies. It is also important to note that when there is no outgoing edge associated to X_i we assume $FD(X_i) = 1$.

Given the definition of FD, we propose to define *domFD*, a new DVO heuristic based on both: the observed weak dependencies of the problem and the well-known fail-first $mindom$ heuristic:

$$\textit{domFD}(X_i) = \frac{|X_i|}{FD(X_i)}$$

Then, the heuristic selects the variable whose *domFD* value is minimal.

4.3.5 Complexities of domFD

Space

We know from Property 4.1 that dependencies are created between variables which share a constraint. Therefore, computing the weak dependency graph requires in the worst case a space proportional to the space used for the representation of the problem. Assuming n variables and m constraints, the space is proportional to $n + m$.

Time

The computation of weak dependencies is tightly linked to constraint propagation. The original schedule procedure only enqueues the propagators related to X_i in Q, therefore its original cost is $O(m)$. Our new procedure creates dependencies each time a variable gets instantiated. Dependencies between variables can be recorded as the result of the instantiation of one or several variables. In the latter case, up to $n-1$ dependencies can be created since the instantiation of up to $n-1$ variables can be responsible for the scheduling of the current propagator (line 3 in Algorithm 4.2). Once dependencies are created, the propagators associated to X_i need to reference it. Here the cost is bounded by m. Overall, the time complexity of the new schedule procedure is $O(n + m)$.

We now have to consider the cost of maintaining the weak dependency graph. Since our heuristic only considers the weights related to the variables which are not instantiated we have to disconnect variables from the graph when they get a value, and we have to reconnect them when the search backtracks. This can be done incrementally.

Practically, we do not have to physically remove a variable from the dependency graph, we can just offset the weight of the recorded dependencies between other variables and that variable. For instance, when X_i gets instantiated as the result of a tree decision or as the result of constraint propagation, we only need to update the out degrees of variables $X_j \in \Gamma^-(X_i)$. The update is done by decreasing their associated counter $X_j.FD$ by $weight(X_j, X_i)$. These counters represent the number of times the weak dependency (X_j, X_i) was observed during the search process. During backtracking, X_i gets back

its domain, and we just have to "reconnect" the associated $X_j \in \Gamma^-(X_i)$ by adding $weight(X_j, X_i)$ to $X_j.FD$. Since a variable can be linked to m propagators, an update of the dependency graph cost $O(m)$. In the worst case, each branching holds no propagation and therefore at each node, the cost of updating the dependency graph is $O(m)$.

Finally, selecting the variable which minimizes *domFD* can cost an iteration over n variables if no special data structure is used.

Now if we consider all the operations, constraint propagation with the new schedule procedure, disconnecting a single variable, and selection of the variable which minimizes *domFD*, we have $O(n + m)$ - as opposed to $O(m)$ initially.

4.4 Experiments

In this section, we propose to study the performance of *domFD* when compared to *dom-wdeg*, a recently introduced heuristic able to focus on the difficult parts of a problem [BHLS04].

In *dom-wdeg*, the priority is given to variables which are frequently involved in failed constraints. A weight is added to each constraint and updated (i.e, incremented by one) each time a constraint fails. Using this value variables are selected based on their domain size and their total associated weight. X_i, the selected variable minimizes *dom-wdeg* (X_i)= $|X_i| / \sum_{c \in prop(X_i)} weight(c)$.

This heuristic is used in the Abscon solver which appeared to be the most robust in the 2006 CSP-competition[2] where it finished 1 time first, 3 times second, 3 times third, and 2 times fourth, when compared against 15 other solvers.

To compare *domFD* against the powerful *dom-wdeg*, we implemented them in Gecode-2.0.1 [Gec06] and used them to tackle several problems. Since Gecode is now widely used, we decided to take from the Internet problems already encoded for the Gecode library. We paid attention to the fact that overall our problems cover a large set of Gecode's constraints.

We used 35 instances coming from 9 different benchmark families. They involve satisfaction, counting, and optimization problems. They were solved using the default Gecode's branch-and-prune strategy, and a modified restart technique based on the default strategy. In

[2]http://www.cril.univ-artois.fr/ CPAI06/round2/results/ranking.php?idev=6

the tests, the value selection ordering was Gecode's `INT_VAL_MIN`, which returns the minimal value of a domain. All the experiments were performed on a MacBook-Pro 2.4GHz Intel Core 2 Duo, under Ubuntu linux 7.10 and gcc version 4.0.1. A time-out (TO) of 10 minutes was used for each experiment.

4.4.1 The problems

In the following, we list the different benchmark families used in this chapter, all these problems (except Crowded-chess) have been widely studied in the CSPLib [GW99]. Note that for all problems (except Quasigroup) the model and its implementation is the one proposed in the Gecode repository[3].

- ***qwh*:** Quasigroup [AGKS00], problem 3 of CSPLib, this problem consists in completing a pre-filled $N \times N$ matrix with the numbers $[1, 2, \ldots, N]$ such that for each column (resp. row) of the matrix, each element occurs exactly once.

- ***gol-rul*:** Golomb-ruler [SSW99], problem 6 of CSPLib, this problem consists in finding a list M of numbers (so-called marks) such that the difference between any pair of marks $M_i - M_j$ $(i \neq j)$ in M are all distinct. The number of elements in M indicate the order of the ruler, and the maximum distance between any two pair of elements in M indicates the length of the ruler. The goal is to find a golomb-ruler with minimal length for a pre-defined order.

- ***all-int*:** All-interval [GMS03], problem 7 of CSPLib, this problem consists in finding all possible permutations $L = [x_1, \ldots x_n]$ of numbers such that L is a permutation of $[0, 1, \ldots, n-1]$ and $[|x_1 - x_2|, |x_2 - x_3|\ , \ldots, |x_{n-1} - x_n|]$ is a permutation of $[1, 2, \ldots n-1]$.

- ***nono*:** Nonogram [Bos01], problem 12 of CSPLib, this problem consists of a matrix with a list of number for each column (resp. row), this list represents a set of rules indicating how many consecutively filled squares are for each column (resp. row).

[3] Available from http://www.gecode.org/gecode-doc -latest/ group__ExProblem.html.

- ***magic-squ*:** Magic-square [Wei], problem 19 of CSPLib, this problem consists of completing a $N \times N$ matrix with the numbers $1, 2, \ldots, N^2$ where each column, row and the two main diagonals sum the same number.

- ***lfn*:** Langford-number [HKS01], problem 24 of CSPLib, this problem consists in arranging k set of N numbers, such as each occurrence of a given number m is m times in from the last one.

- ***sport-lea***: Sport league tournament [Hen99], problem 26 of CSPLib, this problem consists[4] in scheduling a round-robin tournament such that: there are t teams, the season lasts $t-1$ weeks, each game between two different teams occurs exactly once, every team plays one game in each week of the season, there are t/2 periods and each week every period is scheduled for one game, no team plays more than twice in the same period over the course of the season. Notice that due to the specifications of the problem, t must be an even number.

- ***bibd*:** Balanced Incomplete Block Design [Pre01], problem 28 of CSPLib, this problem consists of a $v \times b$ matrix such that: the number of ones for each column (resp. rows) is equal to r (resp. k), and the scalar product for each pair of distinct rows is equal to λ. The original problem contains 5 parameters, however b and r can be derived from λ, v and k.

- ***crow-ch*:** Crowded-chess [Lag08], this problem consists in arranging n queens, n rooks, $2n-1$ bishops and k knights on a $n \times n$ chessboard, so that queens cannot attack each others, no rook can attack another rook and no bishop can attack another bishop. Note that two queens (in general two pieces of the same type) are attacking each other even if there is a bishop (in general another piece of different type) between them.

In the following, when an instance is solved, the number of nodes in the tree(s) (*#nodes*), the number of failures (*#failures*) and the time (*time (s)*) in seconds are reported. If the 10 minutes time-out is reached, TO is reported, and the best performing algorithm (w.r.t. runtime efficiency) is indicated in bold.

[4]This problem description was taken from the gecode [Gec06] definition of the problem.

4.4.2 Searching for all solutions or for an optimal solution

The first part of Table 4.1, presents results related to the finding of all the solutions of all-interval problems of order 11 to 14. We can observe that the trees generated with *domFD* are usually far smaller than the ones generated by *dom-wdeg*. Most of the time, *domFD* runtime is also better. However, the time per nodes (i.e., *#nodes/time* in Table 4.1) is more important for our heuristic. For instance, on all-int-14, *dom-wdeg* does 89973 nodes/s while *domFD* runs at 54122 nodes/s.

The second part of the table presents results for the optimal Golomb-rulers of orders 10 to 12. Here, we can observe that order 10 is easier for *dom-wdeg*, but sizes trees are comparable. Order 11, and 12 give the advantages to *domFD*, with far smaller search trees and better runtimes. As before, the time per node is more important for our heuristic (31771 vs 35852 on gol-rul-11).

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	time (s)	#nodes	#failures	time (s)
all-int-11	100844	50261	0.93	52846	26262	**0.81**
all-int-12	552668	276003	6.92	211958	105648	**3.45**
all-int-13	2.34M	1.17M	**26.13**	1.64M	821419	29.74
all-int-14	15.73M	7.86M	**174.83**	11.27M	5.63M	208.23
gol-rul-10	93732	46866	**1.97**	102910	51449	2.70
gol-rul-11	2.77M	1.38M	77.26	1.77M	889633	**55.71**
gol-rul-12	12.45M	6.22M	404.92	6.97M	3.48M	**266.28**

Table 4.1: All solutions and optimal solution

4.4.3 Searching for a solution with a classical branch-and-prune strategy

Experiments related to the finding of a first solution are presented in Table 4.2. They show results for respectively, quasi-groups, balance incomplete block design, Langford numbers, and nonograms.

Quasi-groups

Three instances of order 30 with 316 unassigned positions were produced with the generator presented in [AGKS00]. On these instances, *domFD* always generates smaller search trees. When this difference is large enough e.g., second instance, the runtime is also better.

Balance incomplete block design

Our heuristic always explores smaller trees which allows better runtimes. Interestingly the third instance is solved in 0.03 seconds by *domFD* while *dom-wdeg* cannot solve it in 10 minutes.

Langford numbers

On these problems, *domFD* is always superior to *dom-wdeg*. For instance, lfn-3-10 can be solved by both heuristics but the performance of *domFD* is far better: 190 times faster.

Nonograms

Table 4.2 shows results for the nonogram problem. Three instances of orders 5, 8, and 9 were generated. Here again, the trees are systematically smaller with *domFD* and when the difference is large enough runtimes are always better.

4.4.4 Searching for a solution with a restart-based branch-and-prune strategy

Restart-based searches are very efficient since they can alleviate the effects of early bad decisions. Therefore, it is important to test our new heuristic with a restart strategy.

A restart is done when some cutoff limit in the number of fails is met, i.e., at some node in a tree. There, the actual *domFD*-graph is stored and used to start the next tree-based search. This allows the early selection of well ranked variables. The same technique is used with *dom-wdeg*, and the next search tree can branch early on well ranked variables.

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	time (s)	#nodes	#failures	time (s)
qwh-30-316-1	1215	603	**0.22**	234	115	0.32
qwh-30-316-2	48141	24063	8.09	10454	5220	**3.62**
qwh-30-316-3	6704	3347	**1.11**	2880	1437	1.15
bibd-7-3-2	100	39	**0.01**	65	28	**0.01**
bibd-7-3-3	383	180	0.03	96	42	**0.01**
bibd-7-3-4	—	—	TO	132	56	**0.03**
lfn-3-9	168638	84316	6.16	7527	3760	**0.26**
lfn-2-19	—	—	TO	1.64M	822500	**43.05**
lfn-3-10	2.21M	1.10M	87.15	12440	6218	**0.46**
nono-5	1785	879	0.12	491	239	**0.11**
nono-8	17979	8983	3.54	1084	537	**0.54**
nono-9	248	115	**0.04**	120	58	0.12

Table 4.2: First solution, branch-and-prune strategy

This part presents results with a restart-based branch-and-prune where the cutoff value used to restart the search was initially set to 1000, and the cutoff increase policy to $\times 1.2$ (geometric factor). The same 10 minutes time-out was used.

Table 4.3 presents the results for magic square, crowded chess, sport league tournament, quasi-groups, and bibd problems.

Magic square

Instances of orders 5 to 11 were solved. Clearly, *domFD* is the only heuristic able to solve large orders within the time limit. For example, *dom-wdeg* cannot deal orders greater than 8, while our technique can. The reduction in the search tree sizes is very significant, e.g., on mag-squ-8, *dom-wdeg* develops 35.18M nodes and *domFD* 152466, which allows it to be more than 100 times faster.

Crowded chess

As before, *domFD* can tackle large problems while *dom-wdeg* cannot.

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	time (s)	#nodes	#failures	time (s)
mag-squ-5	2239	1113	**0.02**	3025	1505	0.06
mag-squ-6	33238	16564	0.32	4924	2440	**0.08**
mag-squ-7	9963	4868	**0.20**	33422	16614	0.86
mag-squ-8	35.18M	17.59M	460.40	152446	75987	**4.51**
mag-squ-9	—	—	TO	66387	32951	**1.64**
mag-squ-10	—	—	TO	83737	41607	**2.17**
mag-squ-11	—	—	TO	8.52M	4.26M	**374.62**
crow-ch-7	2029	1002	**0.04**	3340	1656	0.22
crow-ch-8	16147	8036	0.67	2041	1002	**0.14**
crow-ch-9	129827	64788	**6.15**	228480	114089	37.97
crow-ch-10	—	—	TO	1134052	566761	**263.01**
sport-lea-14	4746	2327	0.68	4814	2359	**0.65**
sport-lea-16	28508	14073	4.05	3913	1912	**0.61**
sport-lea-18	546475	272510	101.70	51680	25549	**10.72**
sport-lea-20	182074	90355	**36.69**	2.07M	1.03M	514.18
qwh-30-316-1	1215	603	**0.22**	234	115	0.32
qwh-30-316-2	118348	59104	20.06	8828	4397	**2.7**
qwh-30-316-3	8944	4451	1.68	3114	1552	**1.01**
qwh-35-405	2.38M	1.19M	562.62	475053	237369	**236.05**
bibd-7-3-2	100	39	**0.01**	65	28	**0.01**
bibd-7-3-3	383	180	0.03	96	42	**0.01**
bibd-7-3-4	6486	3210	0.79	132	56	**0.03**

Table 4.3: First solution, restart-based strategy

Sport league tournament

If we exclude the last instance, *domFD* is always better than *dom-wdeg*.

Quasi-groups

Here, on most problems, *domFD* generates smaller search trees, and can return a solution more quickly. On the hardest problem, (order 35), *domFD* is nearly two time faster.

Balanced incomplete block design

Here *domFD* performs very well, with both smaller search trees and small runtime.

4.4.5 Synthesis

Table 4.4 summarizes the performance of the heuristics. These results were generated by only taking into account the problems which can be solved by both *domFD* and *dom-wdeg* i.e., we removed 6 instances which cannot be solved by *dom-wdeg*.

heuristic	*average*			
	#nodes	#failures	time (s)	nodes/s
dom-wdeg	2.14M	1.07M	56.99	37664
domFD	717202	358419	39.53	18139

Table 4.4: Synthesis of the experiments

We can observe that the search trees generated by *domFD* are on the average three times smaller. The difference in the number of fails is similar. Finally, even if *domFD* is 2 times slower on the time per node, it is 31% faster overall.

Technically, our integration into Gecode is quite straightforward and not particularly optimized. For instance we use *Leda* [LED], an external library to maintain the graph, while a bespoke light class with the right set of features should be used. The way we record weak dependencies is also not optimized and requires extra data structures whose accesses could be easily improved, e.g., the *assigned* list of variables shown in Algorithm

4.2. For all these reasons, we think that it must be possible to increase the speed of our heuristic by some factor.

We also did some experiments to see if the computation of *domFD* could be cheaply approximated. We used a counter with each variable to record the number of times that variable was at the origin of a weak dependency. This represents an approximation of *domFD* since the counter considers dependencies on instantiated variables. Unfortunately, this - fast - approximation is always beaten by *domFD* on large instances.

4.5 Previous work

In [BHLS04], the authors have proposed *dom-wdeg*, an heuristic which gives priority to variables frequently involved in failed constraints. It adds a weight to each constraint which is updated (i.e, incremented by one) each time the constraint fails. Using this value variables are ranked according to domain size and associated weight. X_i, the selected variable minimizes *dom-wdeg* (X_i)= $|X_i|/\sum_{c \in prop(X_i)} weight(c)$. As shown in the previous section, *domFD* is superior to *dom-wdeg* on many problems. Interestingly, while *dom-wdeg* can only learn information from conflicts, *domFD* can also learn from successful branchings. This is an important difference between these two techniques.

In [Ref04], Refalo proposes the *impact* dynamic variable-value selection heuristic. The rational here is to maximize the reduction of the remaining search space. In this context, an *impact* is computed taking into account the reduction of the search space due to an instantiated variable. As a result of this, at each decision point this heuristic suggests a variable, as well as the best value instantiation for such variable.

With *domFD*, a variable is well ranked if its instantiation has generated several others instantiation. This is equivalent to an important pruning of the search space. In that respect *domFD* is close to impact. However, its principle is the dynamic exploitation of functional dependencies, not the explicit quantification of search space reductions. More generally, since DVO heuristics are all based on some understanding of the *fail-first* principle they are all aiming at an important reduction of the search space.

To improve SAT solving, [EGS02] proposes a new pre-processing step that exploits the structural knowledge that is hidden in a CNF formula. It delivers an hybrid formula made

of clauses together with a set of equations of the form $y = f(x_1, ..., x_n)$. This set of functional dependencies is then exploited to eliminate clauses and variables, while preserving satisfiability. This work detects real functions while our heuristic observes weak dependencies. Moreover, it uses a pre-processing step while we perform our learning during constraint propagation.

4.6 Summary

In this chapter, we have presented a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the branching variables. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic.

Experiments done on 9 benchmarks families showed that on the average *domFD* reduces search trees by a factor 3 and runtime by 31% when compared against *dom-wdeg*, one of the best dynamic variable ordering heuristic. *domFD* is also more expensive to compute since it puts some overhead on the propagation engine. However, it seems that our implementation can be improved, for example by using incremental data structures to record potential dependencies in the propagation engine.

Our heuristic learns from successes, allowing a quick exploitation of the solver's work. In a way, this is complementary to *dom-wdeg* which learns from failures. Moreover, both techniques rely on the computation of $mindom$. Combining their respective strengths seems obvious. We did extensive experiments around a new mixture, $dom(x)/(wdeg(x) + FD(x))$ but found out that *domFD* was better than this straightforward combination.

Chapter 5

Building Portfolios for the Protein Structure Prediction Problem

While in the previous chapter we have proposed a new variable selection for CSPs, in this chapter, we explore the application of Machine Learning to select the best COP heuristic in the context of the Protein Structure Prediction Problem. This contribution is twofold. First, the selection criterion is the quality (minimal cost) in expectation of the solution found after a fixed amount of time, as opposed to the expected runtime. Second, the presented approach, based on supervised Machine Learning algorithms, considers the original description of the protein structure prediction problem, as well as the features associated to the CP encoding of the problem.

5.1 Introduction

The protein structure prediction problem (PSP Problem) has been widely studied in the field of bioinformatics, since the three dimensional (3D) conformation of a given protein helps to determine its function. This problem is usually tackled using simplified models such as HP-models in [BW01] and a constraint logic programming approach in [DDF03]. However, even considering these abstractions the problem is computationally very difficult and traditional strategies cannot reach a solution within a reasonable time. Also, there has been several attempts to predict the structure and proteins fold using well known machine

learning techniques.

In this chapter, we propose the use of machine learning to automatically select the most promising Constraint Optimization algorithm for the PSP problem. In this context, proteins are represented as a feature vector in $\mathbb{R}^d$ and the algorithm selection process is based on a well known multi-class classification algorithm called *decision tree*. This way, the *decision tree* technique would suggest the most appropriate variable/value selection strategy used by a branch-and-bound algorithm in order to determine the 3D conformation of a given protein.

Unlike other portfolio-based selection approaches [HHHLB06, XHHLB07, AHS09, HW09b, GHBF05] which select the algorithm that minimizes the expected runtime, our approach selects the strategy that minimizes the expected cost of the solution found after a fixed amount of time. To the best of our knowledge, this is the first work which performs algorithm selection in an optimization setting by taking into account algorithm's solution cost instead of algorithm's runtime. Moreover, and unlike previous works which only extract the features exploited during machine learning from the SAT or CP encoding of the problem (see Chapter 3), our work explores the application of two features sets. In this way, we use features formulated directly from the application domain (*Problem Features*), as well as features from the CP encoding of the problem (*CP Features*).

This chapter is organized as follows. The PSP problem is described in Section 5.2. Section 5.3 shows the features or attributes used in this work. Section 5.4 presents the general idea of algorithms portfolio. Section 5.5 reports our experimental validation and Section 5.6 presents a summary of the chapter.

5.2 The protein structure prediction problem

The PSP problem is well known in computational biology and is currently considered as one of the *grand challenges* in this field. Broadly speaking the problem consists in finding the 3D conformation (so-called ternary structure) of a protein defined by its primary structure or a sequence of residues $S = \{s_1, s_2, \ldots, s_n\}$ where each residue s_i of the sequence represents one of the 20 amino-acids. The ternary structure is often defined by the minimal energy conformation. Figure 5.1 shows an input example (left) representing the

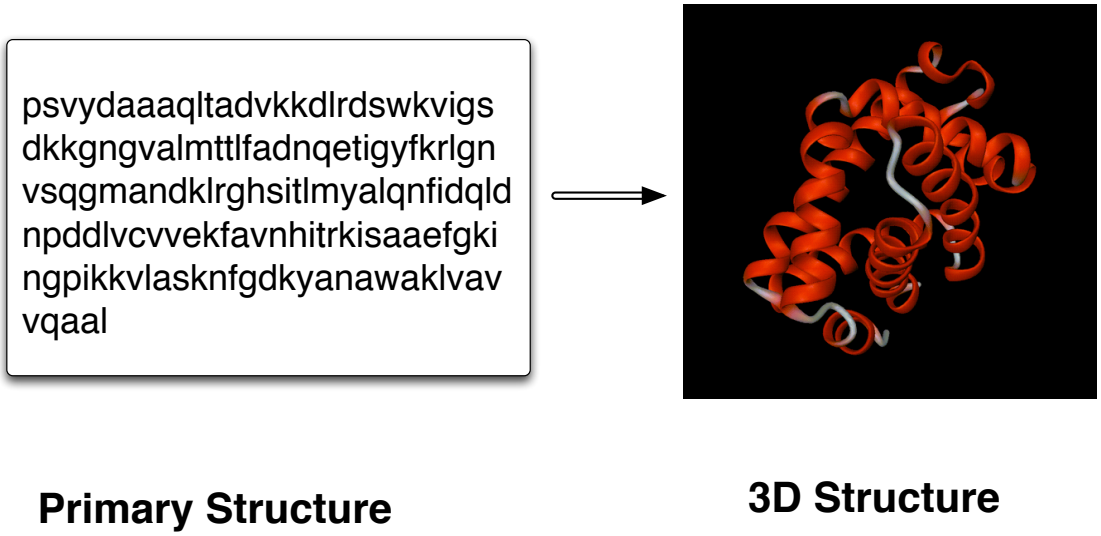


Figure 5.1: 3D conformation of the 3SDHA protein

amino-acid sequence of the 3SDHA protein and the corresponding output (right) of the 3D configuration of such a protein[1].

This problem has been previously studied in [DDP07] using a constraint programming based model. In this model, each amino-acid is seen as a single atom unit and two consecutive amino-acids in the sequence are separated by a fixed distance also known as a lattice unit. In the mathematical representation, each amino-acid is represented by means of a vector w, such that $w(i) \rightarrow \langle x, y, z \rangle$ denotes the current position of the ith amino-acid of the sequence in the three dimensional space, $||w(i) - w(i+1)|| = \sqrt{2}$ indicates that two consecutive amino-acids have the same fixed distance, $\forall_{i \neq j} ||w(i) - w(j)|| \geq 2$ indicates that two consecutive amino-acids cannot overlap one another and each amino-acid occupies a single sphere. This way, the final goal of the PSP problem is to minimize the overall energy conformation of a protein which is defined by the following formula:

$$E(w) = \sum_{1 \leq i < n} \sum_{i+2 \leq j \leq n} contact(w(i), w(j)) \times Pot(s_i, s_j)$$

[1]The 3D position for each atom was obtained using [NLLP10], a specialized bioinformatics package, and the figure was produced with DINO [Phi03]

where, *contact* is 1 iff two amino-acids are immediate neighbors in the three dimensional cube (or lattice) and not sequential in the primary structure, otherwise *contact* is set to 0. And *Pot* defines the energy contribution of two adjacent residues. It is important to note that some other lattice models have been proposed such as the HP-Model [BW01] where each amino-acid in the sequence is translated from the 20 symbols alphabet into a two symbols alphabet (i.e., hydrophobic (H) and polar (P)).

5.3 Features

In order to describe a given problem instance in terms of descriptors or features, a user might choose one of the following kinds of features: *Problem Features* encoding general information about the problem itself and *CP features* (or solver codification features) encoding general information about the CP abstraction of the problem. On the one hand, the *Problem Features* set is flexible enough no matter the solving technique (e.g., SAT, CP, LP, LS, etc.), however, are restricted to a particular problem domain. On the other hand, the *CP feature* set is general enough to be used for several problem domains but is limited to a CP abstraction of the problem.

In general, using one feature set or another is up to the user, in the context of the PSP problem a user with a bioinformatics background might prefer a biological set of features, while one with a mathematical and/or constraint programming knowledge might prefer the CP features. In the following we present both kinds of features taking into account its *pros* and *cons*.

5.3.1 Problem features

This feature set aims to characterize the PSP problem and was obtained from the extensive machine learning literature on protein fold prediction [PC03, DpAD10, CB06]. In order to build the feature set, every amino-acid in the primary structure is replaced by the index 1, 2 or 3 according to the group it belongs, i.e., Hydrophobicity, Volume, Polarity and Polarizability (see Table 5.1). For instance, the sequence RSTVVH is encoded as 122332 based on the hydrophobicity attribute. This encoding is used to compute the following set

of descriptors:

- **Composition**: 3 features representing the percentage of each group in the sequence.
- **Transition**: 3 features representing the frequency with which a residue from $group_i$ is followed by a residue from $group_j$ (or vice versa).
- **Distribution**: 15 features representing the fraction in the sequence where the first residue, 25%, 50%, 75% and 100% of the residues are contained for each encoding in Table 5.1.

Attribute	Group 1	Group 2	Group 3
Hydrophobicity	R,K,E,D,Q,N	G,A,S,T,P,H,Y	C,V,L,I,M,F,W
Volume	G,A,S,C,T,P,D	N,V,E,Q,I,L	M,H,K,F,R,Y,W
Polarity	L,I,F,W,C,M,V,Y	P,A,T,G,S	H,Q,R,K,N,E,D
Polarizability	G,A,S,D,T	C,P,N,V,E,Q,I,L	K,M,H,F,R,Y,W

Table 5.1: Amino-acid feature's group

In total the feature set is a composition of 105 (84+20+1) features or descriptors: 84 ((15+3+3)×4) according to Table 5.1 and the previous descriptors, i.e., Composition, Transition and Distribution. 20 descriptors which represent the proportion of each amino-acid in the sequence. Finally the size of the sequence.

5.3.2 CP features

This feature set is a collection of 32 descriptors. These features include general information about the CP encoding of the problem and are described as follows:

- **Problem definition** (4 features): Number of variables, constraints, variables assigned/not assigned at the beginning of the search.
- **Variables size information** (6 features): `Size prod`, `sum`, `min`, `max`, `mean` and `variance` of all variables domain size.
- **Variables degree information** (8 features): `min`, `max`, `mean` and `variance` of all variables degree (resp. variables' domain/degree)

- **Constraints information** (6 features): The degree (or arity) of a given constraint c is represented by the total number of variables involved in c. Likewise the size of c is represented by the product of its corresponding variables. Taking into account this information, the following features are computed: `min`, `max`, `mean` of constraints size and degree.

- **Filtering cost category** (8 features): Each constraint c is associated a category[2]. In this way, we compute the number of constraints for each category. Intuitively each category represents the implementation cost of the filtering algorithm. *Cat* = $\{$*Exponential*, *Cubic*, *Quadratic*, *Linear expensive*, *Linear cheap*, *Ternary*, *Binary*, *Unary*$\}$. Where *Linear expensive* (resp. *cheap*) indicates the complexity of a linear equation constraint and the last three categories indicate the number of variables involved in the constraint. More information about the filtering cost category can be found in [Gec06].

Notice that similar features have been previously used to characterize SAT problems in [XHHLB07] and CSPs in [GKMN10]. The former set of features is limited to SAT, while the latter, among other properties, include information about partial satisfiability of the constraints (so called alldifferent statistics).

5.4 Algorithm portfolios

A portfolio algorithm is usually built on top of the general framework described in Figure 5.2. This framework is divided in two main phases: *offline* and *online*. During the *offline* phase a heuristic model is defined and used later on during the *online* (or testing) phase to identify the most appropriate algorithm to solve a given problem instance.

The *offline* phase requires an experimented user to identify a target distribution of problems in order to define a representative set of training instances. Afterwards, a pair $\langle x_i, y_i \rangle$ is computed for each training instance, where x_i and y_i represent respectively the vector of

[2]Out of 8 categories, detailed in
http://www.gecode.org/doc-latest/reference/classGecode_1_1PropCost.html

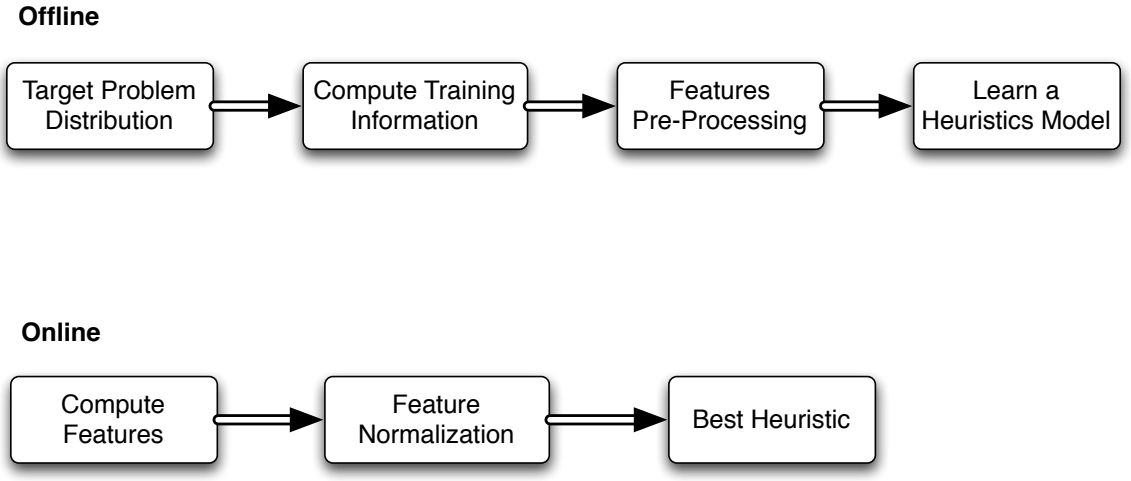


Figure 5.2: Traditional algorithms portfolio framework

features and the best algorithm for the ith example in the training set. Subsequently, a feature pre-processing step is used to remove irrelevant features and normalize feature values. In this context, irrelevant features do not increase the performance of the classifier, for instance, features with the same value overall training instances. Finally, a machine learning technique (e.g., Decision trees, SVMs, Case-based reasoning, Logistic regression, etc) is used to obtain the so-called heuristics model which defines a function $\mathcal{I} \rightarrow Alg$, where $\mathcal{I}$ is a feature vector representing a problem instance and Alg the most suitable algorithm to solve the given instance. On the other hand, the *online* phase is executed each time a new instance I arrives. Thus, it is only necessary to compute and normalize the feature vector corresponding to I in order to predict the best algorithm based on the heuristics model.

As pointed out in Chapter 3, in the literature, there exists a wide variety of machine learning algorithms to build a portfolio. For instance, SATzilla [XHHLB07] uses linear regression to build the portfolio based on an estimation of algorithm's runtime, CPHYDRA [OHH$^+$08] uses a case-based reasoning framework to build a heuristics model, and AQBF [PT07] uses traditional machine learning algorithms (decision trees, 1-nearest neighbor, decision rules and logistic regression). However, it is also worth mentioning that none of these learning techniques can be expected to be the best one for all existing problems. Indeed, the selection of the machine learning algorithm can also be seen as another layer of

the algorithm selection problem [Bre96].

A classical portfolio is learned by taking into account the overall computational time to solve a set of problem instances. For instance, SATzilla [XHHLB07], a well known portfolio for SAT problems, builds a regression model in order to estimate the solving time of each constitutive SAT solver. In this way, once an unseen instance arrives SATzilla selects the algorithm with minimal expected run-time.

However, solving a constraint optimization problem involves finding the best solution and proving that such a solution is indeed the optimal one. Unfortunately, in many cases this process cannot be completed within a reasonable amount of time and the system must provide the user with the best solution found so far. Following this idea, building the portfolio using algorithm's runtime is not an alternative. A solution would be building the portfolio taking into account the quality or cost of the solution found after some fixed amount of computational time (e.g., time-out parameter).

5.4.1 Algorithm subset selection

An important consideration before building a portfolio is the definition of its constitutive algorithms. In many situations, selecting the right subset of algorithms might improve the overall performance of the system. For instance, SATzilla selects the best subset of solvers by selecting candidates that are not well correlated with each other, and executes an exhaustive search with the remaining subset. However, exhaustive search involves exploring $2^n - 1$ (where n is the number of solvers) subsets which is computationally very expensive, so that exhaustive search is not desirable for a large pool of heuristics. Additionally, it is also necessary to determine when two solvers are not well correlated, which is still an open question.

Supporting this claim, we propose to borrow the ideas from feature selection methods [GE03] to choose the best subset of algorithms. Usually a feature selection method implements grid search in order to find the best feature set. The two most common algorithms are forward and backward selection. The former starts with an empty subset and incrementally adds variables until no improvement is found; while the latter starts with the full set and incrementally removes one variable at a time until no improvement is reached. Here,

we chose forward selection since we would prefer a small subset of algorithms to build the classifier.

As a performance metric, we consider the overall mean solution cost. Notice that we could have also used the accuracy of the machine learning algorithm; however, higher accuracy does not necessarily lead to a better performance, since misclassifying near-optimal heuristics is not as critical as correctly classifying the ones with a high solution cost.

In order to validate the algorithm subset selection method we used the traditional 10-fold cross-validation technique (see Chapter 2), but for each iteration the learner selects the best subset of algorithms by performing an inner 10-fold cross-validation. Figure 5.3 depicts the strategy, the entire data-set D is divided into 10 subsets $\{D_1, D_2, \ldots, D_{10}\}$ for each subset D_i the heuristics model is defined with $L = D - D_i$, where L itself employs an inner 10-fold cross-validation to determine the right subset of heuristics in the portfolio before testing on D_i.

Algorithm 5.1 shows the forward heuristic selection method used in order to compute the right subset of algorithms at each iteration of the 10-fold cross-validation procedure. Alg represents the full set of algorithms and D indicates the current training set. It is also important to notice that $EnergyEval$ uses an inner 10-fold cross-validation in order to obtain the mean energy evaluation considering D and SS', where D represents a set of instances and SS' represents a subset of algorithms to build the portfolio. Finally, SS stores the final subset of algorithms that will be then used to build the portfolio for the current iteration of the outer 10-fold cross-validation procedure. Therefore algorithms not included in SS are not considered at this iteration.

5.5 Experiments

In this chapter, we use the Gecode model proposed in [CDD08]. All algorithms (see Chapter 2) are home-made implementations integrated into the Gecode-2.1.1 constraint solver. We experimented with 180 real sequences from [DD17] of sizes ranging from 31 to 100[3], and performed 10-fold cross-validation to evaluate the model with an inner 10-fold cross-validation for learning the final subset of heuristics. All experiments were conducted on

[3]It is important to notice that other constraint programming approaches (e.g., [PDP05], [MSR+09]) are able to deal with instances up to 250 elements.

Algorithm 5.1 Forward Algorithm Selection (Algorithms Alg, Data D)

```
1: SS = {}
2: Best_Energy = ∞
3: repeat
4:    BestA = None
5:    for each Algorithm A in Alg and not in SS do
6:       SS' = SS ∪ {A}
7:       if EnergyEval(SS', data) < BestEnergy then
8:          BestA = A
9:          BestEnergy = EnergyEval(SS', D)
10:      end if
11:   end for
12:   if BestA == None then
13:      SS = SS ∪ {BestA}
14:   end if
15: until BestA == None or SS = Alg
16: return SS
```

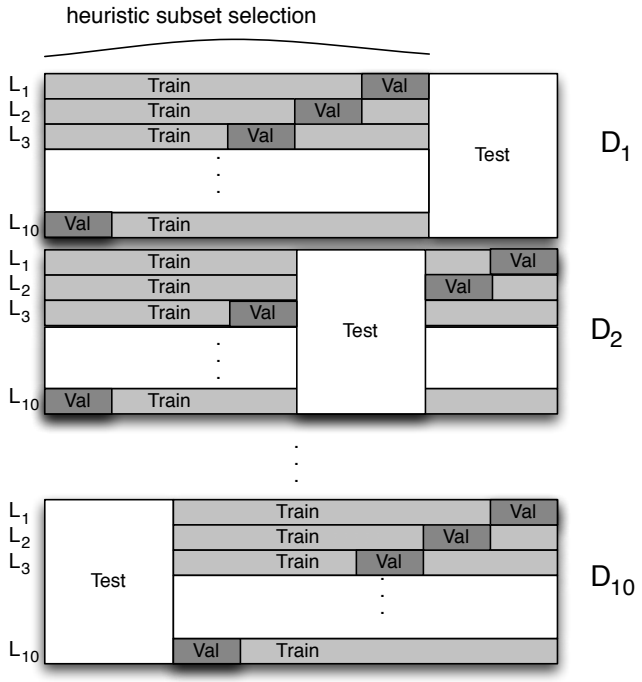


Figure 5.3: Experimental validation using 10-fold cross-validation and an inner forward selection

Linux boxes with 2 GB of RAM and 1.8 Ghz Intel processors.

Initial experiments in [CDD08] suggested that *lexico* is a powerful heuristic for the PSP problem, therefore we explored an extension of traditional variable selection algorithms, this novel version is presented as follows:

1. Select the first unassigned variable X_i if and only if X_{i+1} is assigned.
2. If the previous step cannot be satisfied, then select the variable according to a given heuristic criterion (e.g., *dom-wdeg*, *domFD*, etc.).

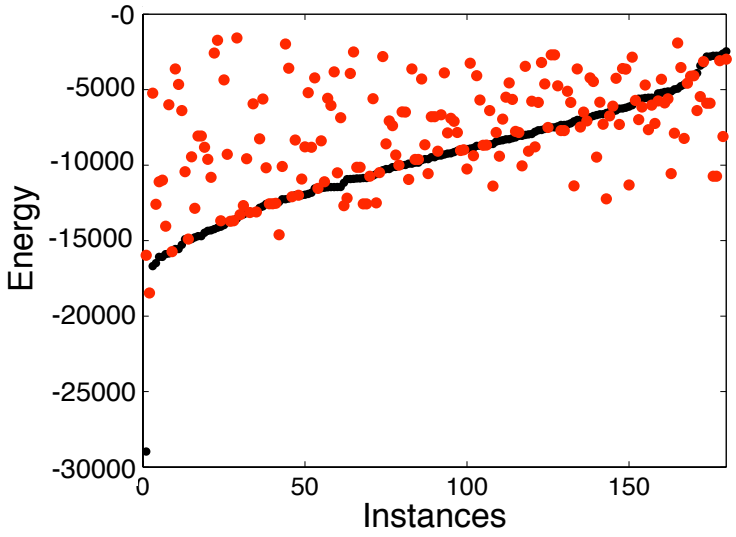


Figure 5.4: *wdeg* Vs $wdeg^+$

The algorithms which follow the strategy mentioned above would be named as: *dom-wdeg*$^+$, *wdeg*$^+$ *domFD*$^+$ and *impacts*$^+$. Figure 5.4 shows the performance of *wdeg* (red points) against its novel version *wdeg*$^+$ (black points), each point (either red and black) indicates the solution cost (y-axis) for a given instance (x-axis) and red points above the black ones indicate that *wdeg*$^+$ is better than *wdeg*. The data have been sorted according to the performance of *wdeg*$^+$. In this figure we observe that *wdeg*$^+$ is pretty effective for the PSP problem, therefore it is worth including the novel version of each algorithm into the portfolio.

Overall, we are considering a set of 10 variable selection heuristics $\mathcal{H}_{var} = \{$*lexico*, *mindom*, *dom-wdeg*, *wdeg*, *domFD*, *impacts*, *dom-wdeg*$^+$, *wdeg*$^+$, *domFD*$^+$, *impacts*$^+$ $\}$, 2 value selection algorithms $\mathcal{H}_{val} = \{$*min-val*, *med-val*$\}$, and finally it is also well known that restarting the search might improve the performance, therefore the initial cutoff value c is set to 1000; the cutoff increase policy is by multiplying c by 1.2 (geometric factor). The restart policy is considered for all heuristics, (except *lexico* and *mindom*). In total we consider a collection of 18 heuristics candidates, 8 variable selection $\times$ 2 value selection + 2 (*impacts* and *impacts*$^+$). Notice that *impacts* and *impacts*$^+$ are variable/value selection techniques.

We use the weka [FHH$^+$05] implementation, i.e., J48, of the C4.5 algorithm [Qui93] with its default parameter settings. Although J48 supports continuous features values we experimentally found that including a feature discretization step improved the accuracy of the machine learning algorithm. So that, a supervised discretization method is used to translate continuous features values into discrete values (see [WF05] for further details).

Strategy	Accuracy %	Mean
⟨*domFD*$^+$, *med-val*⟩	18.3	-10437
⟨*domFD*$^+$, *min-val*⟩	17.3	-10310
⟨*lexico*,*min-val*⟩	21.1	-10109
⟨ALL, bio⟩	38.9	-11418
⟨ALL, cp⟩	**42.7**	-11502
⟨ALL, cp+bio⟩	38.9	-11401
⟨FS, bio⟩	40.6	-12021
⟨FS, cp⟩	42.2	**-12168**
⟨FS, cp+bio⟩	40.0	-12085

Table 5.2: Overall strategies solution cost with a 5-minute timeout

Table 5.2 summarizes the performance for all strategies with a 5-minute timeout. In this table `Accuracy` indicates the percentage of times that a given strategy S_i is the winner considering a perfect portfolio and `Mean` indicates the mean solution cost overall instances using 10-fold cross-validation as explained in Section 5.4.1. Additionally, cp, bio, cp+bio indicate respectively the use of the CP feature set, the bio feature set and a concatenation of CP+bio features, and the best performing strategy (w.r.t. both accuracy and mean solution

cost) is indicated in bold.

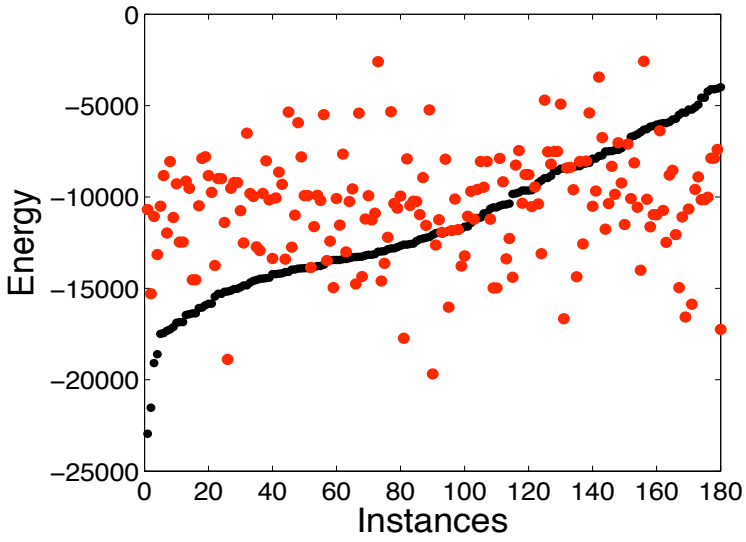
The first three rows indicate the performance of the best three single heuristics, i.e., ⟨*domFD*$^+$,*med-val*⟩, ⟨*domFD*$^+$,*min-val* ⟩ and ⟨*lexico*,*min-val*⟩. The next three rows indicate the performance of the portfolios considering all heuristics candidates, i.e., ⟨ALL,bio⟩, ⟨ALL,cp⟩, and ⟨ALL,cp+bio⟩. Finally, the last three rows indicate the performance of the portfolio using Forward Selection (FS) to automatically identify the best subset of heuristics, i.e., ⟨FS,bio⟩, ⟨FS,cp⟩, and ⟨FS,cp+bio⟩ .

As can be observed the three features sets exhibit close performances; however, using the *cp* feature set is slightly better than the *bio* and *cp+bio* features sets. An alternative explanation lies in the fact that the model is a CP codification of the biological problem which represents a high level abstraction and does not cover all biological properties of the PSP problem. For instance, in real situations amino-acids can be placed anywhere in $\mathbb{R}^3$, but in the CP codification amino-acids must be placed inside the lattice model.

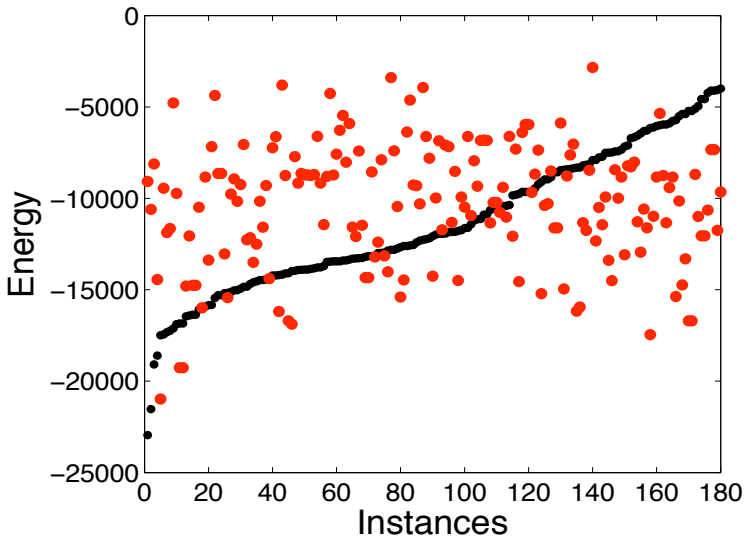
Another observation is that the portfolio with the best accuracy is not necessarily the one with best solution cost. For instance ⟨ALL,cp⟩ reached the best accuracy but its solution cost is -11502 against -12168 for ⟨FS,cp⟩ which reported the second best accuracy overall strategies. Moreover, as one might have expected, the overall solution cost is on average better when considering the automatic algorithm selection process. It is also worth mentioning that we also experimented with a random heuristic selection by computing the mean across 10 independent runs for each instance. However, this random selection strategy exhibited a mean solution cost of -6631 which is outperformed by all the portfolio strategies in Table 5.2.

A detailed examination of the cp+bio feature set is presented in Figures 5.5 and 5.6. Each black point represents the performance of the portfolio for a given instance and each red point represents the performance of each comparative algorithm, i.e., ⟨*domFD*$^+$, *min-val*⟩ and ⟨*domFD*$^+$, *med-val*⟩, the two best single heuristics. For analysis purposes hereafter, data have been sorted according to the performance of black points. Notice that since the optimization goal is to find the minimal energy configuration, red points above the black ones indicate that the portfolio is better.

Figure 5.5(a) shows the performance of ⟨*domFD*$^+$, *med-val*⟩ against ⟨ALL, cp+bio⟩ (building the portfolio using all available heuristics); in this figure ⟨ALL, cp+bio⟩ is better

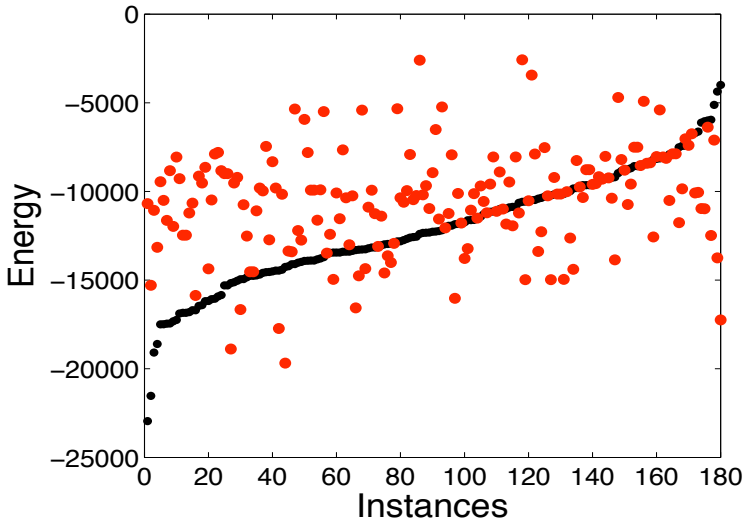


(a) ⟨*domFD*$^+$, *med-val*⟩ Vs ⟨ALL, cp+bio⟩

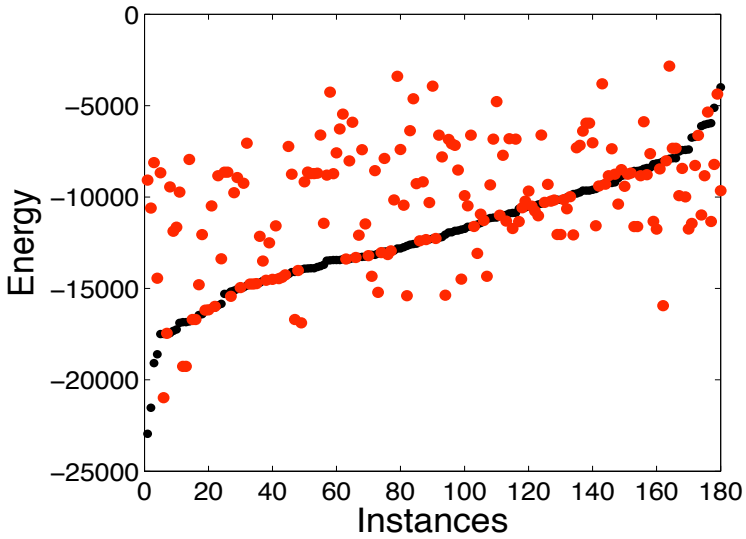


(b) ⟨*domFD*$^+$, *min-val*⟩ Vs ⟨ALL, cp+bio⟩

Figure 5.5: Experimental evaluation using all available heuristics



(a) ⟨*domFD*$^+$, *med-val*⟩ Vs ⟨FS, cp+bio⟩



(b) ⟨*domFD*$^+$, *min-val*⟩ Vs ⟨FS, cp+bio⟩

Figure 5.6: Experimental evaluation using forward heuristic selection

than ⟨*domFD*$^+$, *med-val*⟩ in 107 instances and worse in 73 instances. Figure 5.5(b) shows the performance of ⟨*domFD*$^+$, *min-val*⟩ against ⟨ALL, cp+bio⟩; here the portfolio is better in 108 instances and worse in 62 instances.

Figure 5.6(a) shows the performance of ⟨*domFD*$^+$, *med-val*⟩ againts ⟨FS, cp+bio⟩ (building the portfolio using the algorithm subset selection); in this figure ⟨FS, cp+bio⟩ is better than ⟨*domFD*$^+$, *med-val*⟩ in 109 instances and worse in 43 instances. Figure 5.5(b) shows the performance of ⟨*domFD*$^+$, *min-val*⟩ against ⟨FS, cp+bio⟩; here the portfolio is better in 96 instances and worse in 45 instances.

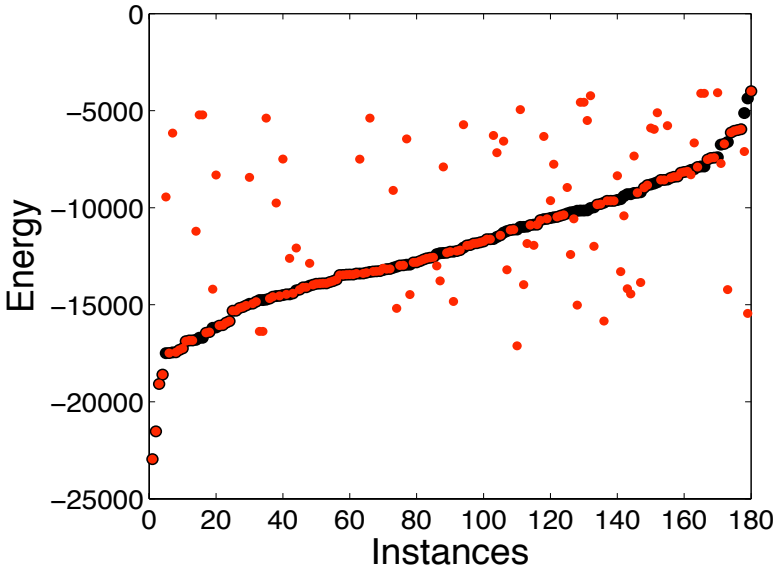


Figure 5.7: ⟨ALL, cp+bio⟩ Vs ⟨FS, cp+bio⟩

Finally, Figure 5.7 depicts the performance using all algorithms, i.e., ⟨ALL, cp+bio⟩, against the selection of the right subset of them, i.e., ⟨FS, cp+bio⟩. In this figure we observe that in 43 instances is better to use the forward heuristic selection method and only in 27 instances is better to build the portfolio with all candidates.

5.6 Summary

In this chapter, we have studied the application of Machine Learning techniques to build algorithms portfolios in the context of the PSP problem. We experimented two different feature sets. That is, features describing general biological properties of the problem and features extracted directly from the CP abstraction of the problem. Interestingly, in both situations, the resulting portfolio outperformed (w.r.t. solution quality) the best single algorithm.

The second contribution lies in the use of algorithm's cost solution in order to build the heuristics model which itself is based on a traditional multi-class classification algorithm, i.e., decision tree learning. Finally, our last contribution corresponds to the use of forward heuristic selection in order to chose the right subset of algorithms before building the heuristics model.

Chapter 6

Continuous Search in Constraint Programming

In the previous chapter, we studied a potential application of Machine Learning to the *Algorithm Selection Problem* for optimization settings by means of a well-known problem drawn from bioinformatics. In this chapter, we extend the traditional viewpoint of the *Algorithm Selection Problem* with Continuous Search, a novel paradigm. Continuous Search comes in two functioning modes: the production mode, which intends to solve new problem instances by means of using the current heuristics model; and the exploration mode, which reuses these instances to train and improve the heuristics model through Machine Learning during the computer idle time.

6.1 Introduction

In order to efficiently solve a Constraint Satisfaction Problem the user is usually left with the tedious task of tuning the search parameters of the constraint solver, and this is both time consuming and not necessary straightforward. Parameter tuning indeed appears to be conceptually simple, (i/ try different parameter settings on representative problem instances, ii/ pick up the setting yielding best average performance). Still, most users would easily consider instances which are not representative of their problems, and get misled.

The goal of this contribution is to allow any user to eventually get their constraint

solver achieving a top performance on their problems. The proposed approach is based on the original concept of Continuous Search (CS), gradually building a heuristics model tailored to the user's problems, and mapping a problem instance onto some appropriate parameter setting. A main contribution compared to the state-of-the-art (see Chapter 3) is to relax the requirement of a large set of representative problem instances to be available beforehand to support offline training. The heuristics model is initially empty (set to the initial default parameter setting of the constraint solver) and it is enriched along a lifelong learning approach, exploiting the problem instances submitted by the user to the constraint solver.

Formally, CS interleaves two functioning modes. In production or exploitation mode, the instance submitted by the user is processed by the constraint solver; the current heuristics model is used to parameterize the constraint solver depending on the instance at hand. In learning or exploration mode, CS reuses the last submitted instance, running other heuristics than the one used in production mode in order to find which heuristics would have been most efficient for this instance. CS thus gains some expertise relative to this particular instance, which is used to refine the general heuristics model through Machine Learning (see Chapter 2). During the exploration mode, new information is thus generated and exploited in order to refine the heuristics model, in a transparent manner: without requiring the user's input and by only using the idle computer's CPU cycles.

The chapter claim is that the CS methodology is realistic (most computational systems are always on, especially production ones) and compliant with real-world settings, where the solver is critically embedded within large and complex applications. The CS computational cost must be balanced against the huge computational cost of offline training [GHBF05, HW09b, GS01, PT07, WB08, XHHLB07]. Finally, lifelong learning appears a good way to construct an efficient and agnostic heuristics model, and able to adapt to new modeling styles or new classes of problem.

This chapter is organized as follows. Section 6.2 introduces the Continuous Search paradigm. Section 6.3 details the proposed algorithm. Section 6.4 reports on its experimental validation. Section 6.5 discusses previous work and the chapter concludes with a summary in Section 6.6.

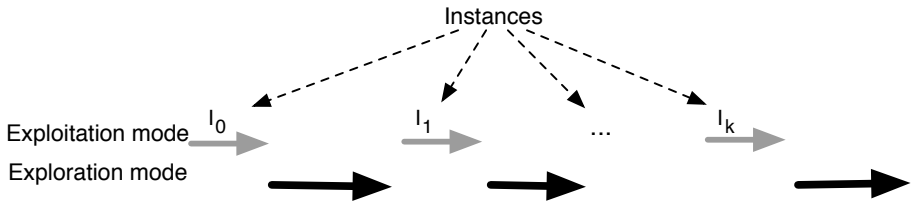


Figure 6.1: Continuous Search scenario

6.2 Continuous Search in Constraint Programming

The Continuous Search paradigm, illustrated on Figure 6.1, considers a functioning system governed from a heuristics model (which could be expressed as e.g., a set of rules, a knowledge base, a neural net). The core of continuous search is to exploit the problem instances submitted to the system along a 3-step process:

1. unseen problem instances are solved using the current heuristics model;

2. these instances are solved with other heuristics, yielding new information. This information associates to the description x of the example (accounting for the problem instance and the heuristics), a boolean label y (the heuristics improves/does not improve on the current heuristics model);

3. the training set $\mathcal{E}$, augmented with these new examples (x, y), is used to revise or relearn the heuristics model.

The Exploitation or production mode (step 1) aims at solving new problem instances as quickly as possible. The exploration or learning mode (steps 2 and 3) aims at learning a more accurate heuristics model.

Definition 6.1 *A continuous search system is endowed with a heuristics model, which is used as is to solve the current problem instance in production mode, and which is improved using the previously seen instances in learning mode.*

Initially, the heuristics model of a continuous search system is empty, that is, it is set to the default settings of the search system. In the proposed CS-based constraint programming, the default setting is a given heuristics noted *DEF* in the following (Section 6.3). Assumedly, *DEF* is a reasonably good strategy on average; the challenge is to improve on *DEF* for the particular types of instances which have been encountered in production mode.

6.3 Dynamic Continuous Search

The Continuous Search paradigm is applied to a restart-based constraint solver, defining the *dyn-CS* algorithm. After a general overview of *dyn-CS*, this section details the different modules thereof.

Figure 6.2 depicts the general scheme of *dyn-CS*. The constraint-based solver involves several restarts of the search. A restart is launched after the number of backtracks in the search tree reaches a user-specified threshold. The search stops after a given time limit. Before starting the tree-based search and after each subsequent restarts, the description x of the problem instance is computed (Section 6.3.1). We will call checkpoints the calculation of these descriptions.

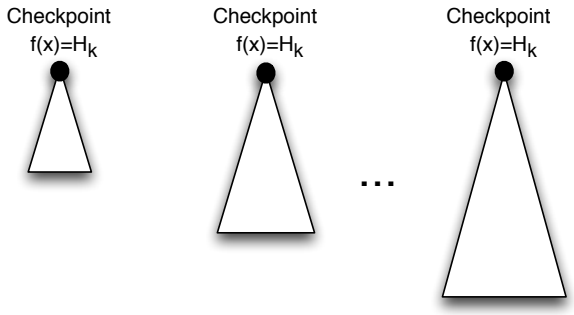


Figure 6.2: *dyn-CS*: selecting the best heuristic at each restart point

The global picture of the Continuous Search paradigm is described in Figure 6.3. In production (or exploitation) mode, the heuristics model f is used to compute the heuristic $f(x)$ to be applied for the entire checkpoint window, i.e., until the next restart. Not to be confused with the *choice point* which selects a variable/value pair at each node in the

search tree, *dyn-CS* selects the most promising heuristic at a given checkpoint and uses it for the whole checkpoint window. In learning (or exploration) mode, other combination of heuristics are applied (Section 6.3.4) and the eventual result (depending on whether the other heuristics improved on heuristics $f(x)$) leads to build training examples (Section 6.3.3). The augmented training set is used to relearn the heuristics model $f(x)$.

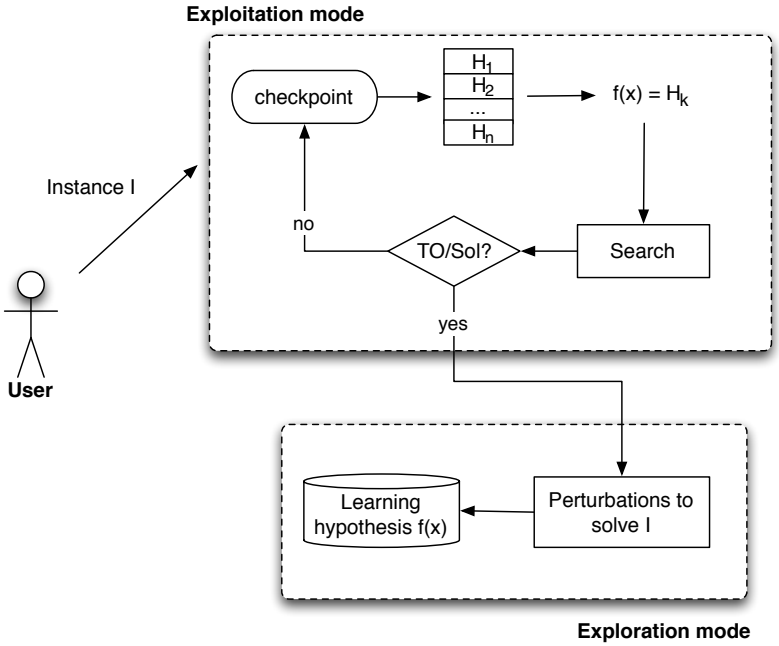


Figure 6.3: Continuous Search in Constraint Programming

6.3.1 Representing instances: feature definition

At each checkpoint (or restart), the description of the problem instance is computed including static and dynamic features.

While a few of these descriptors had already been used in SAT portfolio solvers [HHHLB06, XHHLB07], many descriptors had to be added as CSPs are more diverse than SAT instances: SAT instances only involve boolean variables and clauses, contrasting with CSPs using variables with large domains, and a variety of constraints and pruning rules [BCDP07, BHZ06, PBG05].

6.3.1.1 Static features

This feature set is a collection of 32 features previously defined in Chapter 5 (see Section 5.3.2), these features encode general information of a given problem instance; they are computed once for each instance as they are not modified along the resolution process. The static features also allow one to discriminate between types of problems, and different instances.

6.3.1.2 Dynamic features

Two kinds of dynamic features are used to monitor the performance of the search effort at a given checkpoint: global statistics describe the progress of the overall search process; local statistics check the evolution of a given strategy.

- **Heuristic criteria** (15 features): each heuristic criteria (i.e., *wdeg*, *dom-wdeg*, *impacts*) is computed for each variable; their `prod`, `min`, `max`, `mean` and `variance` over all variables are used as features.

- **Constraints weight** (12 features): likewise report the `min`, `max`, `mean` and `variance` of all constraints weight (i.e., constraints *wdeg*). Additionally the `mean` for each filtering cost category is used as feature. Where category is defined as follows, *Cat* = {*Exponential*, *Cubic*, *Quadratic*, *Linear expensive*, *Linear cheap*, *Ternary*, *Binary*, *Unary* }. Where *Linear expensive* (resp. *cheap*) indicates the complexity of a linear equation constraint and the last three categories indicate the number of variables involved in the constraint. More information about the filtering cost category can be found in [Gec06].

- **Constraints information** (3 features): `min`, `max` and `mean` of constraint's *run-prop*, where *run-prop* indicates the number of times the propagation engine has called the filtering algorithm of a given constraint.

- **Checkpoint information** (33 features): for every checkpoint$_i$ relevant information from the previous checkpoint$_{i-1}$ (when available) is included into the feature vector. From checkpoint$_{i-1}$ we include the total number of nodes and maximum search

depth. From the latest non-failed node, we consider the total number of assigned variables, satisfied constraints, sum of variables *wdeg* (resp. size and degree) and product of variables degree (resp. *domain*, *wdeg* and *impacts*) of non assigned variables. Finally using the previous 11 features the `mean` and `variance` is computed taking into account all visited checkpoints.

The attributes listed above include a collection of 95 features.

6.3.2 Feature pre-processing

Feature pre-processing is a most important step in Machine Learning [WF05], which can significantly improve the prediction accuracy of the learned hypothesis. Typically, the descriptive feature detailed above are on different scales; the number of variables and/or constraints can be high while the impact of (variable, value) is between 0 and 1. A data normalization step is performed using the *min-max normalization* [SSK06] formula:

$$v'_i = \left(\frac{v_i - min_i}{max_i - min_i} \right) \times (max_{new} - min_{new}) + min_{new}$$

Where min_{new}=-1, max_{new}=1 and min_i (resp. max_i) correspond to the normalization value for the *i-th* feature. In this way, feature values are scaled down in $[-1, 1]$. Although selecting the most informative features might improve the performance, in this chapter we do not consider any feature selection algorithm, and only features that are constant over all examples are removed as they offer no discriminant information.

6.3.3 Learning and using the heuristics model

The selection of the best heuristic for a given problem instance is formulated as a binary classification problem, as follows. Let $\mathcal{H}$ denote the set of k candidate heuristics, two particular elements in $\mathcal{H}$ being *DEF* (the default heuristics yielding reasonably good results on average) and *dyn-CS*, the (dynamic) ML-based heuristics model initially set to *DEF*.

Definition 6.2 *Each training example $p_i = (x_i, y_i)$ is generated by applying some heuristics h ($h \in \mathcal{H}, h \neq$ dyn-CS) at some checkpoint in the search tree of a given problem*

instance. Description x_i ($\in \mathbb{R}^{97}$) is made of the static feature values describing the problem instance, the dynamic feature values computed at this check point and describing the current search state, and two additional features: checkpoint-id gives the number of checkpoints up to now and cutoff-information gives the cutoff limit of the next restart. The associated label y_i is positive iff the associated runtime (using heuristic h instead of dyn-CS at the current checkpoint) improves on the heuristics model-based runtime (using dyn-CS at every checkpoint); otherwise, label y_i is negative.

If the problem instance cannot be solved (whatever the heuristics used, i.e., time out during the exploration and exploitation modes), it is discarded (since the associate training examples do not provide any relevant information).

In production mode, the hypothesis f learned from the above training examples (their generation is detailed in next subsection) is used as follows:

Definition 6.3 *At each checkpoint, for each $h \in \mathcal{H}$, the description x_h and the associated value $f(x_h)$ are computed.*
If there exists a single h such that $f(x_h)$ is positive, it is selected and used in the subsequent search effort.
If there exists several heuristics with positive $f(x_h)$, the one with maximal value is selected[1].
If $f(x_h)$ is negative for all h, the default heuristic DEF *is selected.*

6.3.4 Generating examples in Exploration mode

The Continuous Search paradigm uses the idle computer's CPU cycles to explore different heuristic combinations on the last seen problem instance, and see whether one could have done better than the current heuristics model on this instance. The rationale for this exploration is that improving on the last seen instance (albeit meaningless from a production viewpoint since the user already got a solution) will deliver useful indications as to how to best deal with further similar instances. In this way, the heuristics model will expectedly be tailored to the distribution of problem instances actually dealt with by the user.

[1]The rationale for this decision is that the margin, i.e. the distance of the example w.r.t the separating hyperplane, is interpreted as the confidence of the prediction [Vap95].

The CS exploration proceeds by slightly perturbing the heuristics model. Let *dyn-CS*$^{-i,h}$ denote the policy defined as: use heuristics model *dyn-CS* at all checkpoints except the i-th one, and use heuristic h at the i-checkpoint.

Algorithm 6.1 Exploration-time(instance: $\mathcal{I}$)

1: $\mathcal{E} = \{\}$ //initialize the training set
2: **for all** i in checkpoints($\mathcal{I}$) // loop over checkpoints ($\mathcal{I}$) **do**
3: **for all** h in $\mathcal{H}$ // loop over all heuristics **do**
4: Compute x describing the current checkpoint and h
5: **if** $h \neq$ *dyn-CS* **then**
6: Launch *dyn-CS* $^{-i,h}$
7: Define $y = 1$ iff *dyn-CS* $^{-i,h}$ improves on *dyn-CS* and -1 otherwise
8: $\mathcal{E} \leftarrow \mathcal{E} \cup \{x, y\}$
9: **end if**
10: **end for**
11: **end for**
12: return $\mathcal{E}$

Algorithm 6.1 describes the proposed Exploration mode for Continuous Search. A limited number (10 in this work) of checkpoints in the *dyn-CS* based resolution of instance $\mathcal{I}$ are considered (line 2); for each checkpoint and each heuristic h (distinct from the *dyn-CS*), a lesion study is conducted, applying h instead of *dyn-CS* at the i-th checkpoint (heuristics model *dyn-CS* $^{-i,h}$); the example (described from the i-th checkpoint and h) is labelled positive iff *dyn-CS* $^{-i,h}$ improves on *dyn-CS*, and added to the training set $\mathcal{E}$, once the exploration mode for a given instance is finished the hypothesis model is updated by retraining the SVM including the feature pre-processing as stated in Section 6.3.2.

6.3.5 Imbalanced examples

It is well known that one of the heuristics often performs much better than the others for a particular distribution of problems [CB08]. Accordingly, negative training examples considerably outnumber the positive ones (it is difficult to improve on the winning heuristics). This phenomenon, known as *Imbalanced distribution*, might severely hinder the SVM algorithm [AKJ04]. Two simple ways of enforcing a balanced distribution in such cases,

intensively examined in the literature and considered in earlier work [AHS09], are to oversample examples in the minority class (generating additional positive examples by Gaussianly perturbing the available ones) and/or undersample examples in the majority class.

Another options is to use prior knowledge to rebalance the training distribution. Formally, instead of labeling an example positive (resp, negative) iff the associated runtime is strictly less (resp. greater) than that of the heuristic model, we consider the difference between the runtimes. If the difference is less than some tolerance value dt, then the example is relabeled as positive.

The number of positive examples and hence the coverage of the learned heuristics model increase with dt; in the experiments (Section 6.4), dt is set to 20% of the time limit iff *time-exploitation* (time required to solve a given instance in production mode) is greater than the 20% of the time limit, otherwise dt is set to *time-exploitation*.

6.4 Experimental validation

This section reports on the experimental validation of the proposed Continuous Search approach. All tests were conducted on Linux Mandriva-2009 boxes with 8 GB of RAM and 2.33 Ghz Intel processors.

6.4.1 Experimental setting

The presented experiments consider 496 CSP instances taken from different repositories. Details of the *bibd* and *lfn* problem families are presented in Chapter 4 (see Section 4.4.1).

- **nsp**: 100 *nurse-scheduling* [DT00] instances from the MiniZinc [NSB+07] repository. This problem consists in defining the work schedule for a set of nurses such that each nurse might have at least n days off after m consecutive working days, and no nurse can work x consecutive nights.

- **bibd**: 83 *Balance Incomplete Block Design* instances from the XCSP [RL09] repository, translated into Gecode using Tailor [GMR07].

- **js**: 130 *Job Shop* instances [BHL05] from the XCSP repository. This problem consists in finding the schedule that minimizes the time to complete a set of n jobs with a set of m shared resources. Each job consists of a set of operations that might sequentially finished. That is, one operation should be completed before starting the next one. The original problem formulation is an optimization problem, however, instances in the XCSP repository were formulated as CSPs by accepting solutions with a given solution cost.

- **geom**: 100 *Geometric* [BHL05] instances from the XCSP repository. This problem consists of a graph with n variables that are randomly placed in a two dimensional cartesian plane. Edges are added to the graph iff the distance between the two variables is less or equal to $\sqrt{2}$. Finally, similarly to homogeneous random CSPs, some edges from the resulting graph are chosen to select pairs of incompatible values for the variables.

- **lfn**: 83 *Langford-number* instances, translated into Gecode using global and channelling constraints.

The learning algorithm used in the experimental validation of the proposed approach is a Support Vector Machine with Gaussian kernel, using the libSVM implementation with default parameters [CL01]. All considered CSP heuristics (see Chapter 2.2.1) are homemade implementations integrated in the Gecode 2.1.1 [Gec06] constraint solver. *dyn-CS* was used as a heuristics model on the top of the heuristics[2] set $\mathcal{H} = \{$*dom-wdeg*, *wdeg*, *dom-deg*, *min-dom*, *impacts* $\}$, taking *min-value* as value selection heuristic. The cutoff value used to restart the search was initially set to 1000 and the cutoff increase policy to $\times 1.5$ (geometric factor), the same cutoff policy is used in all the experimental scenarios.

Continuous Search was assessed comparatively to the best two dynamic variable ordering heuristics on the considered problems, namely *dom-wdeg* and *wdeg*. It must be noted that Continuous Search, being a lifelong learning system, will depend on the curriculum, that is the order of the submitted instances. If the user "pedagogically" starts by submitting

[2]It is also important to notice that *domFD* is not considered in this chapter due to the complexity of computing weak dependencies might include an overhead when this heuristic is not used at a particular state of the search.

informative instances first, the performance in the first stages will be better than if untypical and awkward instances are considered first. For the sake of fairness, the performance reported for Continuous Search on each problem instance is the median performance over 10 random orderings of the CSP instances.

6.4.2 Practical performances

The first experimental scenario involves a timeout of 5 Minutes. Figure 6.4 highlights the Continuous Search results on Langford-number problems, comparatively to *dom-wdeg* and *wdeg*. The x-axis gives the number of problems solved and the y-axis presents the cumulated runtime. The (median) *dyn-CS* performance (grey line) is satisfactory as it solves 12 more instances than *dom-wdeg* (black line) and *wdeg* (light gray line). The dispersion of the *dyn-CS* results depending on the instance ordering is depicted from the set of dashed lines. Indeed traditional portfolio approaches such as [HHHLB06, SM07, XHHLB07] do not present such performance variations as they assume a complete set of training examples to be available beforehand.

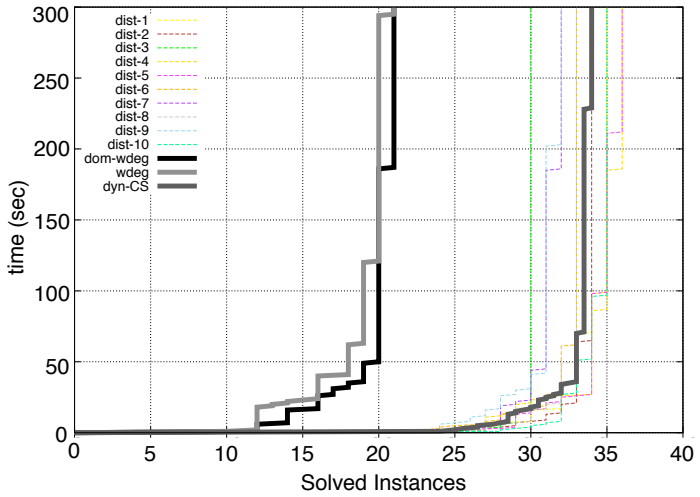


Figure 6.4: Langford-number (lfn)

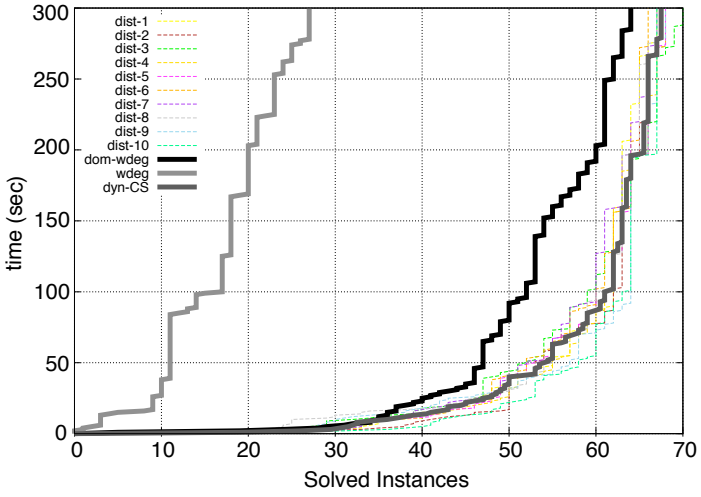


Figure 6.5: Geometric (geom)

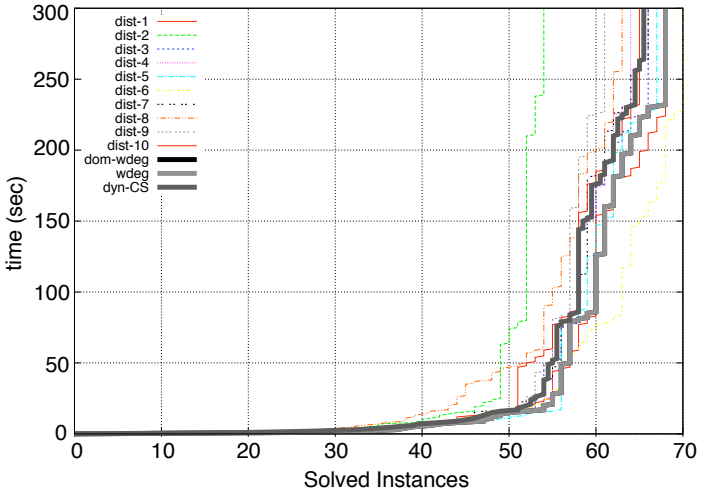


Figure 6.6: Balance incomplete block designs (bibd)

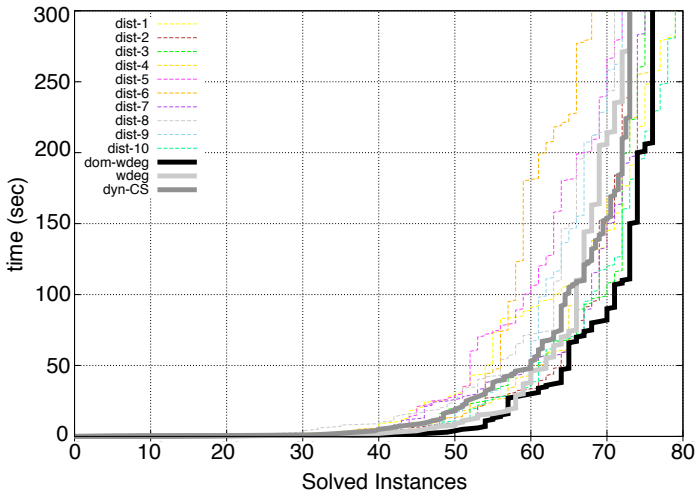


Figure 6.7: Job Shop (js)

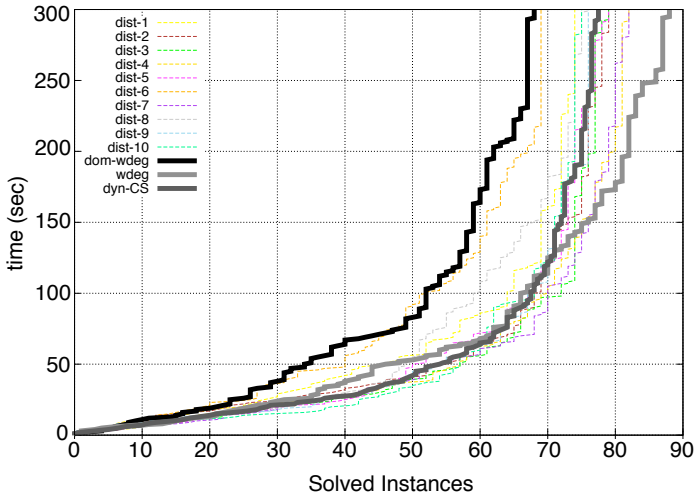


Figure 6.8: Nurse Scheduling (nsp)

Figures 6.5-6.8 depict the performance of *dyn-CS*, *dom-wdeg* and *wdeg* on all other problem families, respectively (bibd, js, nsp, and geom). On the bibd (Figure 6.6) and js (Figure 6.7) problems, the best heuristics is *dom-wdeg*, solving 3 more instances than *dyn-CS*. Note that *dom-wdeg* and *wdeg* coincide on bibd since all decision variables are boolean.

On nsp (Figure 6.8), *dyn-CS* solves 9 more problems than *dom-wdeg*, but is outperformed by *wdeg* by 11 problems. On geom (Figure 6.5), *dyn-CS* improves on the other heuristics, solving respectively 3 more instances and 40 more instances than *dom-wdeg* and *wdeg*.

These results suggest that *dyn-CS* is most often able to pick up the best heuristics on a given problem family, and sometimes able to significantly improve on the best of the available heuristics.

All experimental results concerning the first scenario are summarized in Table 6.1, reporting for each considered heuristics the number of instances solved (#sol), the total computational cost for all instances (time, in hour), the average time (avg-time, in minutes) per instance, and the number of instances for each problem family indicated in parentheses. It is important to notice that the best performing algorithm (w.r.t. runtime efficiency) is indicated in bold. These results confirm that *dyn-CS* outperforms *dom-wdeg* and *wdeg*, solving respectively 18 and 41 instances more out of 315. Furthermore, it shows that *dyn-CS* is slightly faster than the other heuristics, with an average time of 2.11 minutes, against respectively 2.39 for *dom-wdeg* and 2.61 for *wdeg*.

Problem	*dom-wdeg*			*wdeg*			*dyn-CS*		
	#sol	time(h)	avg-time(m)	#sol	time(h)	avg-time(m)	#sol	time(h)	avg-time(m)
nsp (100)	68	3.9	2.34	**88**	**2.6**	**1.56**	77	2.9	1.74
bibd (83)	**68**	**1.8**	**1.37**	68	1.8	1.37	65	2.0	1.44
js (130)	**76**	**4.9**	**2.26**	73	5.1	2.35	73	5.2	2.4
lfn (83)	21	5.2	3.75	21	5.3	3.83	**33**	**4.1**	**2.96**
geom (100)	64	3.9	2.34	27	6.8	4.08	**67**	**3.3**	**1.98**
Total (496)	297	19.7	2.39	274	21.6	2.61	**315**	**17.5**	**2.11**

Table 6.1: Total solved instances (5 Minutes)

The second experimental results using a timeout of 3 Minutes are presented in table 6.2, as can be observed, decreasing the time limit drastically reduce the total number of solved

instances for *dom-wdeg* and *wdeg*. Therefore, selecting the right heuristic becomes critical. Here *dyn-CS* is able to solve 24 and 45 more instances than *dom-wdeg* and *wdeg*.

Problem	*dom-wdeg*			*wdeg*			*dyn-CS*		
	#sol	time(h)	avg-time(m)	#sol	time(h)	avg-time(m)	#sol	time(h)	avg-time(m)
nsp (100)	61	2.8	1.68	**81**	**2.1**	**1.26**	75	2.2	1.32
bibd (83)	**62**	**1.3**	**0.94**	62	1.3	0.94	60	1.4	1.01
js (130)	**74**	**3.1**	**1.43**	69	3.3	1.52	67	3.4	1.57
lfn (83)	20	3.2	2.31	20	3.2	2.31	**32**	**2.5**	**1.81**
geom (100)	56	2.6	1.56	20	4.3	2.58	**63**	**2.2**	**1.32**
Total (496)	273	13.0	1.57	252	14.2	1.72	**297**	**11.7**	**1.42**

Table 6.2: Total solved instances (3 Minutes)

Another interesting lesson learned from the experiments concerns the difficulty of the underlying learning problem, and the generalization error of the learned hypothesis. The generalization error in the Continuous Search framework is estimated by 10-fold Cross Validation [BE93] on the whole training set (including all training examples generated in exploration mode). Table 6.3 reports on the predictive accuracy of the SVM algorithm (with same default setting) on all problem families, with an average accuracy of 67.8%. As could have been expected, the predictive accuracy is correlated to the performance of Continuous Search: the problems with best accuracy and best performance improvement are geom and lfn.

To give an order of idea, 62% predictive accuracy was reported in the context of SATzilla [XHHLB07], aimed at selecting of the best heuristic in a portfolio.

A direct comparison of the predictive accuracy might however be biased. On the one hand SATzilla errors are attributed to the selection of some near-optimal heuristics, after the authors; on the other hand, Continuous Search would involve several selection steps (in each checkpoint) and could thus compensate from earlier errors.

Timeout	bibd	nsp	geom	js	lfn	Total
3 Min	64.5%	64.2%	79.2%	65.6%	68.2%	68.3%
5 Min	63.2%	58.8%	76.9%	63.6%	73.8%	67.3%
Average	63.9%	61.5%	78.0%	64.6%	71.0%	**67.8%**

Table 6.3: Predictive Accuracy of the heuristics model (10-fold Cross Validation)

6.4.3 Exploration time

Now we turn our attention to the CPU time required to complete the exploration mode. Tables 6.4 and 6.5 show the total exploration time considering a timeout of five and three minutes for each problem family, the `median` value is computed taking into account all instance orderings and `instance` estimates the total exploration time for a single problem-instance.

As can be seen the time required to complete the exploration mode after solving a problem-instance is on average no longer than 2 hours. On the other hand, we would like to point out that since the majority of the instances for the bibd and geom problems can be quickly solved by *dyn-CS*, it is not surprising that the required time is significantly inferior compared with nsp, js and lfn.

Problem	Median	Instance
nsp	106.8	1.1
bibd	48.3	0.6
js	135.6	1.0
lfn	100.3	1.0
geom	37.6	0.4

Table 6.4: Exploration time in Hours (time-out 3 Minutes)

Problem	Median	Instance
nsp	151.1	1.5
bibd	73.6	0.9
js	215.6	1.7
lfn	161.8	1.9
geom	71.6	0.7

Table 6.5: Exploration time in Hours (time-out 5 Minutes)

6.4.4 The power of adaptation

Our third experimental test combines instances from different domains in order to show how CS is able to adapt to changing problems distribution. Indeed, unlike classical portfolio-based approaches which can only be applied if the training and exploitation sets come from

the same domain, CS can adapt to changes and provide top performances even if the problems change.

Problem	#Sol	time (h)
nsp-geom‡	55	4.1
nsp-geom†	67	3.4
lfn-bibd‡	23	5.3
lfn-bibd†	63	2.3

Table 6.6: Total solved instances (5 Minutes)

In this context, Table 6.6 reports the results on the geom (left) and bibd (right) problems by considering the following two scenarios. In the first scenario, we are going to emulate a portfolio-based search which would use the wrong domain to train. In *nsp-geom*‡, CS incrementally learns while solving the 100 nsp instances, and then solves one by one the 100 geom instances. However, when switching to this second domain, incremental learning is switched off, and checkpoints adaptation uses the model learnt on nsp. In the second scenario, *nsp-geom*† we solve nsp, then geom instances one by one, but this time, we keep the incremental learning on when switching from the first domain to the second one - as if CS was not aware of the transition.

As we can see in the first line of the Table, training on the wrong domain gives poor performance (55 instances solved in 4.1 hours). At contrary, the second line shows that CS can recover from training on the wrong domain thanks to its incremental adaptation (solving 67 instances in 3.4 hours). The right part of the Table reports similar results for the bibd problem.

As can be observed in nsp-geom† and lfn-bibd†, CS successfully identifies the new distribution of problems solving respectively the same number and 2 less instances than geom and bibd when CS is only applied to this domain starting from scratch. However the detection of the new distribution introduces an overhead in the solving time (see results for single domain in Table 6.1).

6.5 Previous Work

As pointed out in Chapter 3, the application of Machine Learning algorithms to build a portfolio solver has been widely studied during the last decade. Methods such as: SATzilla [XHHLB07], CPHYDRA [OHH+08], *self*-AQME [PT09], etc., typically require a representative set of training examples to properly learn a heuristic model to solve a set of testing instances.

Although previous mentioned strategies were designed for their respective competitions, i.e., SAT, CSP and QBF, some of them can also be used in incremental learning scenarios. However, at this point we would like to remark that the goal of this contribution is not only describing another methodology for using Machine Learning in the context of the *Algorithm Selection Problem*. Instead, we present the Continuous Search paradigm which uses computer's IDLE time to incrementally learn and tune the parameters of a constraint solver.

CPHYDRA is based on lazy learning which means that new cases (or samples) can be easily added to the heuristics model, however, after the training phase is completed each new case would be defined as a list L=[$\langle A_1,t_1\rangle$, $\langle A_2,t_2\rangle$, ..., $\langle A_n,t_n\rangle$]. L represents a switching policy to execute a selected subset of solvers, t_i indicates the time cutoff for the ith solver in L, and no communication is allowed between the solvers. Therefore, each new case after the training phase is a combination of solvers by itself. Thus, including such new case could be impractical as the system might suggest complex switching policies in the future. Nevertheless, the continuous search paradigm can also be used in this context to automatically identify the most informative cases during the exploration mode and defining new switching policies based on those important examples.

In contrast to *dyn-CS* which is proposed to identify the best CSP heuristic at different steps of the search, *self*-AQME was designed for a slightly different context, i.e., QBF problems. This portfolio solver updates the heuristic model after processing each training example during the training phase. This procedure can be extended to the testing phase, however, we foresee two main difficulties. On the one hand, if the expected best solver finds a solution, the heuristics model is not updated even if there exists another solver with better runtime. On the other hand, if the runtime cutoff for each solver candidate is

not properly defined, the heuristics model can be misled with sub-optimal solvers, again because no other heuristic is tried after a solution is reached. These two main disadvantages can be overcome by using the continuous search paradigm and exploiting computer's IDLE time to obtain the *true winner* solver. Moreover, the exploration mode can also be used to update the runtime cutoff parameter for each solver candidate by considering the current distribution of problems.

Hydra [XHLB10] iteratively exploits highly parameterized algorithms (algorithms with hundreds of thousands of parameters) to incrementally obtain promising configurations for such algorithm. This algorithm represents an important contribution to the well-known SATzilla portfolio solver, however, unfortunately such parameterized algorithm does not exists yet to solve CSPs.

Finally, in [CB05] low-knowledge is used to select the best algorithm in the context of optimization problems, this work assumes a black-box optimization scenario where the user has no information about the problem or even about the domain of the problem, and the only known information is the output (i.e., solution cost for each algorithm in the portfolio). Unfortunately, this mechanism is only applicable to optimization problems and cannot be straightforward used to solve CSPs.

6.6 Summary

The main contribution of the presented approach, the Continuous Search framework aims at designing a heuristics model tailored to the user problem distribution, allowing her to get top performance from the constraint solver. The representative instances needed to train a good heuristics model are not assumed to be available beforehand; they are gradually built and exploited to improve the current heuristics model, by stealing the idle CPU cycles of the computing system. Metaphorically speaking, the constraint solver uses its spare time to play against itself and gradually improve its strategy along time; further, this expertise is relevant to the real-world problems considered by the user, all the more so as it directly relates to the problem instances submitted to the system.

The experimental results suggest that Continuous Search is able to pick up the best of a set of heuristics on a diverse set of problems, by exploiting the incoming instances; in

2 out of 5 problems, Continuous Search swiftly builds up a mixed strategy, significantly overcoming all baseline heuristics. With the other classes of problems, its performance is comparable to the best two single heuristics. Our experiments also showed the capacity of adaptation of CS. Moving from one problem domain to another one is possible thanks to its incremental learning capacity. This capacity is a major improvement against classical portfolio-based approaches which only work when offline training and exploitation use instances from the same domain.

Chapter 7

Efficient Parallel Local Search for SAT

Up to now, in this thesis we have explored different approaches to solve combinatorial problems in sequential settings. In this chapter, our objective is to study the impact of knowledge sharing on the performance of portfolio-based parallel local search algorithms. Our motivation is the demonstrated importance of clause-sharing in the performance of complete parallel SAT solvers. Unlike complete solvers, state-of-the-art local search algorithms for SAT are not able to generate redundant clauses during their execution. In our settings, each member of the portfolio shares its best configuration (i.e., one which minimizes conflicting clauses) in a common structure. At each restart point, instead of classically generating a random configuration to start with, each algorithm aggregates the shared knowledge to carefully craft a new starting point.

7.1 Introduction

Complete parallel solvers for the propositional satisfiability problem have received significant attention recently. These solvers can be divided into two main categories the classical divide-and-conquer model and the portfolio-based approach. The first one, typically divides the search space into several sub-spaces while the second one lets algorithms compete on the original formula [HJS09]. Both take advantage of the modern SAT solving architecture [MMZ$^+$01], to exchange the conflict-clauses generated in the system and improve the overall performance.

This push towards parallelism in complete SAT solvers has been motivated by their practical applicability. Indeed, many domains, from software verification to computational biology and automated planning rely on their performance. On the contrary, since local search techniques only outperform complete ones on random SAT instances, their parallelizing has not received much attention so far. The main contribution on the parallelization of local search algorithms for SAT solving basically executes a portfolio of independent algorithms which compete without any communication between them. In our settings, each member of the portfolio shares its best configuration (i.e., one which minimizes the number of conflicting clauses) in a common structure. At each restart point, instead of classically generating a random configuration to start with, each algorithm aggregates the shared knowledge to carefully craft a new starting point. We present several aggregation strategies and evaluate them on a large set of instances. Our best strategies largely improve over a parallel portfolio of non cooperative local searches. We also present an analysis of configurations diversity during parallel search, and find out that the best aggregation strategies are the one which are able to maintain a good diversification/intensification trade off.

This chapter is organized as follows: Section 7.2 presents our methodology and our aggregation strategies, Section 7.3 evaluates them, Section 7.4 highlights previous work on parallel SAT and cooperative algorithms, and Section 7.5 presents a summary of the chapter.

7.2 Knowledge Sharing in Parallel Local Search for SAT

Our objective is to extend a parallel portfolio of state-of-the-art local search solvers for SAT with knowledge sharing or cooperation. Each algorithm is going to share with others the best configuration it has found so far with its respective cost (number of unsatisfied clauses) in a shared pair $\langle M, C \rangle$.

$$M = \begin{pmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ X_{c1} & X_{c2} & \dots & X_{cn} \end{pmatrix} \qquad C = [C_1, C_2, \dots, C_c]$$

Where n indicates the total number of variables of the problem and c indicates the number of local search algorithms in the portfolio. In the following we are associating local search algorithms and processing cores. Each element X_{ji} in the matrix indicates the i^{th} variable of the best configuration found so far by the j^{th} core. Similarly, the j^{th} element in C indicates the cost for the respective configuration in M.

These best configurations can be exploited by each local search to build a new initial configuration. In the following, we propose seven strategies to determine the initial configuration (cf. function *initial-configuration* in Algorithm 2.2, Chapter 2).

7.2.1 Using Best Known Configurations

In this section, we propose three methods to build the new initial configuration *init* by aggregating best known configurations. In this way, we define $init_i$ for all the variables $X_i, i \in [1..n]$ as follows:

1. *Agree*: if there exists a value v such that v=X_{ji} for all $j \in [1..c]$ then $init_i$=v, otherwise a random value is used.

2. *Majority*: if there exists two values v and v' such that $|\{X_{ji} = v | j \in [1..c]\}| > |\{X_{ji} = v' | j \in [1..c]\}|$ then $init_i$=v, otherwise a random value is used.

3. *Prob*: $init_i$=1 with probability p_{ones}=$\frac{ones}{c}$ and $init_i$=0 with probability $1-p_{ones}$, where $ones = |\{X_{ji} = 1 | j \in [1..c]\}|$.

7.2.2 Weighting Best Known Configurations

In contrast with our previous methods where all best known solutions are treated equally important, the methods proposed in this section use a weighting mechanism to consider the cost of best known configurations. The computation of the initial configuration *init* uses one of the following two weighting systems: *Ranking* and *Normalized Performance*, where values from better configurations are most likely to be used.

7.2.2.1 Ranking

This method sorts the configurations of the shared matrix from worst to best according to their cost. The worst ranked one gets weight of 1 (i.e., $RankW_1$=1), and the best ranked c (i.e., $RankW_c$=c).

7.2.2.2 Normalized Performance

This method assigns weights ($NormW$) considering a normalized value of the number of unsatisfied clauses of the configuration:

$$NormW_j = \frac{|\mathcal{C}| - C_j}{|\mathcal{C}|}$$

Using the previous two weighting mechanisms, we define the following four extra methods to determine initial configurations.

To this end, we define $\Phi(val, Weight) = \sum_{k \in \{j|X_{ji}=val\}} Weight_k$.

1. *Majority RankW*: if there exists two values v and v' such that $\Phi(v, RankW) > \Phi(v', RankW)$ then *init*$_i$=v, otherwise a random value is used.

2. *Majority NormalizedW*: if there exists two values v and v' such that $\Phi(v, NormW) > \Phi(v', NormW)$ then *init*$_i$=v, otherwise a random value is used.

3. *Prob RankW*: *init*$_i$=1 with probability P_{Rones}=$\frac{Rones}{Rones+Rzeros}$ and *init*$_i$=0 with probability 1-P_{Rones}, where $Rones$=$\Phi(1, RankW)$ and $Rzeros$=$\Phi(0, RankW)$.

4. *Prob NormalizedW*: *init*$_i$=1 with probability P_{Nones}=$\frac{Nones}{Nones+Nzeros}$ and *init*$_i$=0 with probability 1-P_{Nones}, where $Nones$=$\Phi(1, NormW)$ and $Nzeros$=$\Phi(0, NormW)$

7.2.3 Restart Policy

As mentioned earlier on, shared knowledge is exploited when a given algorithm is restarted. At this point the current working configuration of a given algorithm is re-initialized according to a given aggregation strategy. However, it is important to restrict cooperation since it adds overheads and more importantly tend to generate similar configurations. In this

context, we propose a new restart policy to avoid re-initializing the working configuration again and again. This new policy re-initializes the working configuration for a given restart (i.e., every MaxFlips) if and only if, performance improvements in best known solutions have been observed during the latest restart window. This new restart policy is formally described in the following definition, where we assume that bc_{ki} is the cost of the best known configuration for a given algorithm i up to the $(k-1)^{th}$ restart.

Definition 7.1 *At a given restart k for a given algorithm i the working configuration is reinitialized iff there exists an algorithm q such that $bc_{kq} \neq bc_{(k-1)q}$ and $q \neq i$.*

7.3 Experiments

This section reports on the experimental validation of the proposed aggregation strategies.

7.3.1 Experimental Settings

We conducted experiments using instances from the RANDOM category of the 2009 SAT competition. Since state-of-the-art local search solvers are unable to solve UNSAT instances, we filtered out these instances. We also removed instances whose status was reported as UNKNOWN in the competition. This way, we collected 359 satisfiable instances.

We decided to build our parallel portfolio on UBCSAT-1.1, a well known local search library which provides efficient implementation of the latest local search for SAT algorithms [TH04]. We did preliminary experiments to extract from this library the 8 algorithms which perform best on our set of problems. From that, we defined the following three baseline portfolio constructions where algorithms are independent searches without cooperation. The first one *$p$cores-PAWS* uses p copies of the best single algorithm (PAWS), the second portfolio *4cores-No sharing* uses the best subset of 4 algorithms (PAWS, G2+p, AG2, AG2+p) and the last one *8cores-No sharing* uses all the 8 algorithms (PAWS, G2+p, AG2, AG2+p, G2, SAPS, RSAPS, AN+). All the algorithms were used with their default parameters, and without any restart. Indeed these techniques are equipped with important diversification strategies and usually perform better when the restart flag is switched off (i.e., MaxFlips=∞).

On the other hand, the previous knowledge aggregation mechanisms were built on top of a portfolio with 4 algorithms (same algorithms as *4cores-No sharing*) and a portfolio with 8 algorithms (same algorithms as *8cores-No sharing*). There, we used the modified restart policy described in Section 7.2.3 with $MaxFlips$ set to 10^6.

All tests were conducted on a cluster of 8 Linux Mandriva machines with 8 GB of RAM and two quad-core (8 cores) 2.33 Ghz Intel Processors. In all the experiments, we used a timeout of 5 minutes (300 seconds) for each algorithm in the portfolio, so that for each experiment the total CPU time was set to $c \times 300$ seconds, where c indicates the number of algorithms in the portfolio.

We executed each instance 10 times (each time with a different random seed) and reported two metrics, the *Penalized Average Runtime* (PAR) [HHLB10] which computes the average runtime overall instances, but where unsolved instances are considered as $10\times$ the cutoff time, and the runtime for each instance which is calculated as the median across the 10 runs. Overall, our experiments for these 359 SAT instances took 187 days of CPU time.

7.3.2 Practical Performances with 4 Cores

Figure 7.1(a) shows the results of each aggregation strategy using a portfolio with 4 cores, comparatively to the 4 cores baseline portfolios. The x-axis gives the number of problems solved and the y-axis presents the cumulated runtime.

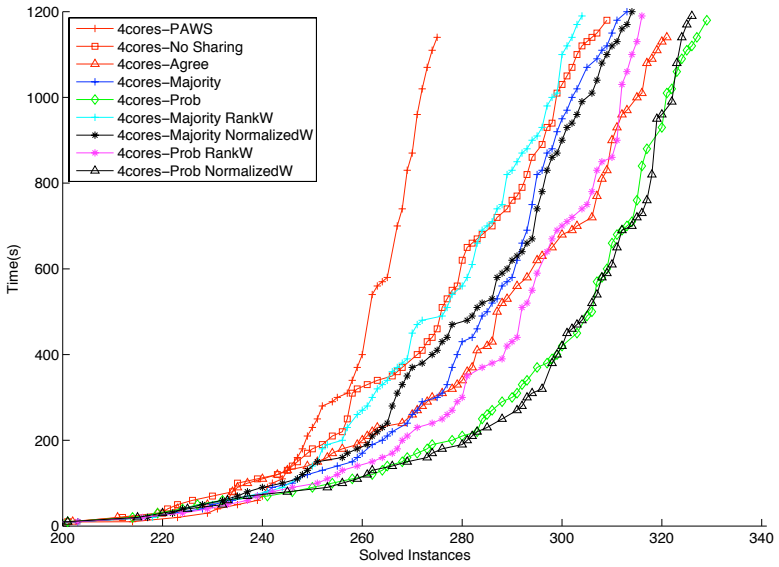
As expected, the portfolio with the top 4 best algorithms (*4cores-No Sharing*) performs better (309) that the one with 4 copies of the best algorithms (*4cores-PAWS*) (275). Additionally, Figure 7.1(b) shows the performance when considering the PAR metric. The y-axis shows the *Penalized Average Runtime* for a given time cutoff given on the x-axis. In this figure, it can be observed that the aggregation policies are also efficient when varying the time limit to solve problem instances.

The performance of the portfolios with knowledge sharing is quite good. Overall, it seems that adding a weighting mechanism can often hurt the performance of the underlying aggregation strategy. Among the weighting options, it seems that the Normalized Performance performs better. The best portfolio implements the *Prob* strategy without any weighting (329). This corresponds to a gain of 20 problems against the corresponding

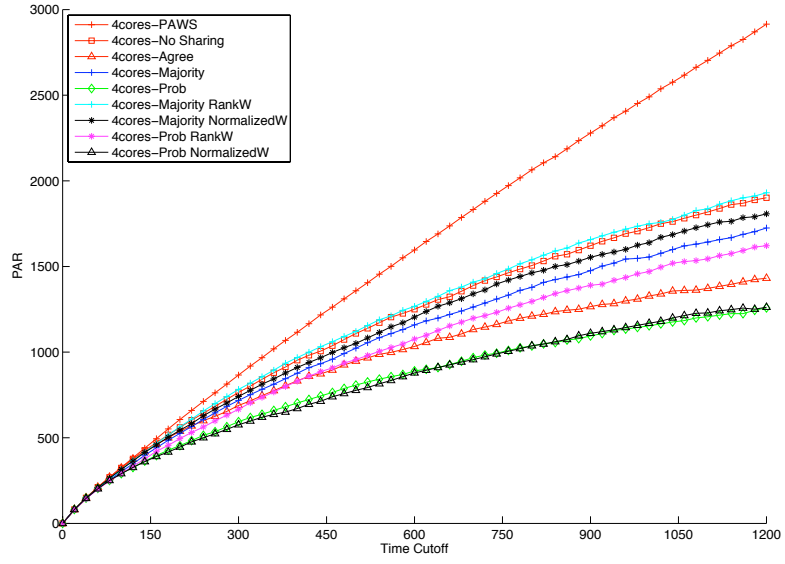
4cores-No Sharing baseline.

A detailed examination of *4cores-Prob* and *4cores-No Sharing* is presented in Figures 7.2 and 7.3. These figures show, respectively, a runtime and a best configuration cost comparison. In both figures, points below (resp. above) the diagonal line indicate that *4cores-Prob* performs better (resp. worse) than *4cores-No Sharing*. In the runtime comparison, we observe that easy instances are correlated as they require few steps to be solved, and for the remaining set of instances *4cores-Prob* usually exhibits a better performance. On the other hand, the second figure shows that when the instances are not solved, the median cost of the best configuration (number of unsatisfied clauses) found by *4cores-Prob* is usually better than for *4cores-No Sharing*. Notice that some points are overlapped because the two strategies reported the same cost.

All the experiments using 4 cores are summarized in Table 7.1, reporting for each portfolio the number of solved instances (#solved), the median time across all instances (median time), the *Penalized Average Runtime* (PAR) and the total number of instances that timed out in all the 10 runs (never solved). These results confirm that sharing best known configurations outperforms independent searches, for instance *4cores-Prob* and *4cores-Prob NormalizedW* solved respectively 20 and 17 more instances than *4cores-No Sharing* and all the cooperative strategies (except *4cores-Majority RankW*) exhibit better PAR. Interestingly, 4cores-PAWS exhibited the best median runtime overall the experiments with 4 cores, this fact suggests that PAWS by itself is able to quickly solve an important number of instances. Moreover, only 2 instances timeout in all the 10 runs for *4cores-Agree* and *4cores-Prob NormalizedW* against 7 for *4cores-No Sharing*. Notice that this Table also includes *1core-PAWS*, the best sequential local search on this set of problems. The PAR score for *1core-PAWS* is lower than the other values of the table because this portfolio uses only 1 algorithm, therefore the timeout is only 300 seconds, while 4 cores portfolios use a timeout of 1200 seconds. Notice that the best performing strategy (w.r.t. each performing metric) is indicated in bold.



(a) Number of solved instances



(b) Penalized Average Runtime

Figure 7.1: Performance using 4 cores in a given amount of time

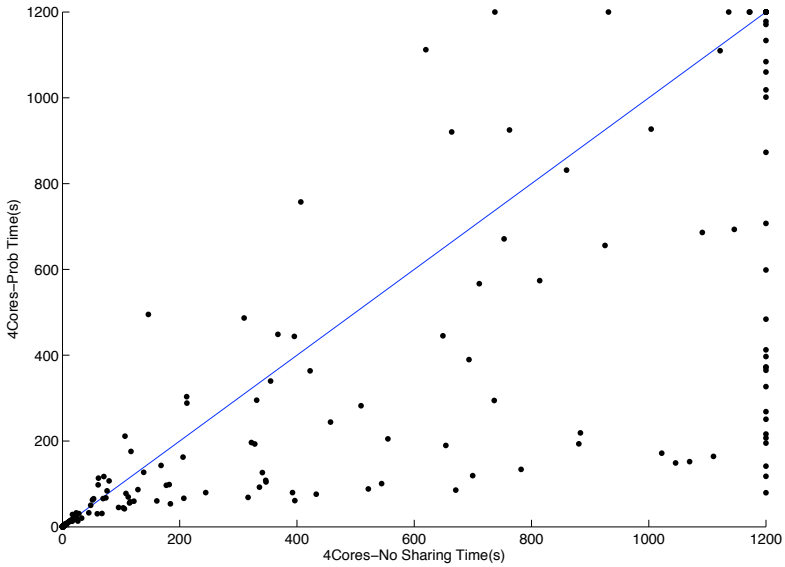


Figure 7.2: Runtime comparison, each point indicates the runtime to solve a given instance using *4cores-Prob* (y-axis) and *4cores-No Sharing* (x-axis)

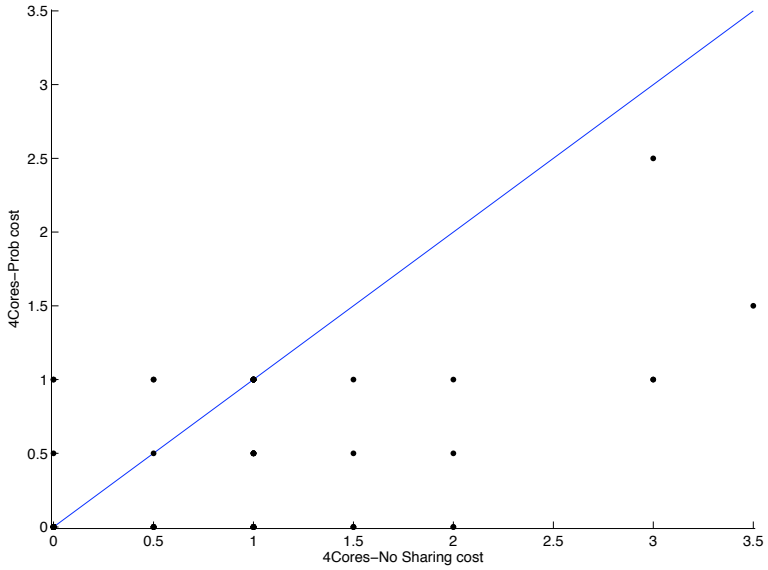


Figure 7.3: Best configuration cost comparison on unsolved instances. Each point indicates the best configuration (median) cost of a given instance using *4cores-Prob* (y-axis) and *4cores-No Sharing* (x-axis)

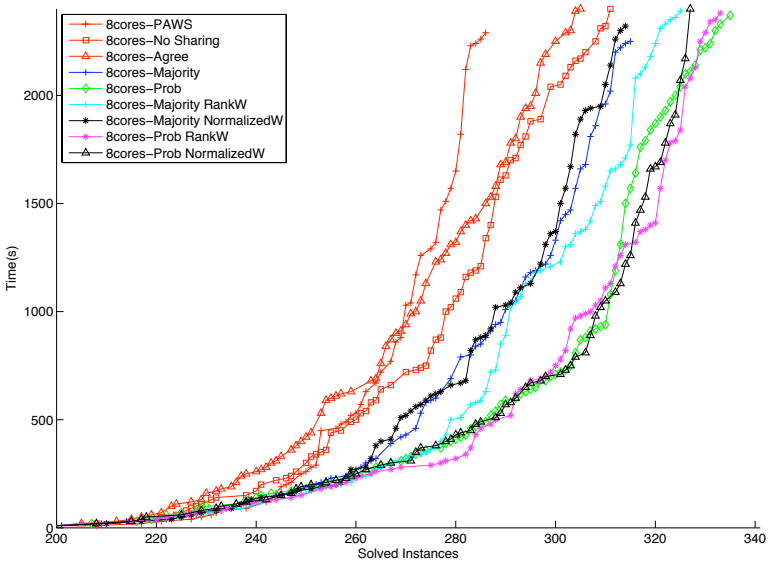
Strategy	#solved	median time	PAR	never solved
1core-PAWS	249	1.76	911.17	71
4cores-PAWS	275	**1.63**	2915.19	61
4cores-No Sharing	309	2.19	1901.00	7
4cores-Agree	321	2.54	1431.33	**2**
4cores-Majority	313	2.53	1724.94	11
4cores-Prob	**329**	2.51	**1257.93**	4
4cores-Majority RankW	304	2.47	1930.61	11
4cores-Majority NormalizedW	314	2.48	1807.42	9
4cores-Prob RankW	316	2.53	1621.33	7
4cores-Prob NormalizedW	326	2.50	1261.82	**2**

Table 7.1: Overall evaluation using 4 cores

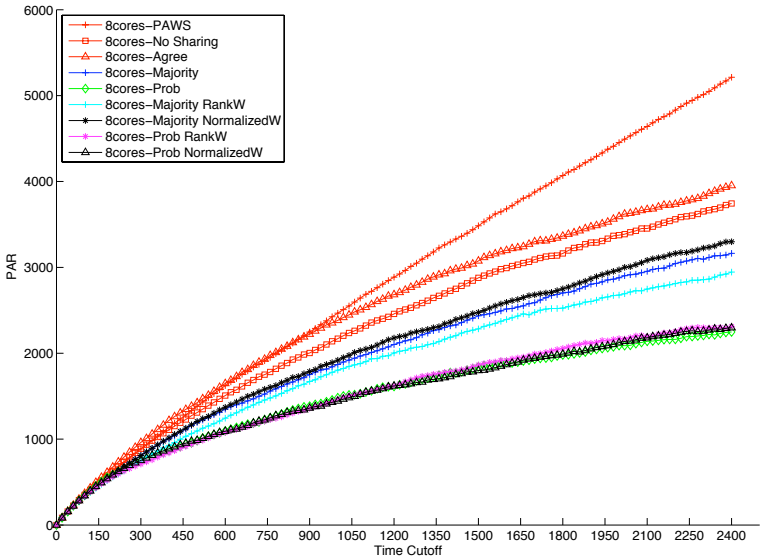
7.3.3 Practical Performances with 8 Cores

We now move on to portfolios with 8 cores. The results of these experiments are depicted in Figure 7.4 indicating the total number of solved instances within a given amount of time. As in previous experiments, we report the results of baseline portfolios *8cores-No Sharing* and *8cores-PAWS*, as well as the seven cooperative strategies. We can observe than the cooperative portfolios (except *8cores-Agree*) largely outperform the non-cooperative ones in both the number of solved instances (Figure 7.4(a)) and the PAR metric (Figure 7.4(b)). Indeed, as it will be detailed in Section 7.3.4, *8cores-Agree* exhibits a poor performance mainly because best known configurations stored in the shared data structure tend to be different from each other. Therefore, this policy tends to generate completely random starting points, and cannot exploit the acquired knowledge.

The Table 7.2 summarizes these results, and once again it includes the best individual algorithm running in a single core. We can remark that *8cores-Prob* , *8cores-Prob RankW* , and *8cores-Prob NormalizedW* solve respectively 24, 22, and 16 more instances than *8cores-No Sharing*. Furthermore, it shows that knowledge sharing portfolios are faster than individual searches, with a PAR of 3743.63 seconds for *8cores-No Sharing* against respectively 2247.97 for *8cores-Prob* , 2312.80 for *8cores-Prob RankW* and 2295.99 for *8cores-Prob NormalizedW* . Finally, it is also important to note that only 1 instance timed out in all the 10 runs for *8cores-Prob NormalizedW* against 8 for *8cores-No Sharing*.



(a) Number of solved instances



(b) Penalized Average Runtime

Figure 7.4: Performance using 8 cores in a given amount of time

Strategy	#solved	median time	PAR	never solved
1core-PAWS	249	1.76	911.17	71
8cores-PAWS	286	**2.00**	5213.84	56
8cores-No Sharing	311	2.33	3743.63	8
8cores-Agree	305	2.48	3952.19	17
8cores-Majority	315	2.47	3163.02	6
8cores-Prob	**335**	2.45	**2247.97**	2
8cores-Majority RankW	325	2.39	2944.92	4
8cores-Majority NormalizedW	314	2.54	3298.60	9
8cores-Prob RankW	333	2.55	2312.80	2
8cores-Prob NormalizedW	327	2.47	2295.99	**1**

Table 7.2: Overall evaluation using 8 cores

These experimental results show that *Prob* (4 and 8 cores) exhibited the overall best performance. We attribute this to the fact that the probability component of this method balances the exploitation of best solutions found so far with the exploration of other values for the variables, helping in this way, to diversify the new starting configuration.

7.3.4 Analysis of the Diversification/Intensification Trade off

Maintaining an appropriate balance between *diversification* and *intensification* of the acquired knowledge is an important step of the proposed cooperative portfolios to improve the performance. In this chapter, *diversification* (resp. *intensification*) refers to the ability of generating different (resp. similar) initial configuration at each restart.

Figure 7.5 aims to analysis the balance between *diversification* and *intensification* by means of computing the average Hamming distance between all pairs of best known configurations (*HamDis*) vs the number of flips for a typical SAT instance. Notice that some lines are of different sizes because some strategies required less flips to solve the instance. *HamDis* is formally described as follows:

$$HamDis = \frac{\sum_{i=1}^{c} \sum_{j=i+1}^{c} Hamming(X_i, X_j)}{c(c-1)/2}$$

Where, $Hamming(X_i, X_j)$ indicates the Hamming distance between the best configurations found so far for the $i^{th}$and j^{th} algorithms in the portfolio.

The Figure 7.5(a) shows the *diversification - intensification* analysis using 4 cores. As one might have expected, among the cooperative strategies *4cores-Majority* reduces *diversification* and shows a high convergence rate, *4cores-Agree* reduces *intensification* and shows a slow convergence rate. In contrast to these two methods, *4cores-Prob* is balancing *diversification* and *intensification*. This phenomenon helps to understand the superiority of this method in the experiments presented in Section 7.3.2.

A similar observation is drawn from the experiments with 8 cores presented in Figure 7.5(b). However, in this case *8cores-Agree* exhibits a dramatic diversification increase which actually degrades its overall performance compared against its counterpart portfolio with 4cores (see Table 7.2). Additionally, Figure 7.5(c) shows the behavior of *8cores-Majority NormalizedW* and *8cores-Prob NormalizedW* , while Figure 7.5(d) shows the behavior of *8cores-Majority RankW* and *8cores-Prob RankW* . From these two last figures, it can be observed that *Majority*-based strategies provide less diversification than the *Prob*-based ones.

Now we switch our attention to Table 7.3, where we extend our analysis to all problem instances. To this end, we launched an extra run for each portfolio strategy to compute *HamIns*, which is formally defined as follows:

$$\textit{HamIns}(i) = \frac{\overline{HamDis}(i)}{\textit{total-vars}(i)} \times 100$$

Where, $\overline{HamDis}(i)$ computes the mean overall *HamDis* values achieved when solving i and *total-vars(i)* indicates the number of variables involved in i. This way, $\overline{HamIns}$ reports the mean *HamIns* over all the instances that required at least 10^6 flips to be solved. Notice that instances requiring less flips do not employ cooperation because the first restart is not reached. On the other hand, strategies reporting the highest degree of *intensification* (resp. *diversification*) using 4 and 8 cores are indicated in bold.

As can be observed, *prob*-based strategies have shown the best performance as they balance *diversification-intensification*. For instance, excluding *4cores-agree* which is already known that provides more diversification than intensification, *4cores-prob* provides

the highest $\overline{HamIns}$ variation among all the cooperative portfolios using 4 cores. Moreover, *Majority*-based strategies are bad for diversification as they might tend to start with a configuration similar to the one given by the best single algorithm. It is also worth mentioning that our baseline portfolios *4cores-PAWS* and *4cores-No Sharing* exhibit the highest values, which is not surprising as no cooperation is allowed.

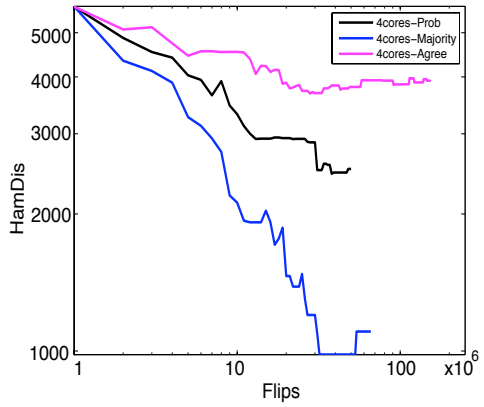
On the other hand, a similar observation is seen in the case of 8 cores. However, it is worth mentioning that *8cores-agree* gives too much diversification which actually degrades the overall performance when compared against its counterpart with 4 cores (see Tables 7.2 and 7.1).

Finally, Figure 7.6 shows a trace of the best configuration cost found so far for each algorithm in the portfolio to solve a typical instance. The x-axis shows the best solution for each algorithm vs the number of flips. The right part of the figure shows the performance of individual searches using 4 and 8 cores without cooperation, while the left part depicts the performance of *4cores-Prob* and *8cores-Prob*. As expected, non-cooperative algorithms exhibit different behaviors, for instance Figure 7.6(d) shows that SAPS and RSAPS are still far from the solution after reaching the timeout, while Figure 7.6(c) shows that using cooperation all the algorithms (including SAPS and RSAPS) are pushed to promising areas of the search.

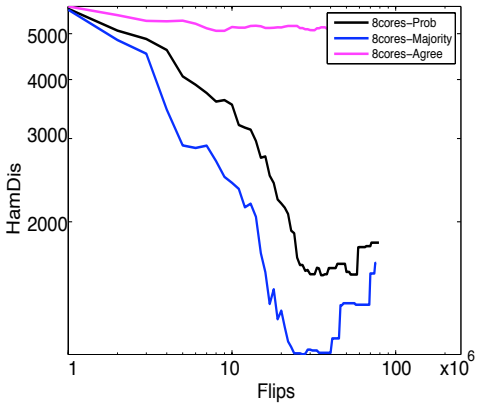
7.3.5 Analysis of the Limitations of the Hardware

In this section, we wanted to assess the inherent slowdown caused by increased cache, and bus contingency when more processing cores are used at the same time. Indeed, having an understanding of this slowdown is helpful to assess the real benefits of parallel search. To this end we decided to run our PAWS baseline portfolio where each independent algorithm uses the same random seed on respectively 1, 4 and 8 cores. Since all the algorithms are executing the same search, this experiment measures the slowdown caused by hardware limitations. The results are presented in Figure. 7.7.

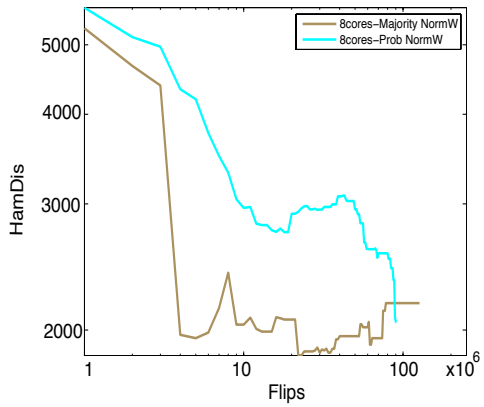
The first case executes a single copy of PAWS with a timeout of 300 seconds, the second case executes 4 parallel copies of PAWS with a timeout of 1200 seconds (4×300) and the third case executes 8 parallel copies of PAWS with a timeout of 2400 seconds (8×300).



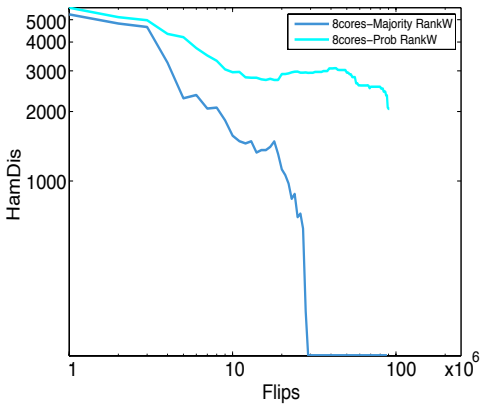
(a) 4cores-Prob, 4cores-Majority, 4cores-Agree



(b) 8cores-Prob, 8cores-Majority, 8cores-Agree



(c) 8cores-Majority NormW, 8cores-Prob NormW

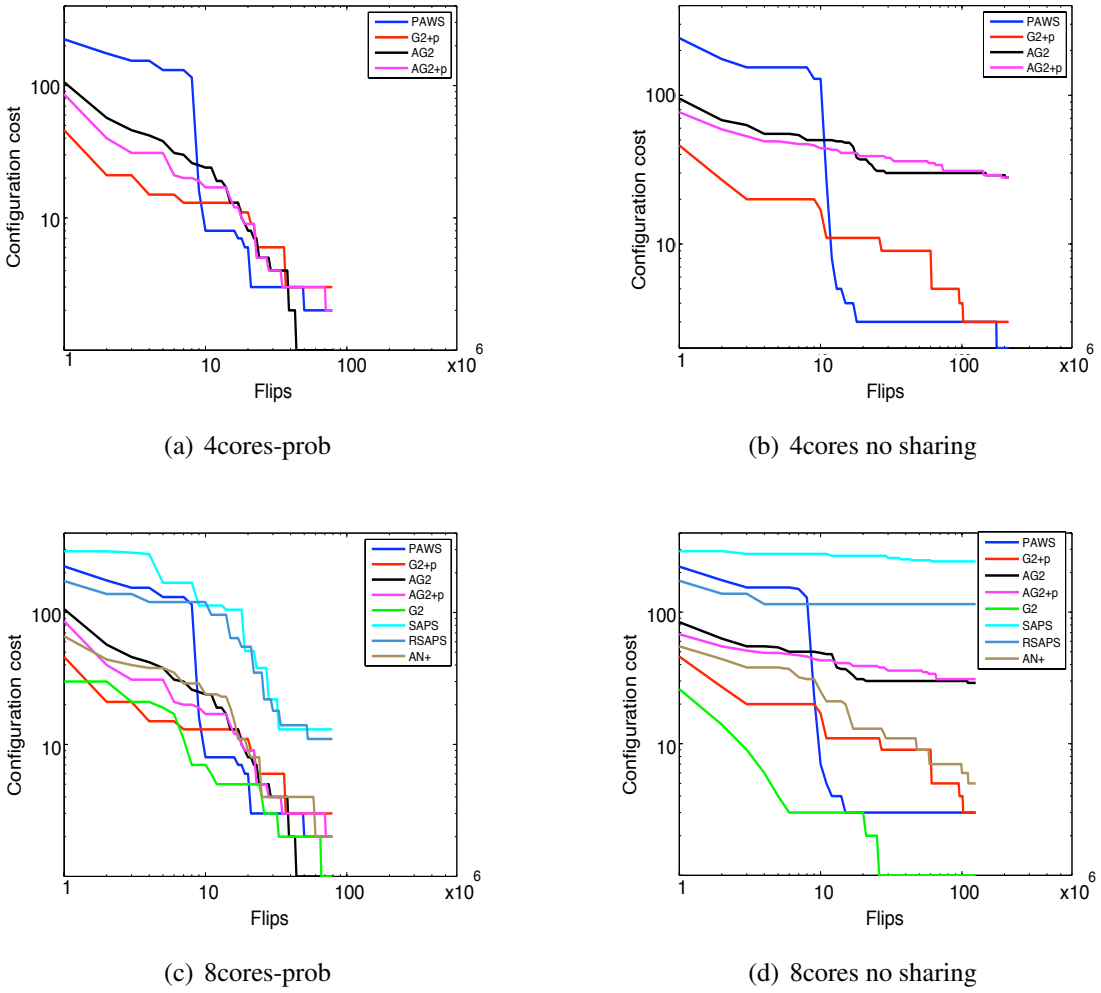


(d) 8cores-Majority RankW, 8cores-Prob NormW

Figure 7.5: Pairwise average Hamming distance (x-axis) vs Number of flips every 10^6 steps (y-axis) to solve the *unif-k3-r4.2-v16000-c67200-S2082290699-014.cnf* instance

Strategy	$\overline{HamIns}$
4cores-PAWS	38.2
4cores-No Sharing	**39.0**
4cores-Agree	35.0
4cores-Majority	31.7
4cores-Prob	33.1
4cores-Majority RankW	**25.9**
4cores-Majority NormalizedW	27.1
4cores-Prob RankW	30.8
4cores-Prob NormalizedW	32.8
8cores-PAWS	38.3
8cores-No Sharing	**39.5**
8cores-Agree	38.3
8cores-Majority	30.8
8cores-Prob	33.4
8cores-Majority RankW	**29.3**
8cores-Majority NormalizedW	29.5
8cores-Prob RankW	33.1
8cores-Prob NormalizedW	33.8

Table 7.3: *Diversification-Intensification* analysis using 8 cores



(a) 4cores-prob

(b) 4cores no sharing

(c) 8cores-prob

(d) 8cores no sharing

Figure 7.6: Individual algorithms performance to solve the *unif-k3-r4.2-v16000-c67200-S2082290699-014.cnf* instance

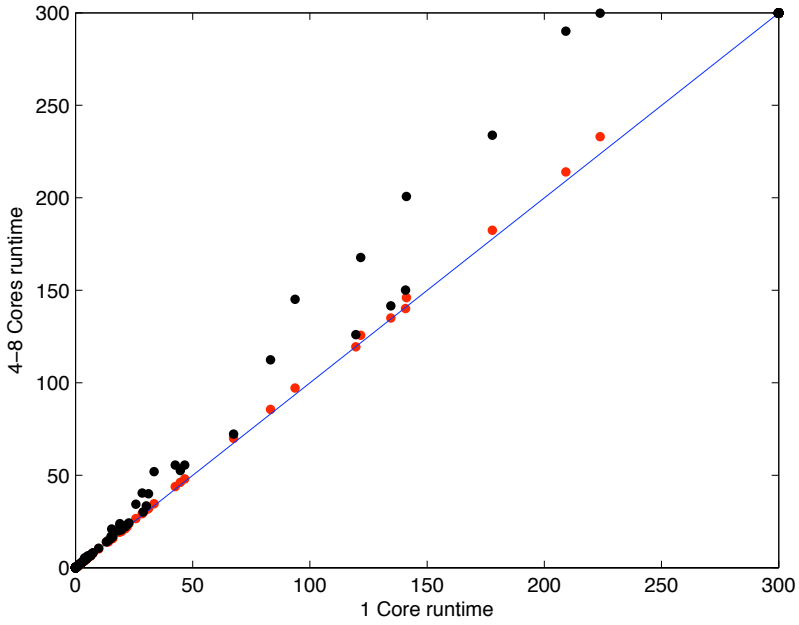


Figure 7.7: Runtime comparison using parallel local search portfolios made of respectively 1, 4, and 8 identical copies of PAWS (same random seed). Red points indicate the performance of 4 cores vs 1 core. Black points indicate the performance of 8 cores vs 1 core, points above the blue line indicate that 1 core is faster

Finally, we estimate the runtime of each instance as the median across 10 runs (each time with the same seed) divided by the number of cores. In this figure, it can be observed that the performance overhead is almost not distinguishable between 1 and 4 cores (red points). However, the overhead between 1 and 8 cores is important for difficult instances (black points).

This simple test can help us to assess the remarkable performance of our aggregation techniques. Indeed, on 8 cores, the best technique is able to solve 86 more problems than the sequential search. This is achieved despite the slowdown caused by cache and bus contingencies revealed by this experiment.

7.4 Previous Work

In this section, we review the most important contributions devoted to parallel SAT solving and cooperative algorithms.

7.4.1 Complete Methods for Parallel SAT

GrADSAT [CW06] is a parallel SAT solver based on the zChaff solver and equipped with a master-slave architecture in which the problem space is divided into sub-spaces, these sub-spaces are solved by independent zChaff clients and learnt clauses whose size (i.e., number of literals) is less or equal to a given limit are exchanged between clients. The technique organizes load-balancing through a work stealing technique which allows the master to push work to idle clients.

In [SLB09] the authors propose PaMiraXT a SAT solver with two layers of parallelization. While on the one hand, the traditional *Message Passing Interfase* (MPI) is used to coordinate the execution of independent workstations in a master/client mode. On the other hand, each client implements MiraXT, a parallel SAT solver which uses the well-known concept of *guiding paths* to divide the search space into several sub-spaces. Each independent MiraXT client (or workstation) uses a local *Shared Clause Database* which systematically stores learnt clauses by each core, and a selected subset of those clauses are sent to the master workstation which checks the consistency of the received information

with the *guided paths* to finally broadcast important clauses to all clients.

Unlike other parallel solvers for SAT which divide the initial problem space into sub-spaces, ManySAT [HJS09] is a portfolio-based parallel solver where independent DPLL algorithms are launched in parallel to solve a given problem instance. Each algorithm in the portfolio implements a different and complementary restart strategy, polarity heuristic and learning scheme. In addition, the first version of the algorithm exchanges learnt clauses whose size is less or equal to a given limit. It is worth mentioning that ManySAT won the 2008 SAT Race, the 2009 SAT Competition and was placed second in the 2010 SAT Race (all these in the parallel track – industrial category). Interestingly all the algorithms successfully qualified in the 2010 parallel track were based on a Portfolio architecture.

In plingeling, [Bie10] the original SAT instance is duplicated by a boss thread and allocated to worker threads. The strategies used by these workers are mainly differentiated around the amount of pre-processing, random seeds, and variables branching. Conflict clause sharing is restricted to units which are exchanged through the boss thread. This solver won the parallel track of the 2010 SAT Race.

In [ZHZ02] the authors proposed a hybrid algorithm which starts with a traditional DPLL algorithm to divide the problem space into sub-spaces. Each sub-space is then allocated to a given local search algorithm (Walksat).

7.4.2 Incomplete Methods for Parallel SAT

PGSAT [Rol02] is a parallel version of the GSAT algorithm. The entire set of variables is randomly divided into τ subsets and allocated to different processors. In this way at each iteration, if no global solution has been obtained, the i^{th} processor uses the GSAT score function (see Chapter 2) to select and flip the best variable for the i^{th} subset. Another contribution to this parallelization architecture is described in [RBB05] where the authors aim to combine PGSAT and random walk, therefore at each iteration, with a given probability wp an unsatisfiable clause c is selected and a random variable from c is flipped and with probability *1-wp*. PGSAT is used to flip τ variables in parallel at a cost of reconciling partial configurations to test if a solution has been found.

gNovelty+ (v.2) [PG09], belongs to the portfolio approach, this algorithm executes n

independent copies of the *gNovelty+ (v.2)* algorithm in parallel, until at least one of them finds a solution or a given timeout is reached. This algorithm was the only parallel local search solver presented in the *random* category of the 2009 SAT Competition[1]

In [KSGS09], the authors studied the application of a parallel hybrid algorithm to deal with the MaxSAT problem. This algorithm combines a complete solver (minisat) and an incomplete one (Walksat). Broadly speaking both solvers are launched in parallel and minisat is used to guide Walksat to promising regions of the search space by means of suggesting values for the selected variables.

7.4.3 Cooperative Algorithms

In [HW93] a set of algorithms running in parallel exchange hints (i.e., partial valid solutions) to solve hard graph coloring instances. To this end, they share a blackboard where they can write a hint with a given probability q and read a hint with a given probability p.

In [SB07] the authors studied a sequential cooperative algorithm to deal with the office-space-allocation problem. In this chapter cooperation takes place when a given algorithm is not able to improve its own best solution, at this point a cooperative mechanism is used to explore suitable partial solutions stored by individual heuristics. This algorithm is also equipped with a diversification strategy to explore different regions of the search space.

Although *Averaging in Previous Near Solutions* [SK93] is not a cooperative algorithm by itself, this method is used to determine the initial configuration for the i^{th} restart in the GSAT algorithm. Broadly speaking, the initial configuration is computed by performing a bitwise average between variables of the best solution found during the previous restart ($restart_{i-1}$) and two restarts before ($restart_{i-2}$). That is, variables with same values in both configurations are re-used, and the extra set of variables are initialized with random values. Since overtime, configurations with a few conflicting clauses tend to become similar, all the variables are randomly initialized after a given number of restarts.

[1] http://www.satcompetition.org/2009/

7.5 Summary

In this chapter, we have studied several knowledge sharing strategies in parallel local search for SAT. We were motivated by the recent developments in parallel DPLL solvers. We decided to restrict the information shared to the best configuration found so far by the algorithms in a portfolio. From that we defined several simple knowledge aggregation strategies along a specific lazy restart policy which creates a new initial configuration when a fix cutoff is meet and when the quality of the shared information has been improved.

Extensive experiments were done on a large number of instances coming from the latest SAT competition. They showed that adding the proposed sharing policies improves the performance of a parallel portfolio, this improvement is exhibited in both number of solved instances and the *Penalized Average Runtime* (PAR). It is also reflected in the best configuration cost of problems which could not be solved within the time limit.

Chapter 8

Conclusions and Perspectives

Along this thesis, we have studied different approaches to efficiently solve combinatorial problems. In particular, we have focussed on Constraint Satisfaction, Constraint Optimization, and SAT problems. In this chapter we conclude this thesis by summarizing our contributions and describing perspectives for future work. This chapter is meant as a complement to the more detailed summaries at the end of previous chapters.

8.1 Overview of the main contributions

In this thesis, we have studied three different perspectives to efficiently solve combinatorial problems. In the first part of the thesis, we have proposed *domFD*, a new variable selection heuristic which exploits the concept of weak dependencies to guide the search at each decision point. Experiments on several problem families showed that *domFD* usually generate search trees smaller than the well-known *dom-wdeg* thus leading this way to better runtimes on the experimented problems.

In the second part of the thesis, we have explored the Algorithm Selection Problem from two different angles. Initially, we investigated the application of a tradition portfolio algorithm to select the best search heuristic for the Protein Structure Prediction Problem by considering features (or descriptors) coming from two different domains. That is, features extracted directly from the biological application, and features from the Constraint Programming encoding of the problem. In both situations, we have observed that the portfolio

of algorithms is able to improve the overall quality of the solutions when compared against the best individual strategy.

Subsequently, we have proposed the Continuous Search paradigm whose main objective is to relax the requirement of a large number of representative instances to be available before hand in order to build a heuristics model. To this end, Continuous Search comes in two modes: the functioning (or exploitation) mode uses the current heuristics model to solve a given problem instance as soon as possible, while the learning (or exploration) mode reuses previous seen instances in order to gradually improve the quality of the heuristics model to become an expert on the user's problem instance distribution. Experimental validation showed that Continuous Search can design efficient mixed strategies after considering a moderate number of problem instances.

In the final part of the thesis, we have explored several knowledge sharing strategies to improve the performance of a parallel portfolio of local search algorithms. The main objective of these strategies is to aggregate the best configuration found so far for each algorithm in the portfolio to carefully build a new configuration to start with. Our results, showed that these simple cooperative strategies along with a specific lazy restart policy help to considerably improve the performance of a parallel portfolio.

8.2 Perspectives

The work presented in this thesis can be extended in many directions. The following are, in the author's opinion, some interesting directions of future work.

- **Variable Selection Heuristics.** A straightforward extension of the *domFD* heuristic would be considering the Multi-level Dynamic Variable Ordering approach [BCS01] which selects the most promising variable by considering neighbors in the constraint graph, in the case of *domFD* one might consider different neighbors in the *Weak Dependencies* graph. Another interesting area of research would be considering [TH10] to build new score functions by normalizing and combining several score metrics (e.g., *domFD*, *mindom*, *dom-wdeg*, etc).

- **Algorithm Selection Problem.** We plan to extend our work in this area mainly

considering the application of Active Learning [BEWB05, DHM07] in order to select the most informative training examples and focus the exploration mode on the most promising heuristics. Another point regards the feature design; better features would be needed to get a higher predictive accuracy, governing the efficiency of the approach. Indeed, we plan to investigate a set of recently proposed descriptors [GJK+10, GKMN10], and also a combination of the low-knowledge feature set proposed in [CB05] with the CP feature set studied in this thesis.

A longer term perspective regards the use of Reinforcement Learning for learning good restart policies; beyond characterizing the best heuristics at a given checkpoint, the goal becomes to find the best sequence of heuristics, to be applied in each checkpoint, in order to solve the instance as fast as possible.

- **Parallel Portfolio Algorithms.** The framework proposed in Chapter 7 intends to be the basics for new parallel local search solvers for SAT. Here much work remains to be done, in particular the use of additional information to exchange, for instance: tabu-list, the age and score of a variable, information on local minima, etc. It should also be important to investigate the best way to integrate this extra knowledge in the course of a given algorithm. As pointed out in Chapter 7, state-of-the-art local search algorithms for SAT perform better when they do not restart. Incorporating extra information without forcing the algorithm to restart is likely to be important.

  In this direction, we plan to equip the local search algorithms used in Chapter 7 with clause learning, as described in [CI96, ALMS09] to exchange learnt clauses, borrowing ideas from portfolios for complete parallel SAT solvers. Another interesting area of research would be designing a parallel portfolio of algorithms which combine complete and incomplete algorithms, and exchange the knowledge achieved for each strategy.

Bibliography

[ADL06] Belarmino Adenso-Díaz and Manuel Laguna. Fine-Tuning of Algorithms Using Fractional Experimental Designs and Local Search. *Operations Research*, 54(1):99–114, 2006.

[AGKS00] Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman. Generating Satisfiable Problem Instances. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 256–261, Austin, Texas, USA, July 2000. AAAI Press / The MIT Press.

[AH09a] Alejandro Arbelaez and Youssef Hamadi. Continuous Search in Constraint Programming: An Initial Investigation. In Karen Petrie and Olivia Smith, editors, *Constraint Programming Doctoral Program*, pages 7–12, Lisbon, Portugal, September 2009.

[AH09b] Alejandro Arbelaez and Youssef Hamadi. Exploiting Weak Dependencies in Tree-Based Search. In *ACM Symposium on Applied Computing (SAC)*, pages 1385–1391, Honolulu, Hawaii, USA, March 2009. ACM.

[AH11a] Alejandro Arbelaez and Youssef Hamadi. Efficient Parallel Local Search for SAT. *Submitted to JAIR*, 2011.

[AH11b] Alejandro Arbelaez and Youssef Hamadi. Improving parallel local search for SAT. In *Learning and Intelligent Optimization, Fifth International Conference, LION 2011 (to appear)*, 2011.

[AHS09] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Online Heuristic Selection in Constraint Programming. In *International Symposium on Combinatorial Search*, Lake Arrowhead, USA, July 2009.

[AHS10a] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Building Portfolios for the Protein Structure Prediction Problem. In *Edinburgh Workshop on Constraint Based Methods for Bioinformatics (WCB)*, Edinburgh, UK, July 2010.

[AHS10b] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous Search in Constraint Programming. In Eric Gregoire, editor, *22th International Conference on Tools With Artificial Intelligence (ICTAI)*, volume 1, pages 53–60, Arras, France, October 2010. IEEE.

[AHS11] Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous Search in Constraint Programming. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*. Springer-Verlag, 2011.

[AKJ04] Rehan Akbani, Stephen Kwek, and Nathalie Japkowicz. Applying Support Vector Machines to Imbalanced Datasets. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *15th European Conference on Machine Learning*, volume 3201 of *Lecture Notes in Computer Science*, pages 39–50, Pisa, Italy, Sept 2004. Springer.

[ALMS09] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. Learning in local search. In *21st IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 417–424, Newark, New Jersey, USA, November 2009. IEEE Computer Society.

[ASBG07] Ali Al-Shahib, Rainer Breitling, and David R. Gilbert. Predicting Protein Function by Machine Learning on Amino Acid Sequences – A Critical Evaluation. *BMC Genomics*, 78(2), March 2007.

[AST09] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In Ian P.

Gent, editor, *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157, Lisbon, Portugal, Sept 2009. Springer.

[AV06] Francisco Azevedo and Hau Nguyen Van. Extra Constraints for the Social Golfers Problem. In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2006.

[BB05] Roberto Battiti and Mauro Brunato. Reactive Search: Machine Learning for Memory-Based Heuristics. Technical report, Teofilo F. Gonzalez (Ed.), Approximation Algorithms and Metaheuristics, Taylor & Francis Books (CRC Press), 2005.

[BCDP07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global Constraint Catalogue: Past, Present and Future. *Constraints*, 12(1):21–62, 2007.

[BCS01] Christian Bessière, Assef Chmeiss, and Lakhdar Sais. Neighborhood-Based Variable Ordering Heuristics for the Constraint Satisfaction Problem. In Toby Walsh, editor, *7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 565–569, Paphos, Cyprus, November 2001. Springer.

[BE93] Timothy L. Bailey and Charles Elkan. Estimating the Accuracy of Learned Concepts. In *IJCAI*, pages 895–901, 1993.

[BEWB05] Antoine Bordes, Seyda Ertekin, Jason Weston, and Léon Bottou. Fast Kernel Classifiers with Online and Active Learning. *Journal of Machine Learning Research*, 6:1579–1619, Sept 2005.

[BF04] J. Christopher Beck and Eugene C. Freuder. Simple Rules for Low-Knowledge Algorithm Selection. In Jean-Charles Régin and Michel Rueher, editors, *First International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*

(CPAIOR), volume 3011 of *Lecture Notes in Computer Science*, pages 50–64, Nice, France, April 2004. Springer.

[BHL05] Frederic Boussemart, Fred Hemery, and Christophe Lecoutre. Description and Representation of the Problems Selected for the First International Constraint Satisfaction Solver Competition. In *CPAI'05 workshop held with CP'05*, pages 7–26, 2005.

[BHLS04] Frederic Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, pages 146–150, Valencia, Spain, Aug 2004. IOS Press.

[BHZ06] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Comput. Surv.*, 38(4), 2006.

[Bie10] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report 10/1, FMV Reports Series, August 2010.

[Bir04] Mauro Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, Bruxelles, Belgium, 2004.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[BM08] Marco Benedetti and Hratch Mangassarian. QBF-Based Formal Verification: Experience and Perspectives. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:133–191, 2008.

[Bos01] Robert Bosch. Painting by numbers. *Optima*, (65):16–17, May 2001.

[Bre79] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22:251–256, 1979.

[Bre96] Leo Breiman. Heuristics of Instability and Stabilization in Model Selection. *Annals of Statistics*, 24(6):2350–2383, 1996.

[BSPV02] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, New York, USA, July 2002. Morgan Kaufmann.

[BT94] Roberto Battiti and Giampietro Tecchiolli. The Reactive Tabu Search. *INFORMS Journal on Computing*, 6(2):126–140, 1994.

[BTW96] James E. Borrett, Edward P. K. Tsang, and Natasha R. Walsh. Adaptive Constraint Satisfaction: The Quickest First Principle. In Wolfgang Wahlster, editor, *12th European Conference on Artificial Intelligence*, pages 160–164, Budapest, Hungary, August 1996. John Wiley and Sons, Chichester.

[BW01] Rolf Backofen and Sebastian Will. Fast, Constraint-based Threading of HP-Sequences to Hydrophobic Cores. In Toby Walsh, editor, *7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 494–508, Paphos, Cyprus, November 2001. Springer.

[CB04] Tom Carchrae and J. Christopher Beck. Low-Knowledge Algorithm Control. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 49–54, San Jose, California, USA, July 2004. AAAI Press / The MIT Press.

[CB05] Tom Carchrae and J. Christopher Beck. Applying Machine Learning to Low-Knowledge Control of Optimization Algorithms. *Computational Intelligence*, 21(4):372–387, 2005.

[CB06] Jianlin Cheng and Pierre Baldi. A Machine Learning Information Retrieval Approach to Protein Fold Recognition. *Bioinformatics*, 22(12):1456–1463, 2006.

[CB08] Marco Correira and Pedro Barahona. On the Efficiency of Impact Based Heuristics. In Peter J. Stuckey, editor, *14th International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 608–612, Sydney, Australia, September 2008. Springer.

[CDD08] Raffaele Cipriano, Alessandro Dal Palù, and Agostino Dovier. A Hybrid Approach Mixing Local Search and Constraint Programming Applied to the Protein Structure Prediction Problem. In *Workshop on Constraint Based Methods for Bioinformatics (WCB)*, Paris, France, 2008.

[CI96] Byungki Cha and Kazuo Iwama. Adding New Clauses for Faster Local Search. In *AAAI/IAAI*, volume 1, pages 332–337, 1996.

[CKL06] Gérard Cornuéjols, Miroslav Karamanov, and Yanjun Li. Early Estimates of the Size of Branch-and-Bound Trees. *INFORMS Journal on Computing*, 18(1):86–96, 2006.

[CL01] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines, 2001. Software from `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CW06] Wahid Chrabakh and Rich Wolski. GridSAT: A System for Solving Satisfiability Problems Using a Computational Grid. *Parallel Comput.*, 32(9):660–687, 2006.

[DD17] Chris H. Q. Ding and Inna Dubchak. Multi-Class Protein Fold Recognition Using Support Vector Machines and Neural Networks. *Bioinformatics*, 4(2001):341–358, 17.

[DDF03] Alessandro Dal Palù, Agostino Dovier, and Federico Fogolari. Protein Folding in CLP(FD) with Empirical Contact Energies. In Krzysztof R. Apt, François Fages, Francesca Rossi, Péter Szeredi, and József Váncza, editors, *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP)*, volume 3010 of *Lecture Notes in Computer Science*, pages 250–265, Budapest, Hungary, June 2003. Springer.

[DDP07] Alessandro Dal Palù, Agostino Dovier, and Enrico Pontelli. A Constraint Solver for Discrete Lattices, its Parallelization, and Application to Protein Structure Prediction. *Softw. Pract. Exper.*, 37(13):1405–1449, 2007.

[DFSS08] Luis Da Costa, Álvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive Operator Selection with Dynamic Multi-Armed Bandits. In Conor Ryan and Maarten Keijzer, editors, *Genetic and Evolutionary Computation Conference (GECCO)*, pages 913–920, Atlanta, GA, USA, July 2008. ACM.

[DGS07] Bistra N. Dilkina, Carla P. Gomes, and Ashish Sabharwal. Tradeoffs in the Complexity of Backdoor Detection. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2007.

[DHM07] Sanjoy Dasgupta, Daniel Hsu, and Claire Monteleoni. A General Agnostic Active Learning Algorithm. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *Proceedings of the Twenty-First Annual*

Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, Dec 2007. MIT Press.

[DO08] David Devlin and Barry O'Sullivan. Satisfiability as a Classification Problem. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[DpAD10] Abdollah Dehzangi, Somnuk phon Amnuaisuk, and Omid Dehzangi. Using Random Forest for Protein Fold Prediction Problem: An Empirical Study. *Journal of Information Science and Engineering*, 26(6):1941–1956, 2010.

[DT00] K. A. Dowsland and J. M. Thompson. Solving a Nurse Scheduling Problem with Knapsacks, Networks and Tabu Search. *Journal of the Operational Research Society*, 51:825–833, 2000.

[EFW+02] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The Adaptive Constraint Engine. In Pascal Van Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 525–542, NY, USA, Sept 2002. Springer.

[EGS02] Bertrand Mazure Eric Gregoire, Richard Ostrowski and Lakhdar Sais. Recovering and Exploiting Structural Knowledge from CNF Formulas. In Pascal Van Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470, pages 185–199, Ithaca, NY, USA, September 2002. Springer.

[FDSS10] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michele Sèbag. Analyzing Bandit-based Adaptive Operator Selection Mechanisms. *Annals of Mathematics and Artificial Intelligence – Special Issue on Learning and Intelligent Optimization*, 2010.

[FHH+05] Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. WEKA - A Machine Learning Workbench for Data Mining. In Oded Maimon and Lior Rokach, editors, *The Data Mining and Knowledge Discovery Handbook*, pages 1305–1314. Springer, 2005. Available from `www.cs.waikato.ac.nz/ml/weka`.

[FRSS10] Álvaro Fialho, Raymond Ros, Marc Schoenauer, and Michèle Sebag. Comparison-Based Adaptive Strategy Selection with Bandits in Differential Evolution. In Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Günter Rudolph, editors, *11th International Conference on Parallel Problem Solving from Nature (PPSN)*, volume 6238 of *Lecture Notes in Computer Science*, pages 194–203, Kraków, Poland, September 2010. Springer.

[GE03] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 4(2003):1157–1182, 2003.

[Gec06] Gecode Team. Gecode: Generic Constraint Development Environment, 2006. Available from `http://www.gecode.org`.

[Gen03] Michel Gendreau. An Introduction to Tabu Search. In *Handbook of Metaheuristics*, volume 57, pages 37–54. Kluwer Academic Publishers, 2003.

[GHBF05] Cormac Gebruers, Brahim Hnich, Derek G. Bridge, and Eugene C. Freuder. Using CBR to Select Solution Strategies in Constraint Programming. In Héctor Muñoz-Avila and Francesco Ricci, editors, *6th International Conference on Case-Based Reasoning, Research and Development*, volume 3620 of *Lecture Notes in Computer Science*, pages 222–236, Chicago, IL, USA, August 2005. Springer.

[GJK+10] Ian P. Gent, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Learning When to Use Lazy Learning in Constraint Solving. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *19th European Conference on Artificial Intelligence*,

volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 873–878, Lisbon, Portugal, August 2010. IOS Press.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A Fast Scalable Constraint Solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *17th European Conference on Artificial Intelligence (ECAI)*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102, Riva del Garda, Italy, August 2006. IOS Press.

[GKMN10] Ian Gent, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Machine Learning for Constraint Solver Design – A Case Study for the alldifferent Constraint. In *3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS)*, 2010.

[GM04] Alessio Guerri and Michela Milano. Learning Techniques for Automatic Algorithm Portfolio Selection. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 475–479, Valencia, Spain, August 2004. IOS Press.

[GMR07] Ian P. Gent, Ian Miguel, and Andrea Rendl. Tailoring Solver-Independent Constraint Models: A Case Study with Essence' and Minion. In Ian Miguel and Wheeler Ruml, editors, *7th International Symposium on Abstraction, Reformulation, and Approximation*, volume 4612 of *Lecture Notes in Computer Science*, pages 184–199, Whistler, Canada, July 2007. Springer.

[GMS03] Ian P. Gent, Iain McDonald, and Barbara Smith. Conditional Symmetry in the All-Interval Series Problem. In Barbara Smith, Ian P. Gent, and Warwick Harvey, editors, *3rd International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 55–65, Kinsale, County Cork, Ireland, September 2003.

[GS01] Carla P. Gomes and Bart Selman. Algorithm Portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[GS07] Sylvain Gelly and David Silver. Combining Online and Offline Knowledge in UCT. In Zoubin Ghahramani, editor, *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280, Corvalis, Oregon, USA, June 2007. ACM.

[GSK98] Carla Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[GW99] Ian P. Gent and Toby Walsh. CSPLIB: A Benchmark Library for Constraints. In Joxan Jaffar, editor, *5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 480–481, Alexandria, Virginia, USA, October 1999. Springer. Available from `www.csplib.org`.

[HE79] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. In *IJCAI*, pages 356–364, San Francisco, CA, USA, 1979.

[Heb08] Emmanuel Hebrard. Mistral, A Constraint Satisfaction Library. In Marc van Dongen, Christophe Lecoutre, and Olivier Roussel, editors, *2nd International CSP Solver Competition*, 2008.

[Hen99] Martin Henz. Constraint-based Round Robin Tournament Planning. In Danny De Schreye, editor, *International Conference on Logic Programming*, pages 545–557, Cambridge, Massachusetts, 1999. The MIT Press.

[HH05] Frank Hutter and Youssef Hamadi. Parameter Adjustment Based on Performance Prediction: Towards an Instance-Aware Problem Solver. Technical Report MSR-TR-2005-125, Microsoft Research, Microsoft Corporation One Microsoft Way Redmond, WA 98052, December 2005.

[HHHLB06] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. In Frédéric Benhamou, editor, *12th International Conference*

on Principles and Practice of Constraint Programming, volume 4204 of *Lecture Notes in Computer Science*, pages 213–228, Nantes, France, Sept 2006. Springer.

[HHLB10] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Tradeoffs in the Empirical Evaluation of Competing Algorithm Designs. *Annals of Mathematics and Artificial Intelligenc (AMAI), Special Issue on Learning and Intelligent Optimization*, 2010.

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[HJS09] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 6:245–262, 2009.

[HKS01] Zineb Habbas, Michaël Krajecki, and Daniel Singer. The Langford's Problem:A Challenge for Parallel Resolution of CSP. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *4th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, volume 2328 of *Lecture Notes in Computer Science*, pages 789–796, Naleczow, Poland, September 2001. Springer.

[HMS11] Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. What Is Autonomous Search? In Panos M. Pardalos, Pascal van Hentenryck, and Michela Milano, editors, *Hybrid Optimization*, volume 45 of *Optimization and Its Applications*, pages 357–391. Springer New York, 2011.

[Hoo99] Holger H. Hoos. On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. In *AAAI/IAAI*, pages 661–666, 1999.

[Hoo02] Holger H. Hoos. An Adaptive Noise Mechanism for WalkSAT. In *AAAI/IAAI*, pages 655–660, 2002.

[HRG+01] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Bart Selman, and David Maxwell Chickering. A Bayesian Approach to Tackling Hard Computational Problems. In Jack S. Breese and Daphne Koller, editors, *17th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 235–244, Washington, USA, August 2001. Morgan Kaufmann.

[HTH02] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. In Pascal Van Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248, Ithaca, NY, USA, September 2002. Springer.

[Hua07] Jinbo Huang. The Effect of Restarts on the Efficiency of Clause Learning. In Manuela M. Veloso, editor, *20th International Joint Conference on Artificial Intelligence*, pages 2318–2323, Hyderabad, India, January 2007.

[Hut09] Frank Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, Department of Computer Science, Vancouver, Canada, October 2009.

[HW93] Tad Hogg and Colin P. Williams. Solving the Really Hard Problems with Cooperative Search. In *AAAI*, pages 231–236, 1993.

[HW08] Shai Haim and Toby Walsh. Online Estimation of SAT Solving Runtime. In Hans Kleine Büning and Xishun Zhao, editors, *11th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 4996 of *Lecture Notes in Computer Science*, pages 133–138, Guangzhou, China, May 2008. Springer.

[HW09a] Shai Haim and Toby Walsh. Online Search Cost Estimation for SAT Solvers. *CoRR*, abs/0907.5033, 2009.

[HW09b] Shain Haim and Toby Walsh. Restart Strategy Selection Using Machine Learning Techniques. In Oliver Kullmann, editor, *12th International Conference on Theory and Applications of Satisfiability Testing*, volume 5584

of *Lecture Notes in Computer Science*, pages 312–325, Swansea, UK, June 2009. Springer.

[KHR+02] Henry A. Kautz, Eric Horvitz, Yongshao Ruan, Carla P. Gomes, and Bart Selman. Dynamic Restart Policies. In *AAAI/IAAI*, pages 674–681, 2002.

[Khu10] Ashiqur R. KhudaBukhsh. SATenstein: Automatically Building Local Search SAT Solvers from Components. Master's thesis, University of British Columbia, Vancouver, British Columbia, Canada, October 2010.

[KMN10] Lars Kotthoff, Ian Miguel, , and Peter Nightingale. Ensemble Classification for Constraint Solver Configuration. In David Cohen, editor, *16th International Conference on Principles and Practices of Constraint Programming (CP'10)*, volume 6308 of *Lecture Notes in Computer Science*, St Andrews, UK, 2010. Springer.

[Knu75] Donald E Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[Koh95] Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, pages 1137–1145, 1995.

[KSGS09] Lukas Kroc, Ashish Sabharwal, Carla P. Gomes, and Bart Selman. Integrating Systematic and Local Search Paradigms: A New Strategy for MaxSAT. In Craig Boutilier, editor, *IJCAI*, pages 544–551, Pasadena, California, July 2009.

[KSTW06] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating Search Tree Size. In *AAAI*, Boston, Massachusetts, USA, July 2006. AAAI Press.

[KXHLB09] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. SATenstein: Automatically Building Local Search SAT Solvers from Components. In Craig Boutilier, editor, *IJCAI*, pages 517–524, Pasadena, California, USA, July 2009.

[Lag08] Mikael Z. Lagerkvist. *Techniques for Efficient Constraint Propagation*. Licentiate thesis, KTH - Royal Institute of Technology, Stockholm, Sweden 2008.

[LB08] Hugo Larochelle and Yoshua Bengio. Classification Using Discriminative Restricted Boltzmann Machines. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *Proceedings of the Twenty-Fifth International Conference on Machine Learning*, volume 307 of *ACM International Conference Proceeding Series*, pages 536–543, Helsinki, Finland, June 2008. ACM.

[LBNS02] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the Empirical Hardness of Optimization Problems: The Case of Combinatorial Auctions. In Pascal Van Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 556–572, Ithaca, NY, USA, September 2002. Springer.

[LBNS06] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical Hardness Models for Combinatorial Auctions. In Peter Cramton, Yoav Shoham, and Richard Steinberg, editors, *Combinatorial Auctions*, chapter 19, pages 479–504. MIT Press, 2006.

[LBNS09] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions. *J. ACM*, 56(4), 2009.

[LED] LEDA – Library of Efficient Data Types and Algorithms. Available from `http://www.algorithmic-solutions.com/`.

[LH05] Chu Min Li and Wen Qi Huang. Diversification and Determinism in Local Search for Satisfiability. In Fahiem Bacchus and Toby Walsh, editors, *8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pages 158–172, St. Andrews, UK, June 2005. Springer.

[LL98] Lionel Lobjois and Michel Lemaître. Branch and Bound Algorithm Selection by Performance Prediction. In *AAAI/IAAI*, pages 353–358, 1998.

[LL01] Michail G. Lagoudakis and Michael L. Littman. Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.

[LS07] Mikael Z. Lagerkvist and Christian Schulte. Advisors for Incremental Propagation. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 409–422, Providence, RI, USA, September 2007. Springer.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Optimal Speedup of Las Vegas Algorithms*, 47(4):173–180, 1993.

[LWZ07] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In João Marques-Silva and Karem A. Sakallah, editors, *10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *Lecture Notes in Computer Science*, pages 121–133, Lisbon, Portugal, May 2007. Springer.

[MCC06] Eric Monfroy, Carlos Castro, and Broderick Crawford. Adaptive Enumeration Strategies and Metabacktracks for Constraint Solving. In Tatyana M. Yakhno and Erich J. Neuhold, editors, *4th International Conference on Advances in Information Systems (ADVIS)*, volume 4243 of *Lecture Notes in Computer Science*, pages 354–363, Izmir, Turkey, October 2006. Springer.

[Min93] Steven Minton. An Analytic Learning System for Specializing Heuristics. In *13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 922–929. Morgan Kaufmann, 1993.

[Min96] Steven Minton. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, 1(1/2):7–43, 1996.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artif. Intell.*, 58(1-3):161–205, 1992.

[MM97] Oded Maron and Andrew W. Moore. The Racing Algorithm: Model Selection for Lazy Learners. *Artif. Intell. Rev.*, 11:1–5, 1997.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[MSG97] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Tabu Search for SAT. In *AAAI/IAAI*, pages 281–285, 1997.

[MSG98] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Boosting Complete Techniques Thanks to Local Search methods. *Ann. Math. Artif. Intell*, 22(3-4):319–331, 1998.

[MSK97] David A. McAllester, Bart Selman, and Henry A. Kautz. Evidence for Invariants in Local Search. In *AAAI/IAAI*, pages 321–326, 1997.

[MSR+09] Martin Mann, Cameron Smith, Mohamad Rabbath, Marlien Edwards, Sebastian Will, and Rolf Backofen. CPSP-web-tools: A Server for 3D Lattice Protein Studies. *Bioinformatics*, 25(5):676–677, 2009.

[NLBH+04] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In Mark Wallace, editor, *10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452, Toronto, Canada, September 2004. Springer.

[NLLP10] Morten Nielsen, Claus Lundegaard, Ole Lund, and Thomas Nordahl Petersen. CPHmodels-3.0–Remote Homology Modeling Using Structure-Guided Sequence Profiles. *Nucleic Acids Res*, 38, 2010.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543, Providence, RI, September 2007. Springer.

[OHH+08] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, August 2008.

[O'S10] Barry O'Sullivan. Automated Modelling and Solving in Constraint Programming. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A Survey of Recent Advances in SAT-based Formal Verification. *STTT*, 7(2):156–173, 2005.

[PC03] Nikhil R. Pal and Debrup Chakraborty. Some New Features for Protein Fold Prediction. In *ICANN*, pages 1176–1183, Istanbul, Turkey, June 2003. Springer.

[PDP05] Alessandro Dal Palù, Agostino Dovier, and Enrico Pontelli. A new constraint solver for 3d lattices and its application to the protein folding problem. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 48–63, 2005.

[PE06] Smiljana Petrovic and Susan L. Epstein. Full Restart Speeds Learning. In Geoff Sutcliffe and Randy Goebel, editors, *Proceedings of the Nineteenth*

International Florida Artificial Intelligence Research Society Conference (FLAIRS), pages 104–103, Melbourne Beach, Florida, USA, 2006. AAAI Press.

[PE08] Smiljana Petrovic and Susan L. Epstein. Random Subsets Support Learning a Mixture of Heuristics. *International Journal on Artificial Intelligence Tools*, 17(3):501–520, 2008.

[PG07] Duc Nghia Pham and Charles Gretton. gNovelty+. In *Solver description, SAT competition 2007*, 2007.

[PG09] Duc Nghia Pham and Charles Gretton. gNovelty+ (v.2). In *Solver description, SAT competition 2009*, 2009.

[Phi03] Ansgar Philippsen. DINO: Visualizing Structural Biology, 2003. Available from `http://www.dino3d.org`.

[Pre01] Steven Prestwich. Balance Incomplete Block Design as Satisfiability. In *12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.

[PT07] Luca Pulina and Armando Tacchella. A Multi-engine Solver for Quantified Boolean Formulas. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 574–589, Providence, RI, USA, September 2007. Springer.

[PT09] Luca Pulina and Armando Tacchella. A Self-Adaptive Multi-Engine Solver for Quantified Boolean Formulas. *Constraints*, 14(1):80–116, 2009.

[PTGS08] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining Adaptive and Dynamic Local Search for Satisfiability. *JSAT*, 4(2-4):149–172, 2008.

[Pug04] Jean-Francois Puget. Constraint Programming Next Challenge: Simplicity of Use. In Mark Wallace, editor, *10th International Conference on Principles*

and Practice of Constraint Programming, volume 3258 of *Lecture Notes in Computer Science*, Toronto, Canada, September 2004. Springer.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):86–106, 1986.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[RAD10] Emmanuel Rachelson, Ala Ben Abbes, and Sèbastien Diemer. Combining Mixed Integer Programming and Supervised Learning for Fast Re-Planning. In Eric Gregoire, editor, *22th International Conference on Tools With Artificial Intelligence (ICTAI)*, volume 2, pages 63–70, Arras, France, October 2010. IEEE.

[R.B92] R.Bisiani. Beam Search. In Stuart C Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1467–1468. John Wiley and Sons, 1992.

[RBB05] Andrea Roli, Maria J Blesa, and Christian Blum. Random Walk and Parallelism in Local Search. In *Metaheuristic International Conference (MIC'05)*, Vienna, Austria, 2005.

[RBM+05] Irina Rish, Mark Brodie, Sheng Ma, Natalia Odintsova, Alina Beygelzimer, Genady Grabarnik, and Karina Hernandez. Adaptive Diagnosis in Distributed Systems. *IEEE Trans. on Neural Networks*, 16:1088–1109, September 2005.

[Ref04] Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In Mark Wallace, editor, *10th International Conference on Principles and Practice of Constraint Programming*, volume 2004 of *Lecture Notes in Computer Science*, pages 557–571, Toronto, Canada, Sept 2004. Springer.

[RHK02] Yongshao Ruan, Eric Horvitz, and Henry A. Kautz. Restart Policies with Dependence Among Runs: A Dynamic Programming Approach. In Pascal Van Hentenryck, editor, *8th International Conference on Principles and Practice*

of Constraint Programming, volume 2470 of *Lecture Notes in Computer Science*, pages 573–586, Ithaca, NY, USA, September 2002. Springer.

[Ric76] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[RK04] Ryan Rifkin and Aldebaro Klautau. In Defense of One-Vs-All Classification. *J. Mach. Learn. Res.*, 5(41):101–141, December 2004.

[RK07] Daniel M. Roy and Leslie Pack Kaelbling. Efficient Bayesian Task-Level Transfer Learning. In Manuela M. Veloso, editor, *IJCAI*, pages 2599–2604, Hyderabad, India, January 2007.

[RL09] Olivier Roussel and Christophe Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[Rol02] Andrea Roli. Criticality and Parallelism in Structured SAT Instances. In Pascal Van Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 714–719, Ithaca, NY, USA, September 2002. Springer.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[SB98] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998.

[SB07] Dario Landa Silva and Edmund K. Burke. Asynchronous Cooperative Local Search for the Office-Space-Allocation Problem. *INFORMS Journal on Computing*, 19(4):575–587, 2007.

[SC06] Christian Schulte and Mats Carlsson. Finite Domain Constraint Programming Systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors,

Handbook of Constraint Programming, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[Sch02] Robert E. Schapire. Advances in Boosting. In Adnan Darwiche and Nir Friedman, editors, *18th Conference in Uncertainty in Artificial Intelligence (UAI '02)*, pages 446–452, Alberta, Canada, August 2002. Morgan Kaufmann.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *European Conference on Artificial Intelligence (ECAI)*, pages 125–129, 1994.

[SGS07] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining Multiple Heuristics Online. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1197–1203, Vancouver, British Columbia, Canada, July 2007. AAAI Press.

[SGS08] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining Multiple Constraint Solvers: Results on the CPAI'06 Competition Data. In Marc van Dongen, Christophe Lecoutre, and Olivier Roussel, editors, *2nd International CSP Solver Competition*, pages 11–18, 2008.

[SHG09] David H. Stern, Ralf Herbrich, and Thore Graepel. Matchbox: Large Scale Online Bayesian Recommendations. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *18th International Conference on World Wide Web (WWW'09)*, pages 111–120, Madrid, Spain, April 2009. ACM.

[Sim05] Helmut Simonis. Sudoku as a Constraint Problem. In Brahim Hnich, Patrick Prosser, and Barbara Smith, editors, *Fourth International Workshop 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, Barcelona, Spain, October 2005.

[SK93] Bart Selman and Henry A. Kautz. Domain-Independent Extensions to GSAT:

Solving Large Structured Satisfiability Problems. In *IJCAI*, pages 290–295, 1993.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *AAAI*, pages 337–343, 1994.

[SLB09] Tobias Schubert, Matthew Lewis, and Bernd Becker. PaMiraXT: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 6:203–222, 2009.

[SLM92] Bart Selman, Hector J. Levesque, and David G. Mitchell. A New Method for Solving Hard Satisfiability Problems. In 440-446, editor, *AAAI*, 1992.

[SM07] Horst Samulowitz and Roland Memisevic. Learning to Solve QBF. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, Vancouver, British Columbia, July 2007. AAAI Press.

[SM08] Kate Smith-Miles. Cross-Disciplinary Perspectives on Meta-Learning for Algorithm Selection. *ACM Comput. Surv.*, 41(1), 2008.

[SMJGT09] Kate Smith-Miles, Ross J. W. James, John W. Giffin, and Yiqing Tu. Understanding Relationships between Scheduling Problem Structure and Heuristic Performance. In Thomas Stützle, editor, *Third International Conference on Learning and Intelligent Optimization (LION 3)*, volume 5851 of *Lecture Notes in Computer Science*, pages 89–103, Trento, Italy, 2009. Springer.

[SS08] Matthew J. Streeter and Stephen F. Smith. New Techniques for Algorithm Portfolio Design. In David A. McAllester and Petri Myllymäki, editors, *24th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 519–527, Helsinki, Finland, 2008. AUAI Press.

[SSH+10] David H. Stern, Horst Samulowitz, Ralf Herbrich, Thore Graepel, Luca Pulina, and Armando Tacchella. Collaborative Expert Portfolio Management. In Maria Fox and David Poole, editors, *AAAI*, pages 179–184, Atlanta, Georgia, USA, July 2010. AAAI Press.

[SSK06] Luai Al Shalabi, Zyan Shaaban, and Basel Kasasbeh. Data Mining: A Preprocessing Engine. In *Journal of Computer Science*, volume 2, pages 735–739, 2006.

[SSW99] Barbara Smith, Kostas Stergiou, and Toby Walsh. Modelling the Golomb Ruler Problem. In *Workshop on non-binary constraints*. 1999.

[ST08] Christian Schulte and Guido Tack. Perfect Derived Propagators. In Peter J. Stuckey, editor, *14th International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 571–575, Sydney, Australia, September 2008. Springer.

[Tac09] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.

[TH04] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. In Holger H. Hoos and David G. Mitchell, editors, *7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320, Vancouver, BC, Canada, 2004. Springer.

[TH10] Dave A. D. Tompkins and Holger H. Hoos. Dynamic Scoring Functions with Variable Expressions: New SLS Methods for Solving SAT. In Ofer Strichman and Stefan Szeider, editors, *13th International Conference on Theory and Applications of Satisfiability Testing*, volume 6175 of *Lecture Notes in Computer Science*, pages 278–292, Edinburgh, UK, July 2010. Springer.

[TPBaFJ04] John Thornton, Duc Nghia Pham, Stuart Bain, and alnir Ferreira Jr. Additive versus Multiplicative Clause Weighting for SAT. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 191–196, San Jose, California, USA, July 2004. AAAI Press / The MIT Press.

[Van06] Peter Van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*

(Foundations of Artificial Intelligence), chapter 4, pages 85–134. Elsevier Science Inc., New York, NY, USA, 2006.

[Vap95] Vladimir Vapnik. *The Nature of Statistical Learning*. Springer Verlag, New York, NY, USA, 1995.

[Wal99] Toby Walsh. Search in a Small World. In Thomas Dean, editor, *IJCAI*, volume 2, pages 1172–1177, Stockholm, Sweden, 1999. Morgan Kaufmann.

[WB08] Huayue Wu and Peter Van Beek. Portfolios With Deadlines For Backtracking Search. *International Journal on Artificial Intelligence Tools*, 17(5):835–856, 2008.

[Wei] Eric W. Weisstein. Magic Square. From MathWorld–A Wolfram Web Resource `http://mathworld.wolfram.com/MagicSquare.html`.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2 edition, June 2005.

[WGS03] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors To Typical Case Complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1173–1178, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[WM97] David Wolpert and William G. Macready. No Fre Lunch Theorems for Optimization. *IEEE Trans. Evolutionary Computation*, 1(1):67–82, 1997.

[XHHLB07] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. The Design and Analysis of an Algorithm Portfolio for SAT. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 712–727, Providence, RI, USA, Sept 2007. Springer.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[XHLB07] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hierarchical Hardness Models for SAT. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 696–711, Providence, RI, USA, September 2007. Springer.

[XHLB10] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. In Maria Fox and David Poole, editors, *AAAI*, pages 210–216, Atlanta, Georgia, USA, July 2010. AAAI Press.

[XSS09] Yuheua Xu, David Stern, and Horst Samulowitz. Learning Adaptation to Solve Constraint Satisfaction Problems. In *Learning and Intelligent Optimization (LION)*, 2009.

[ZE08] Zhijun Zhang and Susan L. Epstein. Learned Value-Ordering Heuristics for Constraint Satisfaction. In *The First International Symposium on Search Techniques in Artificial Intelligence and Robotics*, Chicago, Illinois, USA, July 2008.

[ZHZ02] Wenhui Zhang, Zhuo Huang, and Jian Zhang. Parallel Execution of Stochastic Search Procedures on Reduced SAT Instances. In Mitsuru Ishizuka and Abdul Sattar, editors, *The Pacific Rim International Conferences on Artificial Intelligence (PRICAI)*, volume 2417 of *Lecture Notes in Computer Science*, pages 108–117, Tokyo, Japan, August 2002. Springer.