LEARNING AND PARALLELIZATION BOOST CONSTRAINT SEARCH

by

XI YUN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirement for the degree of Doctor of Philosophy. The City University of New York

2013

© 2013

XI YUN

All Rights Reserved

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy

__________ __________________________

Date | Dr. Susan L. Epstein
Chair of Examining Committee

__________ __________________________

Date | Dr. Theodore Brown
Executive Officer

Dr. Robert M. Haralick

Dr. Andrew Rosenberg

Dr. Phileppe Jégou

Supervision Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

LEARNING AND PARALLELIZATION BOOST CONSTRAINT SEARCH

by

Xi Yun

Advisor: Professor Susan L. Epstein

Constraint satisfaction problems are a powerful way to abstract and represent academic and real-world problems from both artificial intelligence and operations research. A constraint satisfaction problem is typically addressed by a sequential constraint solver running on a single processor. Rather than construct a new, parallel solver, this work develops convenient parallelization methods for pre-existing solvers, and thereby benefits from both state-of-the-art research and increasingly available computational resources.

This work proposes and evaluates several approaches that exploit scheduling and partitioning. It formulates parallel algorithm portfolio construction as an integer-programming problem, and solves the problem with algorithms that combine case-based reasoning with either a greedy algorithm or a complete integer-programming solver. This work also develops a hybrid adaptive paradigm that learns critical information to support search and workload splitting while it solves the problem. Extensive experiments show that these approaches improve the performance of the underlying sequential solvers. The hybrid paradigm solves many difficult problems left open after recent solver

competitions. Although the empirical results are mainly on constraint satisfaction problems, this work also generalizes the reasoning component of the parallel scheduler to a new, case-based paradigm, and successfully applies it to protein-ligand docking.

To my parents and my wife

Acknowledgements

The completion of this dissertation would have been impossible without the help and support of many people. First of all, I would like to express my sincere gratitude to my mentor, colleague, and friend Dr. Susan L. Epstein for her continuous guidance, inspiration, and help. She exemplified for me a model scientific researcher by her hard work, curiosity, devotion, and everlasting enthusiasm for research. She left tremendous space for me to explore different ideas, but she was always available when I needed discussion, intuition, and advice. I have been very fortunate to have her as the advisor for my doctoral study.

I would like to thank my other dissertation committee members, Dr. Robert M. Haralick, Dr. Philippe Jégou, and Dr. Andrew Rosenberg, for their inspiration and for their constructive comments and suggestions. Thanks to Dr. Eugene Freuder and to Dr. Richard Wallace from the Cork Constraint Computation Centre for contributing valuable insights to the development of this research. Appreciation also goes to Dr. Ted Brown and to Lina Garcia for their continuous help during my study at the Graduate Center of The City University of New York (CUNY).

I would like to thank my fellow researchers, Dr. Xingjian Li, Dr. Tiziana Ligorio, Eric Osisek, Ben Hixon, and Eric Schneider, from the Problem Solving and Machine Learning Laboratory at CUNY. We had many thoughtful discussions, and I have learned much from them. I am also grateful to the staff of the High-Performance Computing Center of CUNY, who helped me many times in my battles with the clusters at the Center.

Finally, and above all, I would like to thank my dear family members. I thank my father Yaohui Yun and my mother Guifeng Zhang for their unwavering love, support, and encouragement through all my life. I thank my

wife Shujuan Huang for her firm love, trust, support, and encouragement even in the most difficult days. Their love, expectation, and encouragement are the fuel of my work and my life.

Contents

List of Figures xi

List of Tables xiii

List of algorithms xiv

1 Introduction 1

1.1 Contributions and principal results . . . 3
1.2 Definitions . . . 8
1.3 Solvers . . . 10
1.4 Additional techniques . . . 12

2 Parallel algorithm portfolios 15

2.1 Definitions . . . 16
2.2 Learning an effective algorithm portfolio . . . 20
2.2.1 Optimal portfolio construction . . . 21
2.2.2 Properties of optimal schedules . . . 22
2.3 Greedy constructor methods . . . 24
2.3.1 WG . . . 24
2.3.2 RSR-WG . . . 26
2.3.3 Bounds for RSR-WG . . . 29
2.4 Complete method . . . 31
2.5 Empirical results . . . 34
2.5.1 RSR-WG, RPCPHydra, RPWG, and an oracle . . . 34
2.5.2 Comparison of RSR-WG to a complete constructor . . . 40
2.6 MAMC, a case-based portfolio paradigm . . . 44
2.6.1 Protein-ligand docking . . . 45

2.6.2 Cases, similarity, and combination . . . 47
2.6.3 Experimental design and results . . . 49
2.6.4 Discussion of PLD results . . . 51

3 SPREAD: a Hybrid Paradigm for Adaptive Parallel Search 56
3.1 Iterative bisection partitioning . . . 57
3.2 SPREAD . . . 59
3.2.1 Portfolio phase . . . 60
3.2.2 Splitting phase of SPREAD-S . . . 62
3.2.3 SPREAD-D . . . 66
3.3 Experimental design . . . 67
3.4 Experimental results . . . 69

4 Search Effort Estimation to Boost SPREAD 77
4.1 Search effort estimation . . . 78
4.2 GRE, a recursive estimator . . . 81
4.2.1 The GRE algorithm . . . 81
4.2.2 Properties of GRE . . . 83
4.3 EHP, a stationary estimator . . . 89
4.4 SPREAD*, an advanced adaptive parallel paradigm . . . 94

5 Conclusions 101
5.1 Summary of the contributions . . . 102
5.2 Discussion . . . 103
5.3 Future work . . . 108

Appendix A Available CSP solvers in the Fourth International CSP Solver Competition 110

Appendix B Features used in Mistral 1.550 111

Appendix C Configuration of Mistral-1.331 for experiments in Chapter 3 112

Appendix D Configuration of Mistral-1.550 for experiments in Chapter 4 113

References 115

List of Figures

1.1 Relation among the methods developed in this work 6
1.2 Constraint graph for a CSP 9
1.3 Search tree representations 12

2.1 An algorithm schedule 17
2.2 An example performance matrix for B = 20. 19
2.3 Algorithm portfolio construction as offline learning. 21
2.4 Parallel schedule returned by RSR-WG for Example 2.1 29
2.5 RSR-WG vs. Oracle 39
2.6 ROC curves for PLD predictors on two receptors (a) gpb and (b) pdg . . 50
2.7 ROC curves for MAMC with different $|L|$ 51
2.8 ROC curves for confidence analysis on (a) gpb and (b) pdg 53
2.9 Performance of prediction directly from similarity 54

3.1 Three splitting methods 58
3.2 Splitting phase of Spread-S 63
3.3 Problem distribution 67
3.4 Comparison of Spread-D to benchmark methods 70
3.5 Comparison of the cumulative number of solved problems 71
3.6 Cumulative number of problems from the hard set solved by Spread-S . 74
3.7 Scalability of Spread 75

4.1 Random Sampling with RS 79
4.2 Categorization of nodes of a search tree 81
4.3 Search trees and their size estimation 84
4.4 Two probe-equivalent full binary trees 86

4.5 Left-dominant, full binary search trees for $k \leq 3$ 87
4.6 The impact of cutoff c and iteration K on EHP's estimation accuracy for the unsatisfiable CSP rlfapScens11_f3 91
4.7 Ratio of EHP's estimate to the true search effort 94

List of Tables

2.1 Weight functions for WG 26
2.2 Competition problems by category in CPAI'08 36
2.3 Benchmark results for the 3rd International CSP solver competition . . 36
2.4 Performance of 3 parallel portfolio constructors on 2865 problems 37
2.5 Mean and standard deviation for the number of problems solved by RSR-WG 40
2.6 Comparison of RSR-WG and COM with $B = 1800$ 42
2.7 Comparison of RSR-WG and COM with $B = 180$ 42
2.8 Comparison of RSR-WG and COM with $B = 18$ 43

3.1 Speedup of Spread-D to other approaches. 70
3.2 Cumulative numbers of solved problems for harder problem set. 72
3.3 Challenge problem solution time in seconds for Spread-S (S-S), Spread-D (S-D), and benchmark approaches. 73
3.4 Scalability of Spread 75

4.1 Nodes expanded by Mistral per problem class 91
4.2 Comparison of estimation errors for EHP, GRE, GWBE, and RS on 226 CSP instances of four CSP classes 92
4.3 Comparison of Spread*, and Spread-D and Spread-I on 16 processors. 96
4.4 Comparison of Spread*, and Spread-D and Spread-I on 32 processors. 98
4.5 Comparison of Spread*, and Spread-D and Spread-I on 64 processors. 99
4.6 Average part of runtime devoted to probing in Spread* 99
4.7 Average percentage of search time saved by Spread* compared to Spread-R on 16, 32, and 64 processors over 30 runs 100

List of Algorithms

2.1 Weighted Greedy 25
2.2 RSR-WG 27
2.3 MAMC 45
3.1 Portfolio phase (Manager) 61
3.2 Worker 62
3.3 RS-IBP (Manager) 64
4.1 est(π) 82
4.2 EHP(c, K, w) 89

1

Introduction

The power of constraint satisfaction for abstraction and representation provides a common basis for the analysis and solution of a variety of problems from both artificial intelligence and operations research. Constraint satisfaction has been applied to a broad variety of problems in planning [1], scheduling [2], and resource allocation [3]. In general, solving a constraint satisfaction problem (CSP) [1] on finite domains is NP-complete [4].

A CSP is typically addressed by a *sequential* CSP solver, that is, one running on a single processor (henceforth, simply a *solver*). Researchers in constraint satisfaction have proposed many techniques to speed the search of solvers. Although those methods have dramatically improved the power of modern solvers, they still cannot solve a significant number of CSPs (e.g., the results of the MiniZinc Challenge 2012 [5]). In fact, the pace of performance improvement obtained from algorithmic and heuristic innovation and modification has slowed. For example, many CSPs that went unsolved during the Third International CSP Solver Competition remained open in the competition in the following year [6, 7].

Meanwhile, parallelization for constraint search has drawn considerable recent at-

[1] This thesis provides a glossary immediately before the references.

tention. Indeed, parallel computing resources (e.g., multicore computers, clusters, and computational grids that connect clusters through the Internet) are increasingly available at decreasing cost. Parallelization provides sequential solvers an opportunity to solve open CSPs and to speed their search on solved instances.

Although parallelization of constraint search can exploit quickly increasing massive computing resources, it is rarely straightforward. Parallelization of pre-existing modern solvers can benefit from the efficiency of mature modern solvers without the enormous effort to reconstruct and retune them (e.g., [8, 9]). Unfortunately, modern solvers usually adopt comprehensive design decisions that pose substantial challenges to their efficient and effective parallelization.

The motivation of this work is to develop efficient and effective methods to parallelize modern solvers on identical processors. **The thesis of this work is that learning can boost parallelization and thereby accelerate CSP solution**. Throughout, *learning* refers to both machine-learning methods (e.g., case-based reasoning) and to heuristic search (e.g., weight learning). To benefit from state-of-the-art solvers, this work seeks convenient parallelization methods for pre-existing solvers, rather than construct a new solver. This work shows empirically that it is possible to leverage different solvers to outperform each of them, and develops a method that conveniently parallelizes a solver to address problems too challenging for its sequential version. These results demonstrate the promise that lies in the combination of learning and parallelization for the solution of CSPs, particularly those that require substantial search effort for solution by sequential solvers.

The remainder of this thesis is organized as follows. Section 2 discusses parallel algorithm portfolios. Section 3 proposes a hybrid parallel paradigm for constraint search.

Section 4 improves the paradigm with search effort estimation. Section 5 discusses the impacts and issues of the proposed methods, and future work.

The remainder of this section is organized as follows. Section 1.1 presents the contributions and principal results of this work, and lists the papers published as a result. Section 1.2 defines terminology for CSPs and constraint search. Sections 1.3 and 1.4 describe the common search mechanism for modern solvers, and additional methods. Section 1.5 provides a general introduction on solver scheduling and parallelization.

1.1 Contributions and principal results

Research on parallelization for CSP solvers includes a broad spectrum of parallel programming models (e.g., OpenMP [10, 11, 12], Message Passing Interface (MPI) [13, 14]) and a variety of platforms (e.g., single node [10, 12], cluster [14, 15], and grid [16]), on a scale from a few processors to thousands. As the platform for parallelization, this work considers a non-shared memory environment; communication between multiple processors is restricted to a local network. (A typical example of such an environment is a computer cluster, a set of high-performance computers connected through a local network.) This assumption gives the proposed methods applicability to a variety of environments and scales.

A *scheduler* is an algorithm that schedules several algorithms (or solvers) for execution on a single processor. A *parallel scheduler* is a parallel algorithm scheduling method that schedules several algorithms to run concurrently on multiple processors. Conventionally, such schedules are called algorithm portfolios, and may be either non-parallel or parallel.

This work begins with solver scheduling that treats each solver as a black box.

It presents *weighted greedy* (*WG*), a scheduler for non-parallel algorithm portfolios based on case-based reasoning and a greedy algorithm. It then formulates parallel algorithm portfolio construction as an integer-programming problem, and generalizes WG to RSR-WG (i.e., WG with the heuristics *retain*, *spread*, and *return*). *RSR-WG* is a parallel scheduler based on a property of the optimal solution to the inherent integer-programming problem. To address a set of problems one at a time, RSR-WG creates portfolios of deterministic algorithms offline. This work also generalizes the reasoning component of RSR-WG to a new paradigm, *multi-agent multi-case-based reasoning* (*MAMC*), and applies MAMC to protein-ligand docking.

RSR-WG provides a convenient way to parallelize constraint solving; it requires no message exchanges between solvers. The performance of any algorithm portfolio is, however, bounded by that of an *oracle scheduler* that always selects the fastest solver for each problem. Indeed, the use of algorithms as black boxes eliminates any opportunity to improve the performance of an individual algorithm by parallelization.

Another way to parallelize constraint search is to partition the problem. *Search space splitting* (*SSS*) partitions the search space of a CSP into subspaces and uses different processor to explore different subspaces. This work introduces an SSS method, *iterative bisection partitioning* (*IBP*), for balanced workload partitioning of CSPs. Based on IBP, this work introduces Search by *Probing and REcursive Adaptive Domain-splitting* (SPREAD), a hybrid adaptive paradigm that uses IBP to harness parallel computation and enhance the performance of an underlying sequential constraint solver. SPREAD is intended for an MPI platform, where it can pass messages between processors. In addition, SPREAD requires only minimal programming for use with modern solvers.

Intuitively, SPREAD is a parallel version of branch and bound for constraint search,

where the bound is the cutoff assigned to each split subproblem. Although it performs well on many CSP instances, SPREAD can also benefit from more effective partitioning. This work develops a new approach, *estimation by heuristic probing* (*EHP*), for search effort estimation of CSPs, and investigates both its theoretical bias and practical efficiency. Given EHP's empirical accuracy, this work incorporates EHP into SPREAD to improve its performance. The enhanced paradigm, SPREAD*, more intelligently orders and subdivides subproblems. Despite the overhead introduced by EHP's probing, SPREAD* can balance processor workload better and further improve the performance of parallelized constraint search.

In summary, this work makes the following contributions:

- WG, a scheduler for non-parallel algorithm portfolios
- The formulation of parallel algorithm portfolio construction as an integer-programming problem
- RSR-WG, a generalization of WG for parallel algorithm portfolios
- MAMC, an application-independent generalization of WG
- IBP, a method for balanced workload partitioning of CSPs
- SPREAD, a hybrid adaptive paradigm to parallelize a solver
- GRE, a generalized recursive search tree size estimator
- EHP, a heuristic-based probing method for search effort estimation
- SPREAD*, an enhanced version of SPREAD with EHP

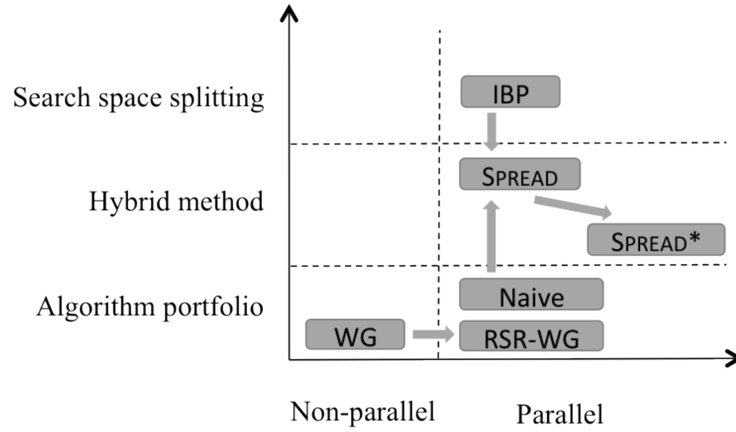


Figure 1.1: Relation among the methods developed in this work

Figure 1.1 illustrates the relationships among these methods.

These are the principal results:

- WG outperformed all other scheduling methods on a benchmark set of CSP instances from recent solver competition.
- Given several additional processors, RSR-WG can construct algorithm portfolios whose performance is competitive with that of an oracle.
- MAMC improves predictive accuracy for protein-ligand docking.
- SPREAD significantly improves the performance of its solver, outperforms a variety of reasonable alternatives, and solves many difficult CSP instances left open after recent solver competitions.
- EHP precisely estimates search tree sizes on a variety of CSP instances.
- SPREAD* balances processor workload better and solves a variety of challenging CSP instances more quickly than SPREAD.

This work consists of an overview and includes material from the following papers:

Susan L. Epstein, **Xi Yun** (2010). From Unsolvable to Solvable: An Exploration of Simple Changes. In *AAAI-10 Workshop on Abstraction, Reformulation, and Approximation (WARA-2010)*, Atlanta, GA.

Xi Yun, Susan L. Epstein (2012). Learning Algorithm Portfolios for Parallel Execution. In *Proceedings of the Sixth Learning and Intelligent OptimizatioN Conference (LION6)*, Paris, France. pp. 323-338.

Xi Yun, Susan L. Epstein (2012). Adaptive Parallelization for Constraint Satisfaction Search. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS-2012)*, Niagara Falls, Canada.

Xi Yun, Susan L. Epstein (2012). A Hybrid Paradigm for Adaptive Parallel Search. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP-2012)*, Quebec City, Canada. pp. 720-734.

Xi Yun, Susan L. Epstein, Weiwei Han, Lei Xie (2013). Case-Based Meta-Prediction for Bioinformatics. Accepted by the Twenty-Fifth Annual Conference on Innovative Applications of Artificial Intelligence (*IAAI-2013*), Bellevue, Washington, USA.

Susan L. Epstein, **Xi Yun**, Lei Xie (2013). Multi-Agent, Multi-Case-Based Reasoning. Accepted by the Twenty-First International Conference on Case-Based Reasoning (*ICCBR-13*), Saratoga Springs, NY, USA.

Xi Yun, Susan L. Epstein (2013). Search Effort Estimation Boosts Parallelized Constraint Search. Under review.

Xi Yun, Susan L. Epstein, Weiwei Han, Lei Xie (2013). A Case-based Meta-Learning Algorithm Boosts the Performance of Structure-Based Virtual Screen-

ing. Under review.

1.2 Definitions

A constraint satisfaction problem P is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of discrete domains associated with those variables, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints. Each constraint C_j has a relation R_j that restricts how the variables in its scope S_j may be assigned simultaneously. A constraint can be defined on one variable (a *unary* constraint), two variables (a *binary* constraint), or n variables (an *n-ary* constraint, $n > 2$). An *extensional* constraint explicitly represents a constraint as a set of tuples; an *intensional* constraint implicitly describes a constraint with a predicate. A *binary* CSP is one with only unary and binary constraints; it can be conveniently visualized by a *constraint graph*, where each vertex represents a variable and each edge represents a binary constraint. The *Boolean satisfiability problem* (*SAT*) and the *satisfiability modulo theories* (*SMT*) are special cases of CSPs.

An instantiation of a CSP assigns values to some of its variables from their respective domains. An instantiation of all the variables is a *complete* instantiation, and a complete instantiation that satisfies all the constraints is a *solution*. A CSP with at least one solution is *solvable*, otherwise it is *unsolvable*. A partial instantiation that does not appear in any solution is a *nogood*. To *solve* a CSP is to find one or more solutions for it, or to prove that none exists. Modeling a problem as a CSP is intended to efficiently support search for a solution to it.

Figure 1.2 shows the constraint graph of a CSP instance (henceforth, Example 1.1) with four variables A, B, C, and E, each associated with domain $D = \{1, 2, 3, 4\}$.

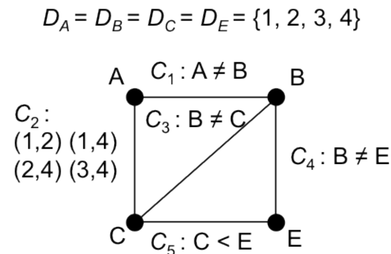


Figure 1.2: Constraint graph for a CSP

This instance has five constraints, which are all binary. C_2, an extensional binary constraint, itemizes permissible tuples for the assignments to A and C; the other constraints are intensional. $\{A = 1, B = 2\}$ is a nogood. This example has two solutions $\{A = 1, B = 3, C = 2, E = 4\}$ and $\{A = 1, B = 4, C = 2, E = 3\}$.

Systematic backtracking (*BT*) sequentially assigns values to variables and validates the consistency of each assignment with propagation. A *consistent* instantiation is one that violates no constraint. After each assignment, propagation temporarily removes any inconsistent value from the domains of all *future* (as yet unassigned) variables. For each future variable X_i, propagation creates a *dynamic domain* $D'_i \subseteq D_i$ that it believes consistent with the current assignment. A *wipeout* occurs when some D'_i becomes empty, which forces search to retract to the most recent assignments until it detects an alternative. Search terminates when a solution is found, or when the problem is proved to be unsolvable.

Propagation infers the implications of a value assignment to one variable on the domains of the future variables; it enforces some level of local consistency. When used either after each assignment during search or as a preprocessing method before search, propagation can eliminate portions of the search space from consideration, and thereby improve search performance [17, 18]. *Forward checking* is a propagation method that

removes inconsistent values from the domains of variables adjacent to X_i after a value assignment to it. Let R_{ij} be the compatible tuples defined by constraint C_{ij} on variables X_i and X_j. X_i is said to be *arc consistent* (AC) relative to X_j if and only if for every $x \in D'_i$, there exists at least one value $y \in D'_j$ such that $(x, y) \in R_{ij}$. The constraint on X_i and X_j is arc consistent if and only if X_i and X_j are arc consistent with one another. A (binary) CSP is said to be *arc consistent* if and only if all its constraints are arc consistent. Maintaining arc consistency (MAC) is a propagation method that enforces AC before search and after each assignment during search. MAC is the standard propagation method for binary CSP solution [18], and applied in the empirical work reported here. Further details on other kinds of consistency can be found in [19, 20, 21].

1.3 Solvers

The way that BT searches is conventionally represented by a search tree, where the root assigns no values, an internal node assigns some, and a leaf is either a wipeout or a solution. The *size* of a search tree is the number of nodes the search visits. Search trees use either k-way or 2-way branching. Let $\Gamma(v)$ be the *children* of v, all the states that can be expanded from node v by one assignment followed by propagation. The *branching factor* b of v is $|\Gamma(v)|$. Under k-way branching, $\Gamma(v)$ consists only of states expanded by value assignments to the same variable chosen at state v. Under 2-way branching, however, the assignments that produce $\Gamma(v)$ need not be to the same variable.

A *variable-ordering heuristic* chooses the next variable to be assigned a value, and ideally makes decisions that improve search performance. An important underlying principle for variable-ordering heuristics is *fail first*; it advocates exploration where search is mostly likely to fail [22]. Boussemart and colleagues proposed a family of

variable-ordering heuristics based on *constraint weights* [23], where a constraint weight counts how often a particular constraint has lead directly to a wipeout during propagation. In addition, the weighted degree of a variable is the sum of the constraint weights on the edges from this variable to other future variables. A high constraint weight suggests extensive contention in a CSP. Besides weight-based approaches, a variable-ordering heuristic can also choose variables based on variable impact [24] or activity [25].

A *value-ordering* heuristic chooses the next value for a selected variable that awaits instantiation. Value-ordering heuristics usually follow the *promise* principle: choose a value that is the most likely to be part of a solution. Here, likelihood either estimates the number of solutions that include each value [26], or approximates the probability that a partial assignment will lead to a solution [27].

Figures 1.3 (a) and (b) compare k-way and 2-way branching for BT on Example 1.1, where the variable-ordering heuristic minimizes the ratio of dynamic domain size and weight, and both value ordering and tie breaking for variable ordering are by lexicographical order. Nodes with label X are $failures$, leaves which force the search to backtrack. Leaves with label V are solutions. Figure 1.3 (b) intuitively explains how 2-way branching gets it name. The nodes labeled NOT-EQUAL (i.e., $\neq$) in (b), however, are not actual assignments. This work therefore adopts a more compact representation, shown in Figure 1.3 (c), where each node (except the root) at depth d corresponds to an assignment at that depth. Note, for example, that the root node has children that result from an assignments to either B or C.

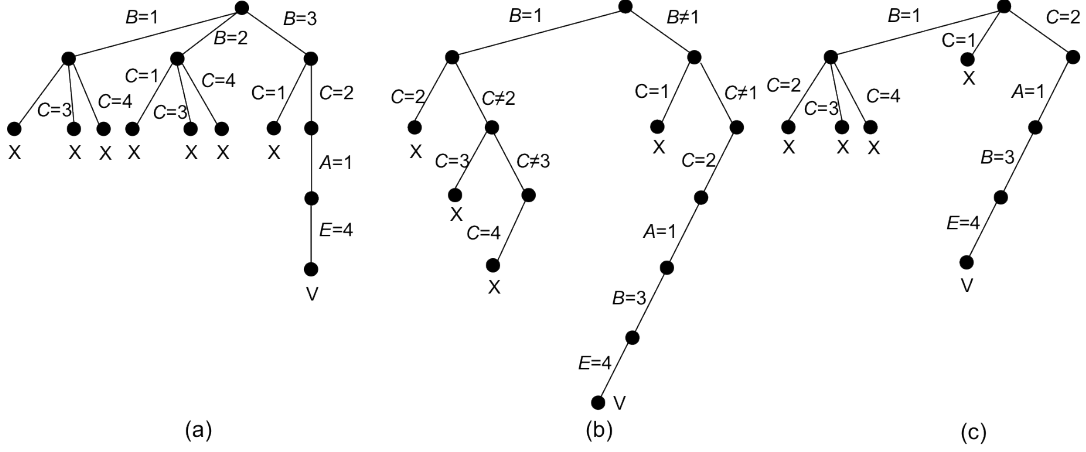


Figure 1.3: Search tree representations. Search trees produced by BT on Example 1.1 with (a) k-way and (b) 2-way branching. (c) A more compact representation of 2-way branching

1.4 Additional techniques

Once *candidates* (possible variable or value choices for the next assignment by a solver) have been ordered by a heuristic, *randomization* chooses one at random, usually from a small set of the top-ranked candidates [28, 29]. The diversity randomization introduces into search can in turn be exploited by *restart*, which halts the current search and starts a new one to extricate search trapped in a large subtree. Formally, a *restart strategy* $S = (t_1, t_2, t_3, \ldots)$ is a sequence of termination conditions t_i that trigger restart. Given full knowledge of the *runtime distribution* (the probability distribution for the number of steps an algorithm executes before it stops), Luby and colleagues showed that the optimal restart strategy has equal values for t_i [30]. They also proposed the *Luby sequence*, a universal restart strategy that relies on no knowledge of the runtime distribution:

$$t_i = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases} \tag{1.1}$$

where $k = 1, 2, \ldots$. The first seven numbers of the Luby sequence are 1, 1, 2, 1, 1, 2, 4. In the worst case, it underperforms the optimal strategy by only a logarithmic factor [30].

A solver may preprocess a problem before full search for a solution. *Preprocessing* manipulates a CSP to reduce the effort expended during the subsequent full search and to improve overall search performance. Two examples of preprocessing are consistency enforcement and probing. *Singleton arc consistency* (*SAC*) is a propagation method; it requires that a CSP remain arc consistent after the assignment of any value to a variable from its associated domain. SAC is usually too time-consuming for propagation during search, but it may be beneficial for preprocessing [17]. *Probing* refers to any method that collects information about a CSP in an independent phase and then exploits that information to enhance the subsequent search. For example, Wallace and Grimes learned constraint weights during preprocessing, and then used them in a variable-ordering heuristic during the subsequent full search [31]. As another example, Epstein and Li used *Foretell* in preprocessing to extract *clusters* (tight and dense substructures), and then guided search toward variables in those clusters [32].

When wipeout occurs, BT can *chronologically backtrack* to the most recently assigned variable, or it can return to any preceding variable that caused the failure. One such *look-back* technique is conflicted-directed backjump (*CBJ*), which maintains a conflict set for each variable. When wipeout occurs on X_i, CBJ returns to the most recently assigned variable in the conflict set associated with X_i. *Nogood learning* records inconsistent assignments so that backtracking can avoid them in subsequent search [33]. A generalized form of nogood learning for CSPs was introduced in [34].

A modern CSP solver, the kind used in this work, is typically a complex combina-

tion that includes (but is not limited to) modeling, fundamental search algorithms, preprocessing techniques, variable-ordering heuristics, value-ordering heuristics, constraint propagation, look-back techniques, nogood learning strategies, and restart policies. Appendix A lists the CSP solvers that participated in a recent CSP solver competition [6].

The performance of a solver is usually measured by its runtime on a set of CSPs. Other metrics include search tree size, number of consistency checks, and number of backtracks. Improvements in solver performance have been driven by the development of effective new methods and techniques, the application of a better parameter configuration (possibly automatically [35]), collaboration among multiple solvers, and the exploitation of parallel computing power. All are touched upon here.

This chapter has introduced the contributions and principal results of this work, and supplied a brief background. The next chapter discusses one proposed approach to parallel constraint search, the parallel algorithm portfolio. This method requires no modification of its underlying solvers, and it applies even if the structure of those solvers is opaque. Empirical results also demonstrate its effectiveness.

2

Parallel algorithm portfolios

Portfolio-based methods leverage several solvers to enhance and stabilize search for a solution to a CSP. These methods often treat a solver as a black box and restrict communication between solvers, and therefore can be conveniently applied to a variety of solvers without knowledge of their internal structures. Although algorithm portfolios for constraint solvers have received much recent attention, their promise for parallel execution on multiple processors has received relatively little.

The first five sections of this chapter address parallel algorithm portfolios with constraint satisfaction solvers. Section 2.1 describes fundamental definitions. Section 2.2 formulates the learning of algorithm portfolios. Sections 2.3 and 2.4 respectively propose a greedy method and a complete method for algorithm portfolio construction. Section 2.5 compares the performance of both methods to a variety of portfolio constructors on varied CSPs from a recent solver competition. Finally, Section 2.6 demonstrates the potential power of this approach beyond constraint satisfaction.

2.1 Definitions

An algorithm portfolio for CSP solution was originally defined as a method that combined different algorithms to improve search performance while it lowered *search risk*, the standard deviation of a performance metric (e.g., expected CPU time or number of backtracks required to solve a problem) [36, 37]. In other words, an algorithm portfolio searched for a Pareto frontier in the two-dimensional space defined by a given performance metric and its standard deviation. Later, an algorithm portfolio was generalized to denote a combination of different algorithms intended to outperform the search performance of any of its constituent algorithms [38, 39, 40, 41, 42, 43]. This work extends that formulation for parallel execution on a set of processors. Although the principal focus is on portfolios of CSP solvers, the proposed methods can construct a portfolio from other sets of candidates, such as a set of search algorithms, a set of heuristics, or a set of predictors. This work adopts the term algorithm portfolio for any such scenario.

Given a set $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ of n algorithms, a set $\mathcal{P} = \{x_1, x_2, \ldots, x_m\}$ of m problems, and a set of B consecutive time intervals $T = \{t_1, t_2, \ldots, t_B\}$, an algorithm portfolio allots computing resources on $K \geq 1$ processors for algorithms in $\mathcal{A}$ to solve problems from $\mathcal{P}$ within time limit $\sum_{i=1}^{B} t_i$. Without loss of generality, this work simplifies T to $[B] = \{1, 2, \ldots, B\}$. (Section 2.5.3 discusses the impact of adjusting interval length.) This work also assumes that, for each x_i, there exists at least one a_j that can solve x_i within B.

A *simple algorithm schedule* S for a problem specifies which algorithm addresses the problem in each time interval on a single processor, that is, $S : T \to A$. *Algorithm selection* seeks a simple algorithm schedule that schedules only one algorithm [42, 44].

An *algorithm schedule* $\mathcal{S}$ for K identical processors is a set $\{S_1, \ldots, S_K\}$ of K simple schedules, one for each processor. An algorithm schedule is a specific algorithm portfolio that determines not only how algorithms share computing resources, but also the order in which they execute. A simple algorithm schedule can be represented by an ordered list of algorithm-duration pairs; an algorithm portfolio can be represented by a set of such order lists. For example, the algorithm schedule in Figure 2.1 can be represented by $\mathcal{S} = \{S_1 = \{\langle a_1, 10\rangle, \langle a_2, 10\rangle\}, S_2 = \{\langle a_3, 3\rangle, \langle a_4, 7\rangle\}\}$

Figure 2.1: An algorithm schedule. This schedule has $|\mathcal{A}| = 4$ algorithms, $K = 2$ processors, and time limit $B = 20$.

A *deterministic* algorithm consistently produces the same output given the same problem and time cutoff. In contrast, the output of a randomized algorithm may change from one run to the next. Given $\mathcal{A}$ and $\mathcal{P}$, an oracle solver always chooses the best solver in $\mathcal{A}$ to solve each problem in $\mathcal{P}$. Of course, neither a simple nor a parallel schedule can outperform an oracle's perfect algorithm selection on $\mathcal{P}$.

Given time limit B, at most B time can be allotted to any algorithm on one processor. The performance of $\mathcal{A}$ on $\mathcal{P}$ can thereby be represented as an m by n performance matrix τ, where the entry $\tau_{ij} \in [B]$ is the runtime that a_j requires to solve x_i. For a deterministic algorithm, each τ_{ij} is fixed; for a randomized algorithm, τ_{ij} is a random number.

Given a problem, a *sequential* algorithm schedule executes algorithms in a specific order, but does not preserve any intermediate search states for an algorithm when the schedule leaves it. Thus, a sequential schedule must restart on the problem if it later reapplies a previous algorithm to it. In contrast, a *switching* algorithm portfolio interleaves algorithms, and preserves intermediate search states, so that search can continue from a previous state when it returns to an earlier algorithm.

A *static* algorithm schedule is constructed in advance, and goes unchanged during search. In contrast, a *dynamic* algorithm schedule can profit from feedback as it executes, and adjust its behavior accordingly. For example, the dynamic algorithm schedules in [45, 46] iteratively share a (possibly varying-length) time slice among all available algorithms, but modify the algorithms' relative priorities based on their progress. Adjustments for a dynamic schedule can be triggered by unsatisfactory performance during execution [47, 48]. Most of the work referenced thus far is for simple schedules, which interleave algorithms on a single processor.

Figure 2.2 shows τ for four deterministic algorithms $\mathcal{A} = \{a_1, \ldots, a_4\}$ on a problem set $\mathcal{P} = \{x_1, \ldots, x_4\}$, where $B = 20$. For this example, the simple sequential schedule $S_1 = \{\langle a_2, 4\rangle, \langle a_3, 1\rangle, \langle a_1, 9\rangle\}$ solves x_4 with total runtime $t = 14$; and the simple switching schedule $S_2 = \{\langle a_2, 8\rangle, \langle a_4, 5\rangle, \langle a_2, 2\rangle\}$ solves x_1 with $t = 8$, or solves x_3 with $t = 15$. The schedule $\mathcal{S} = \{S_1, S_2\}$ on two processors can thereby solve x_1, x_3, or x_4, each with $t < B$. In practice, τ is unknown until each problem in $\mathcal{P}$ is solved by all the algorithms in $\mathcal{A}$.

Portfolio problems may have different objective functions for optimization. For example, a portfolio may be required to minimize its expected runtime on a problem generated at random from some problem distribution. (Alternatives are introduced in

$$
\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}
\begin{array}{c} \begin{array}{cccc} a_1 & a_2 & a_3 & a_4 \end{array} \\ \left[\begin{array}{cccc} 11 & 8 & 19 & - \\ - & 17 & - & 18 \\ 16 & 10 & - & - \\ 9 & - & 11 & - \end{array} \right] \end{array}
$$

Figure 2.2: An example performance matrix for B = 20.

[43].) Recent CSP solver competitions have evaluated solvers on how many problems they solved under a fixed, per-problem time limit, and have broken ties on average solution time across solved problems [6, 7]. This work compares algorithm portfolio construction methods with the same standard. Under this criterion, a_2 is the winner for the example in Figure 2.2. (In contrast, SAT solver competitions have compared solvers with a complex scoring function that includes the performance of all competitors [49].)

Algorithms introduced in this chapter intensively depend on *case-based reasoning* (*CBR*), which solves a new problem based on learned experience during the solution of similar past problems. CBR is usually formalized as a four-step procedure: retrieve, reuse, revise, and retain [50]. Given a new problem, *retrieve* gets some relevant case from all cases stored for reference. *Reuse* then maps the solution of the relevant case to the new problem, with any modifications necessary to adapt to the new situation. *Revise* adjusts the preliminary solution to correct any problem observed during testing. Finally, after successful solution of the new problem, *retain* stores the new problem with the resultant solution in memory for future use.

2.2 Learning an effective algorithm portfolio

A *constructor* creates an algorithm portfolio for a portfolio problem. A *scheduler* is a constructor that constructs algorithm schedules. A *converting heuristic* changes an algorithm portfolio into an algorithm schedule by specifying the execution order of algorithm-duration pairs on every processor.

Algorithm portfolio constructors that learn are classified as online or offline based on the way they use their training problems. An *offline constructor* observes the performance of algorithms on a set of training problems and then builds a portfolio of those algorithms to optimize its performance on an entire testing set [39, 40, 42]. An *online constructor* establishes one portfolio for an individual problem at a time, and the knowledge it relies on for that problem comes only from the problems that preceded it [45, 46]. This work focuses on offline algorithm constructors.

Figure 2.3 represents offline algorithm portfolio construction with feature extraction as a machine-learning task. Given a set $\mathcal{P}_{train}$ of training problems, a set $\mathcal{P}_{test}$ of testing problems, and a performance matrix $\tau(a, x)$ that stores the time required by each algorithm $a \in \mathcal{A}$ to solve each problem $x \in P_{train}$, the constructor's task is to find a schedule S with optimal performance that uses $\mathcal{A}$ to solve $\mathcal{P}_{test}$. Here, all entries in τ are discrete, fixed positive integers, that is, all algorithms are assumed to be deterministic. $\mathcal{P}^*(y)$ is the *neighbor set* of testing problem $y \in \mathcal{P}_{test}$, a set of training CSPs similar to y. The similarity of two CSPs is measured by the Euclidean distance between their feature vectors. (Appendix B lists the features used for the experiments in this chapter.) Portfolios of randomized algorithms are discussed in [40, 51], which are outside the scope of this work.

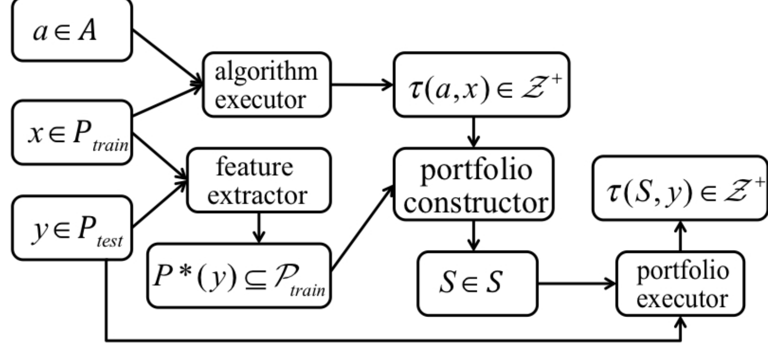


Figure 2.3: Algorithm portfolio construction as offline learning.

2.2.1 Optimal portfolio construction

Given a testing example y and its neighbor set $\mathcal{P}^*(y)$, this work assumes that a portfolio that solves the most problems in $\mathcal{P}^*(y)$ within B per neighbor is also most likely to solve y within B. Let $f(\mathcal{S})$ be the number of problems solved by a schedule $\mathcal{S}$ in $\mathcal{P}^*(y)$ within time B per problem on K processors. Then the optimization objective of a constructor is to find a schedule $\mathcal{S}$ such that, within time B, solves as many neighbors as possible:

$$\max \ f(\mathcal{S}) \tag{2.1}$$

$$s.t. \quad length(S_k) \leq B, \quad \text{for } 1 \leq k \leq K \tag{2.2}$$

To exploit similar training examples more effectively, a case-based constructor can refine f to be weakly optimal or strongly optimal. Let indicator $\chi_{ij}(t)$ record whether a_j solves x_i within t

$$\chi_{ij}(t) = \begin{cases} 1 & \text{if } \tau_{ij} \leq t \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

In a *weakly-optimal portfolio problem*, the objective is to choose an $\mathcal{S}$ that maximizes f_w, the similarity-weighted sum of training problems solved under $\mathcal{S}$:

$$f_w(\mathcal{S}) = \sum_{i=1}^{m} w_i (1 - \prod_{k=1}^{K} \prod_{j=1}^{n} (1 - \chi_{ij}(t_j^k))) \tag{2.4}$$

where $m = |\mathcal{P}^*(y)|$, $n = |\mathcal{A}|$, w_i is the weight of the ith problem in $\mathcal{P}^*(y)$, and t_j^k is the time $\mathcal{S}$ allocates to algorithm a_j on the kth processor. Since the allocation of more time to any algorithm never reduces the number of solved problems, f_w is monotonic increasing, that is, for any $\mathcal{S}_1$ and $\mathcal{S}_2$, $f_w(\mathcal{S}_1) \leq f_w(\mathcal{S}_1 \oplus \mathcal{S}_2)$ and $f_w(\mathcal{S}_2) \leq fw(\mathcal{S}_1 \oplus \mathcal{S}_2)$, where $\oplus$ concatenates time durations for each algorithm on each processor.

In a *strongly-optimal portfolio problem*, the objective is to maximize f_w and, secondarily, to minimize the total time expended. Formally, the objective function f_s for a strongly-optimal portfolio problem is

$$f_s(\mathcal{S}) = \left\{ \frac{KB+1}{\min_i \{w_i\}} \sum_{i=1}^{m} w_i (1 - \prod_{k=1}^{K} \prod_{j=1}^{n} (1 - \chi_{ij}(t_j^k))) + \sum_{k=1}^{K} (B - \sum_{j=1}^{n} t_j^k) \right\} \tag{2.5}$$

where the first summand is the weighted number of solved problems, and the second term is the total unused runtime M. The factor $\frac{KB+1}{\min_i\{w_i\}}$ ensures that solving one additional problem is always more beneficial than increasing the remaining time $M(\leq KB)$.

2.2.2 Properties of optimal schedules

Unlike f_w, f_s is not monotonic increasing. Indeed, given schedules $\mathcal{S}_1$ and $\mathcal{S}_2$, if $\mathcal{S}_1 \oplus \mathcal{S}_2$ does not solve more problems than $\mathcal{S}_1$, then $\mathcal{S}_1 \oplus \mathcal{S}_2$ is a poorer schedule than $\mathcal{S}_1$, since it wastes more time. Our first lemma indicates that an optimal solution under f_s schedules each algorithm on at most one processor.

Lemma 2.2.2.1. *Let* $\mathcal{S}^* \in \{\mathcal{S} : \operatorname{argmax}_{\mathcal{S}} f_s\}$. *For any* $a_j \in \mathcal{A}$, *there exists at most one* $k \in [K]$ *such that* $t_j^k > 0$ *in* $\mathcal{S}^*$.

Proof. This proof shows that, if $\mathcal{S}^*$ has more than one $k \in [K]$ such that $t_j^k > 0$, $\mathcal{S}^*$ cannot be an optimal schedule. Without loss of generality, assume that for some schedule $\mathcal{S}'$ for f_s, two different processors k and q have $t_j^q \geq t_j^k > 0$ for some a_j. Since

$\chi_{ij}(t)$ is monotonically non-decreasing for all j, $\chi_{ij}(t_j^q) \geq \chi_{ij}(t_j^k)$. Let $\mathcal{S}''$ be identical to $\mathcal{S}'$ except that $\mathcal{S}''$ allots no time to a_j on the processor k, that is, $\mathcal{S}''$ has $t_j^k = 0$. This reduction does not impact the first summand in formula 2.5. Indeed, if $\chi_{ij}(t_j^q) = 1$, then $1 - \prod_{k=1}^{K} \prod_{j=1}^{n} (1 - \chi_{ij}(t_j^k))$ always equals 1, while if $\chi_{ij}(t_j^q) = 0$, then $\chi_{ij}(t_j^k) = 0$ both before and after such a reduction. The reduction does, however, increase the remaining unused runtime M by t_j^k. $\mathcal{S}''$ is therefore better than $\mathcal{S}'$, which means that $\mathcal{S}'$ cannot be optimal. □

Possible optimal schedules under f_s can be further constrained on the execution duration for each algorithm. Given performance matrix τ, denote the ordered runtimes for all problems solved by each a_j as $0 < \tau_{(1)j} < \cdots < \tau_{(m_j)j} \leq B$, where duplicate values are removed and problems unsolved by a_j are ignored. Let $\mathcal{S}_0$ denote the set of all algorithm-duration tuples, where the duration of a_j is restricted to $\{\tau_{(i)j}\}$. The next lemma shows that tuples in $\mathcal{S}_0$ are building blocks for an optimal solution under f_s. Thus the number of choices of t_j^k for a_j in an optimal solution under f_s is bounded by m_j, which may be much smaller than B.

Lemma 2.2.2.2. *Any* $\mathcal{S}^* \in \{\mathcal{S} : \underset{\mathcal{S}}{\operatorname{argmax}}\ f_s\}$ *can be represented as a set of algorithm-duration tuples* $\{\langle a_j, t_j^k \rangle\}$*, where* $\langle a_j, t_j^k \rangle \in \mathcal{S}_0$*, and* k *is unique for each* a_j*.*

Proof. By Lemma 2.2.2.1, an optimal solution can be represented as a set of tuples $\langle a_j, t_j^k \rangle$, where a_j executes only on processor k with duration t_j^k. Assume that, for some optimal schedule $\mathcal{S}'$, there exists for a_j some t_j^k such that $\tau_{(i)j} < t_j^k < \tau_{(i+1)j}$. Let $\mathcal{S}''$ be identical to $\mathcal{S}'$ except that in $\mathcal{S}''$ the runtime for a_j is reduced from t_j^k to $\tau_{(i)j}$. Clearly a_j solves exactly the same problems under $\mathcal{S}''$ as under $\mathcal{S}'$ but in less time. $\mathcal{S}''$ is therefore

better than $\mathcal{S}'$, which means that $\mathcal{S}'$ cannot be optimal. □

For the example in Figure 2.2, any optimal schedule under f_s should choose its algorithm-duration tuples from $\mathcal{S}_0 = \{\langle a_1, 9\rangle, \langle a_1, 11\rangle, \langle a_1, 16\rangle, \langle a_2, 8\rangle, \langle a_2, 10\rangle, \langle a_2, 17\rangle, \langle a_3, 11\rangle, \langle a_3, 19\rangle, \langle a_4, 18\rangle\}$.

2.3 Greedy constructor methods

2.3.1 WG

The Weighted Greedy (WG) algorithm is a greedy constructor that constructs approximate solutions to weakly-optimal switching portfolio problems on one processor. WG was inspired by GASS [52] and CPHydra [39]. For a single processor, CPHydra uses CBR to select a small set of similar training problems for each testing problem. It then does a complete search to find a schedule that solves most neighbors for the new problem. This method is efficient for small values of n, but is problematic for larger values, because it is NP-hard as a generalization of the knapsack problem [39]. In contrast, the impact of m on GASS is at worst quadratic; GASS' greedy approach is heavily dependent on the number of training problems n instead. WG exploits the fact that some problems are far more similar to a given testing problem than others, so that a properly selected subset of problems can estimate the runtime of the testing problem more precisely.

On one processor, to schedule within time limit B algorithms from $\mathcal{A}$ for a problem x^* given prior experience on a set of problems $\mathcal{P}$ (i.e., $\mathcal{P}_{train}$ in Figure 2.3), WG combines GASS and CPHydra into a single framework for switching scheduling. (See Algorithm 2.1.) WG is similar to GASS, except that it represents each problem by a numeric feature vector, and restricts its attention to similar problems (i.e., it rea-

Algorithm 2.1 Weighted Greedy

Input: training problem set $\mathcal{P} = \{x_1, \ldots, x_n\}$, weight function w, testing problem x^*, algorithm $\mathcal{A} = \{a_1, \ldots, a_m\}$, time limit B, neighbor set ratio r

Output: simple schedule S for a non-parallel algorithm portfolio

1: **for all** x_i in $\mathcal{P}$ **do**
2: Compute Euclidean distance d_i between x_i and x^*
3: **end for**
4: Collect $100r\%$ of problems in $\mathcal{P}$ closest to x^* into temporary set $\mathcal{P}_0$
5: Initialize time step $z \leftarrow 0$, overall time spent by all algorithms $T \leftarrow 0$, and overall time spent by each a_j: $t_j \leftarrow 0$
6: **for all** x_i in $\mathcal{P}_z$ **do**
7: Compute $w_i = w(x_i)$ for x_i
8: **end for**
9: **while** $\mathcal{P}_z \neq \emptyset$ **and** $T < B$ **do**
10: Select a_j with execution time Δ_z to maximize $\frac{N_j^z(t+\Delta_z)}{\Delta_z}$
11: Schedule chosen a_j with its execution duration Δ_z in $\mathcal{S}$
12: Remove all $x_i \in \mathcal{P}_z$ solved by a_j during this step
13: Update $t_j \leftarrow t_j + \Delta_z$, $T \leftarrow T + \Delta_z$, and $z \leftarrow z + 1$
14: **end while**
15: **return** $\mathcal{S}$

sons based only on similar cases). WG initially selects a neighbor set $\mathcal{P}_0$ (i.e., $\mathcal{P}^*(y)$ in Figure 2.3) that contains the $100r\%$ most similar training problems (i.e., have feature vectors closest in Euclidean distance to that of x^*), where the *neighbor set ratio* $r = \frac{|\mathcal{P}^*(y)|}{|\mathcal{P}_{train}|}$ and $0 < r \leq 1$ (Algorithm 2.1, lines 1-4). The influence of these problems in the selection of an algorithm may be uniform, or may be weighted in proportion to their distance d_i from x^*. The weight functions investigated here are shown in Table 2.1, where d_{min} denotes the smallest distance from a neighbor set problem to x^*, and d_{max} denotes the largest distance.

During each new interval Δ_z, WG counts from τ and weights how many training problems in the current neighbor set $\mathcal{P}_z$ it could solve within time $t + \Delta_z$ if it allotted

Table 2.1: Weight functions for WG. Three weight functions that measure problem similarity, where d_i denotes the Euclidean distance of testing problem x^* from the training problem x_i. Here, $\epsilon = 0.001$.

Reciprocal	Normalized weight	Normalized-fixed weight
$w_i = \frac{1}{1+d_i}$	$w_i = 1 - \frac{(n-1)(d_i - d_{min})}{n(d_{max} - d_{min})}$	$w_i = 1 - \frac{(1-\epsilon)(d_i - d_{min})}{d_{max} - d_{min}}$

that interval Δ_z to algorithm a_j:

$$N_j^z(t) = \sum_{x_i \in \mathcal{P}_z} w_i \chi_{ij}(t) \quad (2.6)$$

where χ_{ij} is defined in formula 2.3. WG then greedily maximizes $N_j^z(t)$ per unit of time expended, that is, it calculates

$$\underset{\langle a_j, \Delta_z \rangle}{\text{argmax}} \frac{N_j^z(t + \Delta_z)}{\Delta_z} \quad (2.7)$$

and removes those now-solved similar problems from $\mathcal{P}_z$ (Algorithm 2.1, line 10). Based on [52], the time complexity of WG is $O(rnm(\log rn) \min\{rn, Bm\})$ because it considers every algorithm a_j with every interval length Δ_z.

2.3.2 RSR-WG

RPWG (randomized parallel WG) is an intuitive way to parallelize WG for K identical processors $\pi_1, \pi_2, \ldots, \pi_K$. It partitions the initial neighbor set $\mathcal{P}_0$ into K subsets $\mathcal{P}_0^1, \mathcal{P}_0^2, \ldots, \mathcal{P}_0^K$ at random, and then uses WG to construct a schedule for processor π_k based on its corresponding subset $\mathcal{P}_0^k$. With uniform weights $w_i = 1$, RPWG is a naive parallel version of GASS. (To reduce the impact of randomness, RPWG could construct such a partition v times, although to conserve time $v = 1$ here.) Thus the overall complexity of RPWG is $O(vrnm \log(rn/K) \min\{rn/K, Bm\})$. Similarly, RPCPHydra, the naive parallel version of CPHydra, randomly partitions the neighbor set into K

subsets and then uses CPHYDRA on each subset to construct a schedule for each processor. Section 2.5 investigates both these naive parallel constructors as baselines. (Other recent work relevant to parallel algorithm portfolios includes online learning [45, 46] and methods that split problems [53, 54].)

Algorithm 2.2 RSR-WG

Input: training problem set $\mathcal{P} = \{x_1, \ldots, x_n\}$, weight function w, testing problem x^*, algorithm $\mathcal{A} = \{a_1, \ldots, a_m\}$, time limit B, neighbor set ratio r, processors $\{\pi_1, \ldots, \pi_K\}$
Output: schedule $\mathcal{S}$ for a parallel switching algorithm portfolio
1: **for all** x_i in $\mathcal{P}$ **do**
2: Compute Euclidean distance d_i between x_i and x^*
3: **end for**
4: Collect $100r\%$ of problems in $\mathcal{P}_0$ closest to x^* into temporary set $\mathcal{P}_N$
5: Initialize time step $z \leftarrow 0$, overall time spent by all algorithms on π_k: $T^k \leftarrow 0$, and overall time spent by a_j: $t_j^k \leftarrow 0$
6: **for all** x_i in $\mathcal{P}_z$ **do**
7: Compute $w_i = w(x_i)$ for x_i
8: **end for**
9: **while** $\mathcal{P}_z \neq \emptyset$ **and** $T < B$ for at least one π_k **do**
10: Select a_j on π_k with execution time $\Delta_z \in \mathcal{S}_0$ to maximize $\frac{N_j^z(t_j^k+\Delta_z)}{\Delta_z}$
11: Schedule chosen a_j with its execution time Δ_z on π_k in $\mathcal{S}$
12: Remove all $x_i \in P_z$ solved by a_j during this step
13: Update $t_j \leftarrow t_j + \Delta_z$, $T^k \leftarrow T^k + \Delta_z$, and $z \leftarrow z + 1$
14: **end while**
15: **for all** k such that $T^k < B$ **do**
16: **if** $T^k == 0$ **then**
17: Execute a_j on π_k for B that solves the most problems in $\mathcal{P}_0$ but not in $\mathcal{S}$
18: **else**
19: Expand the simple schedule on π_k to utilize all of B
20: **end if**
21: **end for**
22: return $\mathcal{S}$

Algorithm 2.2 describes RSR-WG, a weighted greedy algorithm that constructs a parallel switching schedule with heuristics *Retain*, *Spread*, and *Return*. Let T^k be the total duration of a simple schedule S, and let *allocation vector* $T = \langle T^1, \ldots, T^K \rangle$

record the total time allocated to each processor (i.e., $T^k = \sum_j t_j^k$)). At each step z, RSR-WG greedily chooses an algorithm by its execution duration, to maximize the weighted number of problems solved at the current time step that were not solved earlier. (Note that the schedule constructed here is not intended to be used on the training problems, but on a testing problem.) The weighted number of problems solved by algorithm a_j within t is computed in lines 6-8. By Lemma 2.2.2.1, RSR-WG chooses for a_j the processor that hosted a_j earlier, if there is one (i.e., applies the heuristic *Retain*); otherwise it chooses the processor thus far used the least (i.e., applies the heuristic *Spread*). By Lemma 2.2.2.2, RSR-WG only allocates total time t_j^k to a_j such that $t_j^k \in \mathcal{S}_0$. The while loop (lines 9-14) repeats this greedy selection until the schedule solves all problems, or no time resource is left. Finally, RSR-WG makes full use of any processor not fully utilized by $\mathcal{S}$. In particular, a completely idle processor executes the algorithm that solves the most problem in $\mathcal{P}_0$ but not in $\mathcal{S}$ for entire B (lines 17), and a partially-occupied processor executes the first algorithm on it to the end of B (i.e., applies the heuristic *Return*, line 19). RSR-WG is further clarified by following example.

Example 2.1. Consider a portfolio problem with uniform weights 1 and performance matrix τ in Figure 2.2, where the initial set of neighbors is $\mathcal{P}_0 = \{x_1, ..., x_4\}$, $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$, $K = 2$, $B = 20$. As proved in Section 2.2.2, the building blocks are $\mathcal{S}_0 = \{\langle a_1, 9\rangle$, $\langle a_1, 11\rangle$, $\langle a_1, 16\rangle$, $\langle a_2, 8\rangle$, $\langle a_2, 10\rangle$, $\langle a_2, 17\rangle$, $\langle a_3, 11\rangle$, $\langle a_3, 19\rangle$, $\langle a_4, 18\rangle\}$. Our task is to build a schedule that will be applied repeatedly to all unsolved problems. At $z = 0$, $\langle a_2, 10\rangle$ has the largest ratio 0.2 (calculated in line 10 of Algorithm 2.2, where N_j^z is the weighted number of training problems in $\mathcal{P}^z$ solved at step z by a_j), so RSR-WG greedily chooses a_2, and allots time 10 to it on processor 1. Now $S_1 = \{\langle a_2, 10\rangle\}$.

After execution of S_1, a_2 solves both x_1 and x_3, so $\mathcal{P}_0$ is reduced to $\mathcal{P}_1 = \{x_2, x_4\}$. At $z = 1$, $\langle a_2, 17\rangle$ ($\Delta_2 = 17 - 10 = 7$) has the largest ratio, 0.14, so RSR-WG greedily chooses a_2, and continues to use the same processor (i.e., applies the heuristic *Retain*). Now $S_1 = \{\langle a_2, 10\rangle, \langle a_2, 7\rangle\}$ and $\mathcal{P}_1$ becomes $\mathcal{P}_2 = \{x_4\}$. At $z = 2$, RSR-WG greedily chooses a_1 and allots 9 to it on processor 2, the first idle processor (i.e., $S_2 = \{\langle a_1, 9\rangle\}$), so $\mathcal{P}^3 = \emptyset$ (i.e., applies the heuristic *Spread*). Then to make full use ($B = 20$) of each processor, RSR-WG allots 11 more to a_1 and 3 more to a_2 (line 19 of Algorithm 2.2) (i.e., applies the heuristic *Return*). Finally, RSR-WG returns schedule $\mathcal{S} = \{S_1 = \{\langle a_2, 20\rangle\}, S_2 = \{\langle a_1, 20\rangle\}\}$ (Figure 2.4).

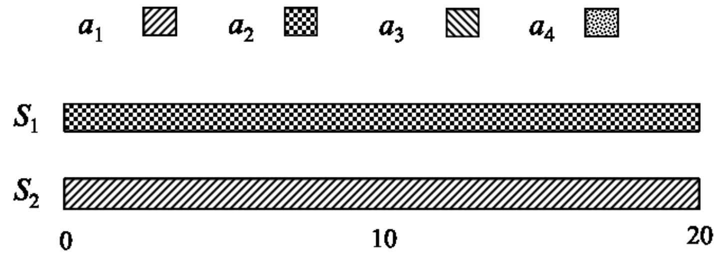


Figure 2.4: Parallel schedule returned by RSR-WG for Example 2.1. This schedule has $|\mathcal{A}| = 4$ algorithms, $K = 2$ processors, and time limit $B = 20$.

2.3.3 Bounds for RSR-WG

The performance of a schedule constructed by RSR-WG depends heavily on the performance of the schedule obtained from the its greedy procedure. In fact, when the greedy procedure generates a schedule $\mathcal{S}$ such that $T^k \leq B$ for every k, the performance of $\mathcal{S}$ has a lower bound. Proof of this bound requires Theorem 3 in [55], which is modified here as Lemma 2.3.3.1 for a weakly optimal scheduling problem. Let $G = \langle g_1, g_2, ...\rangle$ be the simple schedule defined inductively as follows: $G_1 = \emptyset, G_j = \langle g_1, g_2, ..., g_{z-1}\rangle$ for

$z > 1$, and

$$g_z = \underset{\langle a_z, \Delta_z \rangle \in \mathcal{A} \times [B]}{\text{argmax}} \left\{ \frac{f(G_z \bigoplus \{\langle a_z, \Delta_z \rangle\}) - f(G_z)}{\Delta_z} \right\} \tag{2.8}$$

Lemma 2.3.3.1. *[Streeter 07]. Let L denote any positive integer, and $T = \sum_{z=1}^{L} \Delta_z$, where $g_z = (a_z, \Delta_z)$, then $f_w(G_{\langle T \rangle}) > (1 - \frac{1}{e}) \max_{\mathcal{S}}\{f_w(\mathcal{S}_{\langle T \rangle})\}$.*

Lemma 2.3.3.1 shows that, if the overall duration of the non-parallel schedule generated by the inductive procedure equals the time limit (i.e., wastes no time), then f_w for this non-parallel schedule is always better than $(1 - \frac{1}{e})$ of the performance of any schedule within the time limit. Theorem 2.3.3.1 generalizes this result for the while loop of Algorithm 2.2, where the schedule is parallel, and the search space for $\langle a_z, \Delta_z \rangle$ is reduced.

Theorem 2.3.3.1. *Let $\mathcal{S}_A$ denote the approximation solution to f_w generated by the greedy procedure (i.e., while loop) during RSR-WG . If $\mathcal{S}_A$'s allocation vector T satisfies $T^k \leq B$ for every $k \in [K]$, then $f_w(\mathcal{S}_A) > (1 - \frac{1}{e}) \max_{\mathcal{S}}\{f_w(S_{<T>})\}$.*

Proof. Any parallel solution generated by RSR-WG for optimization under f_w is equivalent to some non-parallel solution for optimization under f_w with time limit KB. (Simply allocate each $\langle a_z, \Delta_z \rangle$ to a single processor in the order in which it was chosen.) Because RSR-WG tries to return any a_j to the processor that hosted it earlier, and no execution duration is truncated ($T^k \leq B$ for every $k \in [K]$), this conversion to a non-parallel schedule does not change any $\tau_{ij}(t_j^k)$ in equation 2.4, and thus retains the same value of f_w on S.

It remains to show that, for each step z, the choice $\langle a_z, \Delta_z \rangle$ from the while loop of RSR-WG satisfies formula 2.8. Let $\langle a_j, \Delta_j^z \rangle$ denote any selection such that the

execution duration Δ_j^z leads to a total execution time $t_j \notin \{\tau_{(i)j}\}$. This $\langle a_j, \Delta_j^z \rangle$ can always be eliminated from the search space for $\langle a_z, \Delta_z \rangle$ because if $\tau_{(i)j} < t_j^k < \tau_{(i+1)j}$, then reducing Δ_z by $t_j^k - \tau_{(i)j}$ never reduces $f_w(G_z \bigoplus \{\langle a_z, \Delta_z \rangle\})$. Thus the while loop of RSR-WG considers all possible candidates for the solution to formula 2.8. □

Because the proof of Lemma 2.3.3.1 relies on the monotonicity of f_w, the lower bound in it does not hold for optimization under f_s. Indeed, when RSR-WG generates a (strongly-optimal or weakly-optimal) schedule, time limit B may truncate some duration, and thus degrade the performance of a schedule generated by the while loop. The following example shows that there is no lower bound on performance of such a truncated schedule.

Example 2.2. Consider a weakly-optimal portfolio problem with $|\mathcal{P}| = n$, $\mathcal{A} = \{a_1, a_2\}$, $K = 1$, and $B = n + 1$. All training problems have uniform weight 1; a_1 solves x_1 in $\tau_{11} = 1$, and cannot solve any other problem within time B; for a_2, $\tau_{i2} = n+1$ for all i. The greedy procedure of RSR-WG chooses $\langle a_1, 1 \rangle$ at step $z = 1$, since it has the largest ratio, 1. Then the greedy procedure halts, since a_1 cannot solve any other problem, and a_2 cannot solve any problem within the remaining time n. The optimal solution to the weakly optimal scheduling problem is, however, $\{\langle a_2, n+1 \rangle\}$, which solves all of $\mathcal{P}$. Obviously, $f_w(\{\langle a_1, 1 \rangle\}) = 1$, $f_w(\{\langle a_2, n+1 \rangle\}) = n$, and their ratio $\frac{1}{n} \to 0$ as $n \to \infty$. A similar argument can be constructed for the strongly-optimal version.

2.4 Complete method

Intuitively, when the while loop of RSR-WG generates a schedule that is truncated by the time limit, its performance is suspect, and a complete search algorithm may

do better. This section reformulates the weakly-optimal portfolio problem on multiple processors as an IP problem and addresses it with an IP solver instead; the method is called *COM*. Let $Y = \{s_1, s_2, \ldots, s_V\}$ be a collection of sets with associated time costs $\{t_1, t_2, \ldots, t_V\}$ defined over a set of items $\mathcal{P} = \{x_1, x_2, \ldots, x_m\}$ with associated weights $\{w_1, w_2, \ldots, w_m\}$. A *K-budgeted maximum coverage problem* (*K-BMCP*) seeks K collections of sets $\{Y_1, Y_2, \ldots, Y_K\}$ such that the total cost of the sets in each collection does not exceed a given budget B, and the overall weighted sum of all items in the union of sets across all collections is maximized.

The correspondence between optimization under f_w and a K-BMCP is as follows. The item set $\mathcal{P}$ in a K-BMCP corresponds to $\mathcal{P}_{train}$ under f_w, where the associated weights have the same meaning. Recall that, for every j, $\tau_{(i)j}$ denotes the set of m_j ordered runtimes ($\leq B$) for all x_i, where duplicate values are removed. Each a_j in f_w corresponds to m_j sets $s_{(i)j}$, $i = 1, \ldots, m_j$ in Y, where each $s_{(i)j}$ contains all problems solved by a_j within $\tau_{(i)j}$, and $V = \sum_j m_j$. Moreover, $s_{(i)j}$ is associated with cost (i.e., runtime) $\tau_{(i)j}$. Note that, for $j_1 \neq j_2$, $s_{(i)j_1} \neq s_{(i)j_2}$even if they address the same problems, because they correspond to different algorithms.

Example 2.3. In Example 2.1, a_1 corresponds to sets $\{x_4\}_{j=1}$, $\{x_1, x_4\}_{j=1}$, and $\{x_1, x_3, x_4\}_{j=1}$ with respective costs 9, 11, and 16; a_2 corresponds to $\{x_1\}_{j=2}$, $\{x_1, x_3\}_{j=2}$, and $\{x_1,\ x_2,\ x_3\}_{j=2}$ with respective costs 8, 10, and 17; a_3 corresponds to sets $\{x_4\}_{j=3}$ and $\{x_1, x_4\}_{j=3}$ with respective costs 11 and 19; and a_4 corresponds to sets $\{x_4\}_{j=4}$ with cost 18. One possible solution to this weakly-optimal portfolio problem (and to the strongly-optimal version as well) is $\mathcal{S} = \{S_1 = \{\langle a_1, 9\rangle, S_2 = \langle a_2, 17\rangle\}$. Cast as a K-BMCP, this solution chooses $\{x_4\}_{j=1}$ for one budget and $\{x_1, x_2, x_3\}_{j=2}$ for the other. No cost exceeds the given budget of 20, and the overall weight sum of 3 is maximized.

Example 2.3 maps a weakly-optimal algorithm portfolio problem into a K-BMCP. In reverse, any K-BMCP is readily transformed into an algorithm portfolio problem on one processor, where each s_v corresponds to a unique solver that solves all problems in s_v with exact time t_v.

Theorem 2.4.1. *The weakly-optimal portfolio problem is NP-complete.*

Proof. If $K = 1$, K-BMCP is polynomial-time reducible to a non-parallel weakly-optimal portfolio problem on one processor. Because 1-BMCP is NP-complete [56], the weakly-optimal portfolio problem is NP-hard. In addition, the weakly-optimal portfolio problem is trivially NP, and therefore it is NP-complete. □

The solution of any strongly-optimal portfolio problem is also the solution to its corresponding weakly-optimal portfolio problem. Therefore we have

Theorem 2.4.2. *The strongly-optimal portfolio problem is NP-complete.*

Let binary variables I_v^k and X_i indicate, respectively, whether set s_v is selected for processor k ($I_v^k = 1$) or not ($I_v^k = 0$), and whether x_i is solved ($X_i = 1$) or not ($X_i = 0$). Then the K-budgeted maximum coverage problem (and weakly-optimal portfolio problem) can be formulated as the IP problem:

$$\max \sum_i w_i X_i \tag{2.9}$$

$$s.t. \quad \sum_v I_v^k t_v \leq B, \quad 1 \leq k \leq K \tag{2.10}$$

$$\sum_k \sum_{v: x_i \in s_v} I_v^k \geq X_i \tag{2.11}$$

$$I_v^k, X_i \in \{0, 1\} \tag{2.12}$$

Formula 2.10 specifies the budget (i.e., time-limit) constraint, and formula 2.11 requires that any selected x_i is contained in at least one chosen set. To solve the strongly-optimal problem, IP retains the constraints, but modifies the objective function 2.9 to

$$\frac{BK+1}{\min_i\{w_i\}}\sum_i w_i X_i + \sum_k (B - \sum_v I_v^k t_v) \tag{2.13}$$

As explained above, the solution of the IP problem is a portfolio about how to share an available time resource, not a schedule that consists of an ordered sequence of algorithm-duration tuples. In Section 2.5.2, this work develops and test two heuristics to convert an IP-produced sharing policy into a schedule. *Maximum-ratio-first* (MRF) prefers an algorithm-duration tuple with the largest ratio of the weighted number of problems solved by that algorithm to duration. *Maximum-weighted-number-first* (MWNF) prefers a tuple with the largest weighted number of problems solved by that algorithm within that duration.

2.5 Empirical results

Thus far, this chapter has discussed the theoretical performance of algorithm portfolio constructors on a neighbor set. To evaluate the performance of algorithm portfolios constructed by the proposed methods for practical CSP solution, this section compares the proposed constructors (RSR-WG and COM with converting heuristics) to several other constructors on problems from recent CSP solver competitions [6].

2.5.1 RSR-WG, RPCPHYDRA, RPWG, and an oracle

This section compares RSR-WG to two straightforward parallelization methods, RPCPHYDRA and RPWG (introduced in Section 2.3.2), adapted from their sequential versions. To

evaluate RSR-WG's difference from an ideally perfect constructor, this section also compares RSR-WG to an oracle.

Identification of neighbors is based on the Euclidean distance between pairs of 36 feature vectors. To extract the 36 features values (e.g., number of variables, maximum domain size) used by CPHydra and WG, this work ran the CSP solver Mistral 1.550 ([8]). For feature extraction, this work allotted 1 second on an 8 GB Mac Pro with a 2.93 GHz Quad-Core Intel Xeon processor.

The problem repository of CPAI'08 includes 3307 problems in 5 categories. Some solvers could not address problems in every category; this work merged the 2-ARY-INT and N-ARY-INT ($N > 2$) categories because the same solvers addressed both. Because our experiments count solved problems (those where a solver finds a solution or proves that none exists), this work excluded any problem that was not solved by any solver within the CPAI08 time limit of 1800 seconds. If CPHydra does not extract features quickly enough, it simply splits its schedule evenly among its three algorithms. Rather than test portfolios' luck with an algorithm this way (and penalize a portfolio with more algorithms at its disposal), this work chooses to exclude such problems. Table 2.2 summarizes the remaining 2865 problems in 4 categories.

Stratified partitioning was used in all runs, to maintain the proportions of problems from different categories in each subset. Table 2.3 reports the performance, in number of problems solved within 1800 seconds each, of an oracle and three non-parallel algorithm portfolio constructors as baselines: CPHydra_k_10, CPHydra_k_40, and GASS. (CPHydra chose 10 or 40 similar problems for its k_10 and k_40 versions, so here RPCPHydra selects 10*K neighbors, randomly distributes them to K processors, and executes a complete search for the optimal schedule on each processor.) The data for

Table 2.2: Competition problems by category in CPAI'08. Experiment problems were those for which at least one solver found a solution or showed that none existed, and also had features extractable within one second. Solvable problems had at least one solution.

Applicable solvers	**Category**	**Competition problems**	**Experiment problems**	**Solvable experiment problems**
17	GLOBAL	556	493	256
22	k-ARY-INT ($k \leq 2$)	1412	1303	739
23	2-ARY-EXT	635	620	301
24	N-ARY-EXT ($N > 2$)	704	449	156
Total	–	3307	2865	1452

GASS was obtained by 10-fold cross-validation with stratified partitioning on the 2865 problems.

All portfolio construction experiments ran under 10-fold cross-validation on a Dell PowerEdge 1850 cluster with one head node and 86 compute nodes, each with four Intel 2.80 GHz Woodcrest dual-core processors. RSR-WG results reported here are for portfolio construction (i.e., scheduling) time plus runtime. The runtimes of RPWG and RPCPHydra did not include portfolio construction time, which gave them a slight advantage. In extensive testing, uniform weighting and the three weight functions in Table 2.1 produced slightly different performance improvements in RSR-WG, but no one weight function statistically significantly outperformed the others consistently. Thus this work adopts the normalized-fixed weight function in Table 2.1.

RPCPHydra's portfolio construction time was limited to 180 seconds. If it did not

Table 2.3: Benchmark results for the 3rd International CSP solver competition. Experiment problems are from the fourth column of Table 2.2.

Solver	**Oracle**	**GASS**	**CPHydra_k_10**	**CPHydra_k_40**
Number solved	2865	2773	2577	2573
% solved	100	96.79	89.95	89.81

Table 2.4: Performance of 3 parallel portfolio constructors on 2865 problems. Best value for K processors is in boldface. * means RSR-WG outperformed RPWG; † means RSR-WG outperformed RPCPHYDRA.

K	RPCPHYDRA	Neighbor set ratio					
		0.005		0.01		0.02	
		RPWG	RSR-WG	RPWG	RSR-WG	RPWG	RSR-WG
1	2779	2771	2773	2778	2779	**2787**	2786
2	2807	2801	***2826**	2799	*2821	2802	*2823
3	2817	2808	*†**2841**	2810	*†2836	2808	*2839
4	2827	2810	*†**2850**	2812	*†2847	2811	*2847
5	2830	2817	*†**2855**	2819	*†2851	2816	*2852
6	2831	2821	*†**2857**	2818	*†2855	2819	*2856
7	2834	2823	*†**2858**	2823	*†**2858**	2824	*2857
8	2834	2825	*†2859	2825	*†**2860**	2825	*2858

produce the optimal schedule in that time, the best schedule found so far was used. To reduce search time, any algorithm dominated by another algorithm (i.e., always outperformed by it on all 2865 problems) was also eliminated from RPCPHYDRA's consideration. RPCPHYDRA also scaled all schedules to exploit the full time limit B.

Table 2.4 compares the performance of parallel portfolios from three constructors: RPCPHYDRA (the parallel version of CPHYDRA), RPWG (the naive parallel version of GASS), and RSR-WG with neighbor set ratios 0.005, 0.01, and 0.02. It lists the total number of problems (out of 2865) solved by each constructor's portfolios, and flags experiments where RSR-WG portfolios were statistically significantly better ($p < 0.005$) than those of a naive parallel constructor.

For RSR-WG this work simulated all 24 solvers from the original competition [6]. For RSR-WG only, this work tested as many as $K = 16$ processors. Both $K = 8$ and $K = 16$ produced near-oracle performance; indeed, 2 out of 10 runs for $K = 16$ were perfect. Execution of RSR-WG on $K = 16$ processors is a reasonable approach for modern computers, where it would produce portfolios able to solve only one fewer

problem than an oracle. (Execution of RSR-WG on one computer with multiple cores could degrade performance, for example, due to the overhead introduced by memory sharing.)

One important question is the number of training problems to use, as measured by the neighbor set ratio. For RSR-WG and K from 1 to 16, Table 2.5 reports on neighbor set ratios of 0.00125 to 0.16, which yield neighbor sets that range up to as many as 412 similar problems. Table 2.5 shows how many problems (out of 2865) RSR-WG solved, and shows how the neighbor set ratio impacts performance under different numbers of processors K. Clearly RSR-WG efficiently generates effective algorithm portfolios, and does best with smaller neighbor set ratios for $K > 1$ (i.e., for r between 0.0025 and 0.02). In contrast, RSR-WG does best with larger neighbor set ratios for $K = 1$. On $K = 1$, RSR-WG generates simple algorithm schedules, and it outperforms GASS, CPHydra_k_10, and CPHydra_k_40. (Refer to Table 2.3.)

Finally, Figure 2.5 compares the runtimes of an oracle solver and RSR-WG in one run with neighbor set ratio 0.005. (Again, RSR-WG's time includes both portfolio construction and search.) Each circle represents one of the 2865 problems. Those at the far right correspond to problems that went unsolved by RSR-WG in 1800 seconds. Those on the diagonal correspond to problems that were solved by RSR-WG as quickly as an oracle would have solved them. Clearly, more processors reduced the number of unsolved problems (from 90 to 6 in this particular run) and solved more problems as quickly as an oracle.

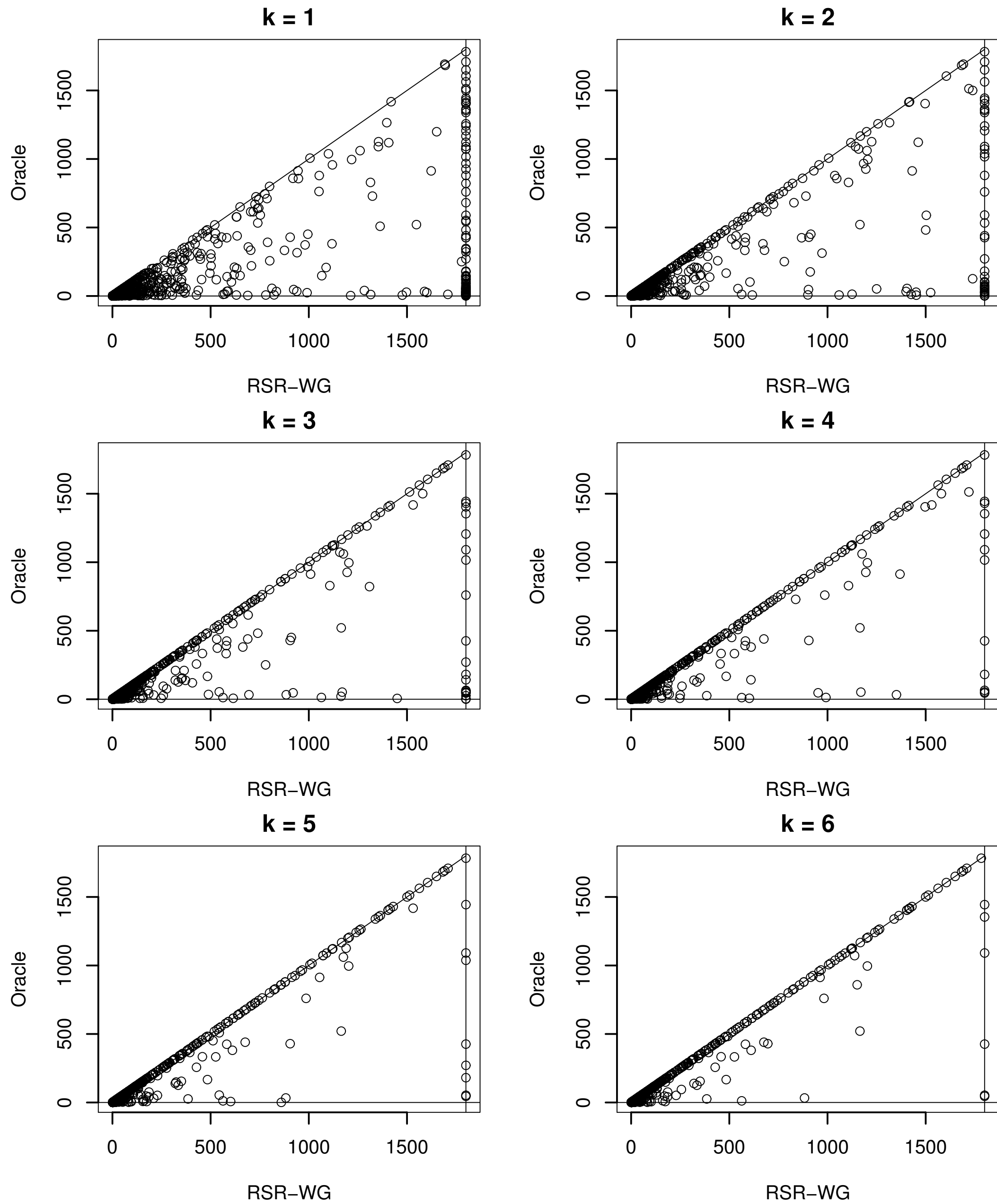


Figure 2.5: RSR-WG vs. Oracle. (Ideal) oracle runtime (y-axis) compared to RSR-WG time (x-axis) on k processors with neighbor set ratio $r = 0.005$. Each circle is a result on one of the 2865 problems.

Table 2.5: Mean and standard deviation for the number of problems solved by RSR-WG. Mean and standard deviation for the number of problems solved by RSR-WG out of 2865, with normalized-fixed weight function over 10 runs with K processors. Best value for K processors is in boldface.

		Neighbor set ratio							
K		0.00125	0.0025	0.005	0.01	0.02	0.04	0.08	0.16
1	μ	2757	2751	2773	2779	2786	**2789**	2788	**2789**
	σ	3.49	5.48	3.65	3.20	2.30	3.17	3.09	2.51
2	μ	2809	**2828**	2826	2821	2823	2816	2810	2809
	σ	2.41	2.90	3.51	2.49	3.16	2.97	2.99	2.87
3	μ	2834	**2844**	2841	2836	2839	2832	2827	2819
	σ	3.74	1.89	2.12	1.93	2.56	2.07	2.27	2.07
4	μ	2834	2848	**2850**	2847	2847	2843	2838	2832
	σ	4.21	1.34	2.15	1.57	2.63	2.06	2.22	2.50
5	μ	2846	2853	**2855**	2851	2852	2850	2845	2843
	σ	3.07	1.27	1.37	2.35	0.88	1.78	2.72	3.26
6	μ	2850	2856	**2857**	2855	2856	2853	2851	2850
	σ	3.84	1.17	0.95	1.07	1.26	1.64	1.03	1.07
7	μ	2853	2857	**2858**	**2858**	2857	2855	2854	2854
	σ	1.73	1.15	0.79	0.57	0.82	1.83	2.35	1.14
8	μ	2858	2859	2859	**2860**	2858	2858	2856	2855
	σ	2.67	1.90	1.18	1.34	1.06	1.18	0.74	1.43
16	μ	2860	2863	**2864**	**2864**	**2864**	2863	2861	2861
	σ	1.51	1.03	0.42	0.00	0.00	0.00	0.42	0.47

2.5.2 Comparison of RSR-WG to a complete constructor

Empirical results from the previous subsection show that RSR-WG is a promising approach to the construction of parallel algorithm schedules. This subsection explores the difference between RSR-WG's greedy approach and the complete method introduced in Section 2.4. Note the difference between an oracle scheduler and an IP-based complete scheduler. An oracle chooses for each testing problem the algorithm that solves it fastest to move toward optimal (likely an impractical assumption), while an IP-based complete scheduler optimizes the objective function f_w or f_s on a set of training problems similar to the testing problem, and thus can be suboptimal in practice.

This subsection compares RSR-WG to the IP-based complete constructor COM. The complete constructor converts each portfolio problem into an IP problem, formats it as a ZPL file, forwards that to SCIP 2.1.0 for solution, and finally transforms the SCIP output into a schedule. (SCIP is among the fastest non-commercial mixed-integer-programming solvers [57].) This work implements the two heuristics MRF and MWNF for COM, introduced in Section 2.4.

Table 2.6 reports the number of problems solved by RSR-WG and the complete constructor with two heuristics (COM+MRF and COM+MWNF), and neighbor sets of sizes 10, 50, and 100, respectively. The time to generate a schedule was included. For both versions of COM, this included the time to generate IP problems, solve them with SCIP, and transform their output. For each number of processors K, the first row provides the results with the return heuristic and the second row provides the results with scale. (Instead of returning to the first algorithm, *scale* allocates the remaining time resource on each processor to each algorithm in proportion to its runtimes on that processor in $\mathcal{S}$.) The time limit is again $B = 1800$ seconds.

Table 2.6 shows that, although COM+MRF and COM+MWNF together solved most problems more often than RSR-WG, neither alone could consistently outperform the other two for all K-N combinations. For both RSR-WG and COM, however, scale always performs at least as well as return for schedules on one processor, an advantage that diminishes as K increases.

To investigate the impact of granularity (i.e., to what extent to partition B into multiple pieces), the comparison was executed at two more levels of granularity. Tables 2.7 and 2.8 report on unit time intervals of length 10 seconds and 100 seconds, respectively. The results in Tables 2.6, 2.7, and 2.8 show that for the same time limit, as

Table 2.6: Problems solved by RSR-WG and COM when $B = 1800$. Best performance for each $K - N$ pair is in boldface.

K	$N = 10$ RSR-WG	$N = 10$ COM MRF	$N = 10$ COM MWNF	$N = 50$ RSR-WG	$N = 50$ COM MRF	$N = 50$ COM MWNF	$N = 100$ RSR-WG	$N = 100$ COM MRF	$N = 100$ COM MWNF
1	2763	2765	2771	2790	2789	2788	2782	2782	2780
	2810	2799	2797	2805	**2806**	2804	2791	**2795**	2793
2	2827	2826	2825	2819	2817	2808	2814	2816	2811
	2834	2833	2834	2830	2826	**2832**	2819	**2831**	2826
3	2838	**2843**	2838	2833	2834	2834	2832	2834	2832
	2843	**2843**	2840	2833	**2842**	2841	**2837**	**2837**	2830
4	2849	**2850**	2848	2846	2845	**2847**	2842	2840	2844
	2845	2847	**2850**	2840	2846	**2847**	2840	**2848**	2843

Table 2.7: Problems solved by RSR-WG and COM constructors when $B = 180$. Best performance for each $K - N$ pair is in boldface.

K	$N = 10$ RSR-WG	$N = 10$ COM MRF	$N = 10$ COM MWNF	$N = 50$ RSR-WG	$N = 50$ COM MRF	$N = 50$ COM MWNF	$N = 100$ RSR-WG	$N = 100$ COM MRF	$N = 100$ COM MWNF
1	2578	2594	2603	2588	2605	2619	2572	2600	2609
	2604	2626	**2628**	2589	2627	**2629**	2580	2599	**2610**
2	2656	2666	2652	2652	2655	2660	2644	2638	2634
	2663	**2667**	2665	**2663**	2660	2657	2644	**2655**	2640
3	2694	2680	2688	2672	2677	2674	2668	2673	2669
	2689	**2689**	2684	2675	**2688**	2683	2673	**2681**	2672
4	**2700**	2696	2692	2694	2694	2687	2686	2695	2696
	2696	2692	2693	2694	**2698**	2692	2687	**2701**	2692

unit time intervals becomes more coarse, the performance of both RSR-WG and COM degenerated.

As indicated above, the 1800-second runtime per problem for RSR-WG and COM in these experiments includes the time to extract features, construct the schedule, and to execute it. RSR-WG adopts a greedy approach that dramatically reduces its scheduling time but still generates effective portfolios. For example, over 10 runs the average scheduling time of RSR-WG for $K = 8$ processors ranged from 14.56 to 14.96 seconds (σ in [6.05, 6.35]) with normalized-fixed weights and a neighbor set ratio of 0.16. For $K = 1$

Table 2.8: Problems solved by RSR-WG and COM constructors when $B = 18$. Best performance for each $K - N$ pair is in boldface.

K	$N = 10$			$N = 50$			$N = 100$		
	RSR-WG	COM		RSR-WG	COM		RSR-WG	COM	
		MRF	MWNF		MRF	MWNF		MRF	MWNF
1	2136	2158	2152	2099	**2133**	**2133**	2088	2108	2094
	2128	2156	**2163**	2100	2130	2122	2099	2100	**2109**
2	2220	2232	2235	2196	**2218**	2181	**2182**	2148	2138
	2224	**2246**	2232	2196	2212	2182	2173	2144	2128
3	2260	2278	**2279**	2236	2221	2228	2218	2179	2166
	2254	2275	2273	2234	**2237**	2235	**2227**	2171	2172
4	2291	2294	2298	2259	**2263**	2252	**2248**	2167	2161
	2295	**2303**	2302	2259	2251	**2263**	2239	2153	2144

processor under the same conditions, average scheduling time ranged from 14.30 to 14.80 seconds (σ in [5.75, 6.15]). These are small but statistically significant differences. In contrast, RPCPHydra sometimes failed to compute an optimal schedule within 180 seconds (the maximum time allocated to it for scheduling). When $K = 1$, CPHydra failed to compute an optimal schedule 4.81% of the time. When $K > 1$, CPHydra must construct a schedule for each processor on training sets that may be considerably more diverse. This can increase the search effort; indeed, for $K = 8$, CPHydra failed to compute an optimal schedule 14.39% of the time. As for GASS, because it learns on all the training problems, it required more than 5 days of execution time for its single entry in Table 2.3.

Coarser granularity (indicated by a smaller B, which allocates longer intervals) impacts the scheduling efficiency of RSR-WG, but the effectiveness of the resultant portfolio depends on the performance matrix entries for the neighbors of the testing problem. A smaller B does not necessarily reduce the effectiveness of the resultant algorithm portfolio; if that were the case, a switching (or scheduling) portfolio would

always be superior to algorithm selection. In addition to Tables 2.6, 2.7, and 2.8, where B = 1800, 180, and 18, this work also tested RSR-WG with B = 20, 10, 5, 4, 3, 2, and 1. (This is equivalent to time allocations that, instead of 1 second on a processor, are 90, 180, 360, 450, 600, 900, or 1800 seconds. Note that $B = 1$ is equivalent to racing one algorithm on each processor to address a problem.) In these granularity experiments, for $K = 1$ the number of solved problems peaked at $B = 10$. For $1 < K \leq 8$, no coarser granularity ever showed a significant improvement; indeed, performance degraded slightly as B decreased. Both improvement on $K = 1$ and failure to improve when $K > 1$ were consistent across all neighbor set ratios reported here, with peaks at either $B = 5$ or $B = 10$ when $K = 1$.

2.6 MAMC, a case-based portfolio paradigm

The principal idea for portfolio-based methods is that multiple solvers can collaborate to outperform each individual. In general, this idea can be exploited in a variety of domains beyond constraint satisfaction. This section generalizes the reasoning mechanism of portfolio constructors to a new case-based reasoning paradigm $MAMC$ (Multi-Agent Multi-Case-based reasoning).

MAMC, outlined in Algorithm 2.3, assembles a set of cases similar to a new case, solicits the opinions of multiple agents on them, and then combines their output to predict for a new case. Unlike earlier work with multiple cases, which drew from multiple case bases for related tasks [58] or focused on the distribution of resources across multiple machines [59], MAMC reasons over multiple cases resident on the same computer and for the same task. MAMC can also be used to estimate the reliability of its output, an essential but rarely available property in bioinformatics.

Obviously, methods for algorithm portfolio construction lie within in the MAMC framewwork. The agent $\mathcal{A}$ under consideration is a set of algorithms. Portfolio construction, either parallel (e.g., RSR-WG, COM) or non-parallel (e.g., CPHydra), involves reasoning about a new CSP (i.e., new case e) from similar cases (i.e., C). The effectiveness of each specialization of MAMC relies heavily on the efficient combination of output from multiple agents, just as RSR-WG does. The remainder of this section introduces a specialization of MAMC and explores its application to protein-ligand docking.

Algorithm 2.3 MAMC

Input: new case e, case base C, agents $\mathcal{A}$, pairwise similarity metric s, number of reference cases q

Output: prediction or recommendation for e

1: Select a subset L of q cases in C most similar to e as measured by s
2: Predict or recommend on each case in L with $\mathcal{A}_j$ for all $a_j \in \mathcal{A}$
3: Combine output from all $a_j \in \mathcal{A}$ weighted by their performance on L as output for e

2.6.1 Protein-ligand docking

The central topic of rational drug design is protein-ligand interaction, where a small molecule (a *ligand*) binds to a specific position (e.g., an open cavity) in a protein [60]. *Protein-ligand docking* (*PLD*) evaluates a ligand's orientations and conformations (three-dimensional coordinates) when it is bound to a receptor. PLD seeks ligands with the strongest (i.e., minimal) total binding energy in a protein-ligand complex, but most PLD software predicts binding energy poorly. Thus, for reliability, conventional PLD meta-predictors use *consensus scoring*, which averages scores or takes a majority prediction from several predictors. Because it ignores similarities between examples, as well as domain-specific and example-specific information about its individual predictors,

consensus scoring is inaccurate when most of its component predictors are inaccurate, as shown below.

In a single docking run, virtual high-throughput screening predicts which of thousands of compounds should be tested in the laboratory [61, 62, 63]. Recent approaches have tried chaining [64] or bootstrapping with an ensemble based on a single function [65]. This work, however, first combines different PLD predictors based on case similarity plus information from and about individual predictors.

Here the agents $\mathcal{A}$ are three pre-existing PLD scoring functions: eHiTS[1], AutoDock Vina[2], and AutoDock[3]. Although all rely on some form of machine learning, each has its own conformational sampling, scoring, and feature-based representation. They often perform dramatically differently on the same data, with no consistent winner.

A case is the binding energy measured in the laboratory between a given *receptor* (a target in a protein) and a chemical compound. Each chemical compound is a potential ligand, represented by a boolean feature vector that reports the presence or absence of chemical properties (e.g., whether it is a hydrogen-bond donor, or whether its topological distance between two atoms lies in some specific range). These features are different from those used by the agents. The agents consider three-dimensional chemical conformation, but the cases describe physiochemical and topological properties derived from two-dimensional chemical structure. In a case base C, all cases address the same receptor. To describe a case, values for its standard chemical $footprint$ of 1024 boolean features were calculated offline with programs such as openbabel[4,5]. Given a case base

[1]http://www.simbiosys.ca/ehits/ehits_overview.html

[2]http://vina.scripps.edu/

[3]http://autodock.scripps.edu/

[4]http://openbabel.org/wiki/Main_Page

[5]The raw data on the chemical features and docking energy was provided by the author's collaborator Weiwei Han. The author was responsible for the algorithm design, data processing and analysis,

C, a new chemical e, chemical-similarity metric s, and performance matrix τ for the agents $\mathcal{A}$ on C, MAMC's task is to predict the binding energy between C's receptor and e.

2.6.2 Cases, similarity, and combination

This work tests MAMC on datasets from *DUD* (Directory of Useful Decoys), a set of benchmarks for virtual screening [36]. A *decoy* is a molecule similar to its ligand in its physical properties but dissimilar in its topology. DUD has multiple ligands for each receptor, and 36 decoys for each of its ligands. This work considers two receptors from DUD: `gpb` and `pdg`. All three agents perform relatively poorly on them. Moreover, eHiTS, is the worst of the three on `gpb` but the best on `pdg`. Together these receptors challenge MAMC to choose the most accurate predictor for each chemical.

The similarity metric s on example e and case $c_i \in C$ is defined by the Tanimoto coefficient, the ratio of the number of features present in their intersection to the number of features present in their union, where $N(c)$ is the number of 1's in c:

$$s(e, c_i) = \frac{N(e\&ci)}{N(e|c_i)} \tag{2.14}$$

Each predictor a_j was asked to calculate a score for each ligand and decoy in the dataset. This work eliminates the very few chemicals that did not receive three scores; this left 1901 chemicals for `gpb`, and a separate set of 5760 for `pdg`. The agents' incomparable scales, however, required a simple but robust rank-regression scoring mechanism to map raw scores uniformly to a normalized rank score that reflects only the preference of an agent for one case over another. For each agent $a_j \in A$, MAMC sorts the raw scores from a_j for all $c_i \in C$ in ascending order, replaces each score with its rank, and

and presentation of the results.

normalizes the ranks in [0,1]. The normalized rank, denoted by $p(c_i, a_j)$, predicts the score of a_j on c_i. Higher-ranked cases thereby receive lower scores, in line with the premise that lower binding energy is better.

MAMC represents the performance of $\mathcal{A}$ on C in a $|C| \times |A|$ performance matrix τ. To evaluate the performance $\tau(e, a_j)$ of $a_j \in A$ on e, this work uses the set of cases L similar to e, but weights more heavily those more similar to it:

$$\tau(e, a_j) = \sum_{c_i \in L} s(e, c_i)\tau_{ij} \tag{2.15}$$

Then, to predict on e with all of $\mathcal{A}$, this work takes the agent a_j with the highest $\tau(e, a_j)$ and combines its predicted scores on L with normalized rank, again weighting more similar cases more heavily:

$$p(e) = \sum_{c_i \in L} s(e, c_i)p(c_i, a_j) \tag{2.16}$$

Intuitively, a scoring function that accurately distinguishes ligand set G from decoy set Y (where $Y \cup G = C$) should predict lower scores for ligands and higher scores for decoys. In other words, agent a_j is more accurate on ligand g only if its prediction for g is generally lower than its predictions for Y, and it is more accurate on decoy y only if its prediction for y is generally higher than its predictions for G. The *performance score* of agent a_j on case c is thus

$$\tau(c, a_j) = \begin{cases} \dfrac{|\{y \in Y | p(y, a_j) > p(c, a_j)\}|}{|\{y \in Y\}|} & \text{if } c \in G \\ \\ \dfrac{|\{g \in G | p(g, a_j) < p(c, a_j)\}|}{|\{g \in G\}|} & \text{if } c \in Y \end{cases} \tag{2.17}$$

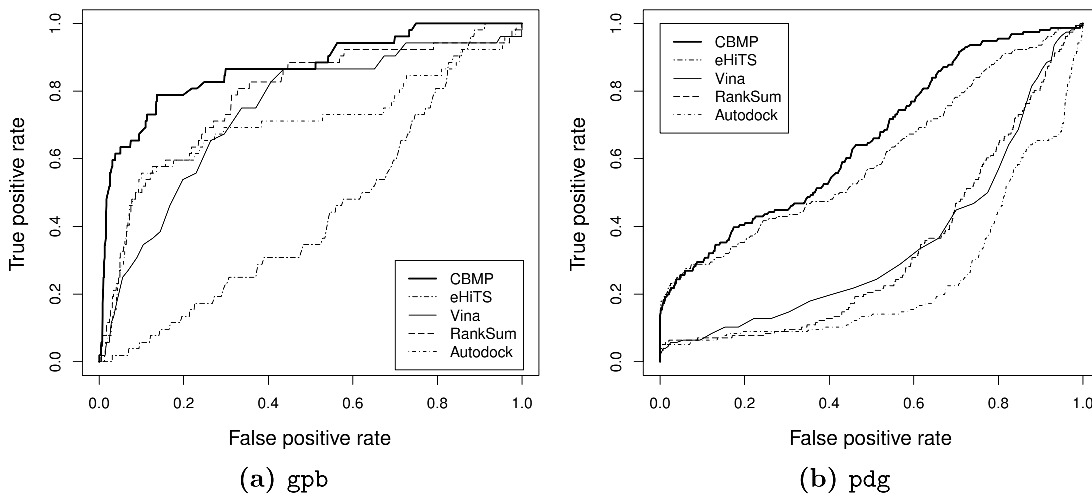
Scores in equation 2.17 lie in [0,1], where a higher value indicates better performance.

2.6.3 Experimental design and results

Each of these experiments predicts the binding energy of a chemical e to a receptor. This work examines the accuracy of five predictors: the three individual agents (eHiTS, AutoDock Vina, AutoDock) and two meta-predictors: MAMC and *RankSum*, a typical bioinformatics consensus-scoring meta-predictor. To predict the score on example e, RankSum adds the rank-regression scores from the three predictors, where a lower sum is better. In advance, for MAMC, this work first computed the similarities between all $\binom{n}{2}$ pairs of chemicals (about 1.8 million for `gpb` and 16.6 million for `pdg`), and recorded the five chemicals most similar to each chemical, along with their similarity scores. The experiments in this section ran on an 8 GB Mac Pro with a 2.93 GHz Quad-Core Intel Xeon processor, and analysis was with the R package ROCR.

First, this work evaluates the three individual predictors with leave-one-out validation: in turn, each of the n chemicals for a receptor served as the testing example e, while the other $n-1$ served as C. MAMC extracted the $|L|$ cases in C most similar to e, and then used equation 2.17 to gauge the accuracy of each individual predictor across all the cases in L. MAMC then chose the individual predictor with the best predictive accuracy on L and reported as a score the rank-regression score on e from that best individual predictor as in 2.15.

This work compares predictors' performance by their hit ratio across C. *ROC* (Receiver Operating Characteristic) curves illustrate the tradeoff between true positive and false positive rates, an important factor in the decision to test a likely ligand in the laboratory. (Classification accuracy alone would be less helpful, because the prevalence of so many decoys heavily biases the data sets. Simple prediction of every chemical



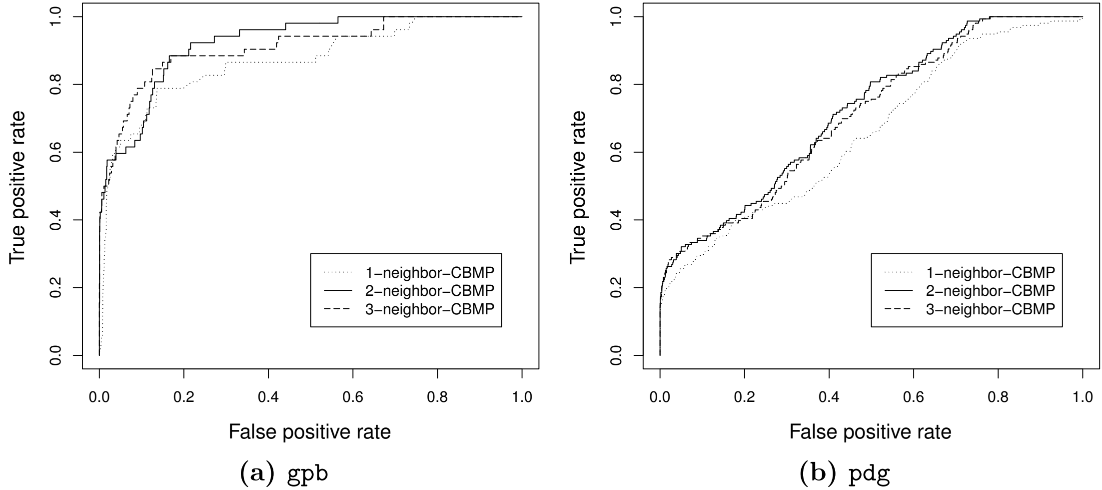
(a) gpb **(b)** pdg

Figure 2.6: ROC curves for PLD predictors on two receptors (a) gpb and (b) pdg

as a decoy would be highly accurate but target no chemicals for investigation as likely ligands.) A prediction $p(c)$ on any c produces true positives $C_1 = \{g \in G | p(g) \leq p(c)\}$ and false positives $C_2 = \{y \in Y | p(y) \leq p(c)\}$. Thus the true positive rate for c is $\frac{|C_1|}{|G|}$ and the false positive rate is $\frac{|C_2|}{|Y|}$.

For $|L| = 1$, MAMC uses only the single case c_i and need only reference τ_{ij} for each $a_j \in A$. The ROC curves in Figure 2.6 compare the performance of all five predictors on receptors gpb and pdg for $|L| = 1$, based on the predictors' scores and DUD's class labels. MAMC clearly outperforms the other predictors on both gpb and pdg. In particular, MAMC outperforms the best individual predictor eHiTS on pdg, even though the majority of its individual predictors perform poorly. In contrast, the performance of the consensus scorer RankSum on pdg was considerably worse; it requires accurate rankings from most of its constituent predictors for satisfactory performance, rankings the individual predictors could not provide.

To predict with larger $|L|$, each of the three scoring functions predicts for e with



(a) gpb

(b) pdg

Figure 2.7: ROC curves for MAMC with different $|L|$

equation 2.15, and then evaluates its performance on L with equation 2.17. MAMC then combines the predicted scores from all three predictors with equation 2.16, which allows it to consider the overall weighted performance of each predictor on a set of similar cases, and then take the weighted prediction of the agent with the best overall performance on those similar cases. Figure 2.7 shows a clear performance improvement for $|L| = 2$ on both receptors; for $|L| = 3$, this improvement is only marginal. Recall, however, that under leave-one-out validation, $|C| = 1900$ for `gpb`, and 5759 for `pdg`.

2.6.4 Discussion of PLD results

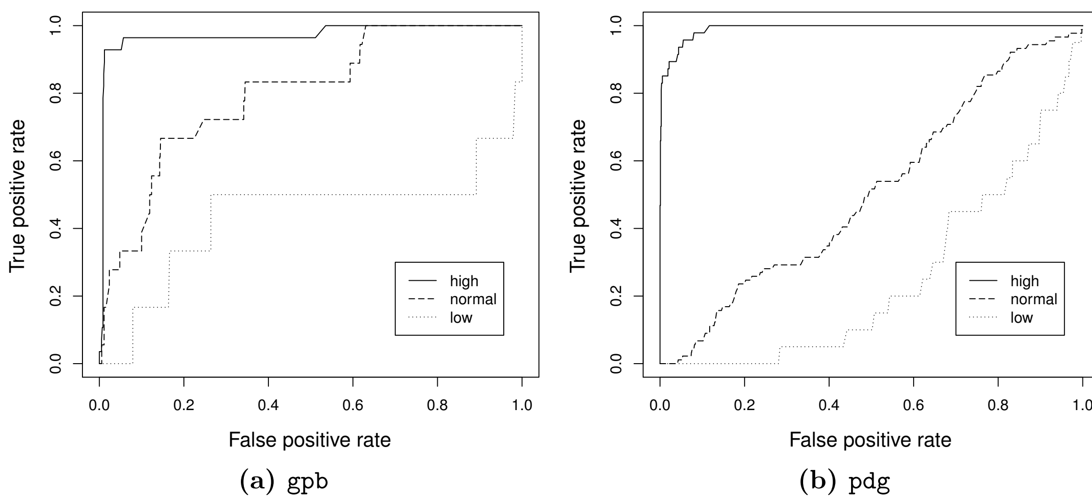
The nature of the data accounts for the difference between CSPs and PLD in an appropriate choice for $|L|$. The PLD data is inherently noisy and incomplete; the cases in C are only those that have been tested by a laboratory and made publicly available. There may be cases, for example, whose dismal performance dissuaded further testing of similar ones. For such a case there would be very few similar cases, so a larger L would provide little benefit. In contrast, the CSPs in competitions are typically submitted

by many competitors. A class of CSPs consists of problems that may vary somewhat in their size or anticipated difficulty but have some structural or modeling commonality. Each CPAI'08 problem class typically had dozens of problems. Even under 10-fold cross-validation, MAMC is likely to find more than a few similar CSP cases. Thus the determination of neighbor set size should be dependent on how likely MAMC is to find strongly similar cases.

Given a fixed size for L, MAMC's confidence about its results is still likely to vary from one case to the next. For example, as noted above, a particular case may have an L whose members are only slightly similar to it. Moreover, individual agents may perform poorly on some members of L. In both situations, MAMC should be less confident about its prediction on the original case. Intuitively, if MAMC could categorize individual cases by confidence level, it might improve its performance on the cases where its confidence level is high.

MAMC confidence analysis considers three kinds of predictions, demonstrated here on protein-ligand docking, where confidence before real-world laboratory testing is particularly important. Two cases c_i and c_j are said to be *similar* if and only if $s(c_i, c_j) > t1$ (here, 0.8), and *dissimilar* otherwise. A *reliable* predictor is one whose performance, as calculated by equation 2.15, is greater than t_2 (here, 0.9); otherwise it is *unreliable*. Together t_1 and t_2 define three categories of agent a_j's ability to decide on example e. A prediction has *high confidence* if e's closest neighbor c is similar to e and a_j is reliable on c. A prediction has *low confidence* if c is dissimilar to e and a_j is unreliable on c. In all other situations, a prediction has *normal confidence*.

Figure 2.8 isolates the performance of MAMC at these three confidence levels on PDL for `gpb` and `pdg`. For `gpb`, 31.77%, 49.50%, and 18.73% of the chemicals had



(a) gpb **(b)** pdg

Figure 2.8: ROC curves for confidence analysis on (a) `gpb` and (b) `pdg`

high, normal, and low confidence, respectively. For `pdg`, these percentages were 19.77%, 60.63%, and 19.60%. As expected, MAMC performed far better on the high-confidence chemicals for both receptors than it did on the full set. The benefit introduced by the confidence-based classification for `pdg` is particularly promising: although most candidate scoring functions had unreliable performance, confidence-based MAMC achieved almost perfect prediction on the high-confidence chemicals.

Might s alone have accurately predicted whether a chemical was a ligand? To investigate, this work ranks by similarity all pairs of cases that included at least one ligand: each pair is either a match (two ligands) or a non-match (a ligand and a decoy). Ideally, match pairs should have higher similarity scores than non-match pairs. In Figure 2.9, the ROC curve for each receptor is based only on s and whether or not a pair was a match. Although chemical similarity alone clearly distinguishes ligands from decoys in the DUD benchmark data set, it provides fewer likely ligands than MAMC, whose predictive performance is considerably better, especially when its confidence is

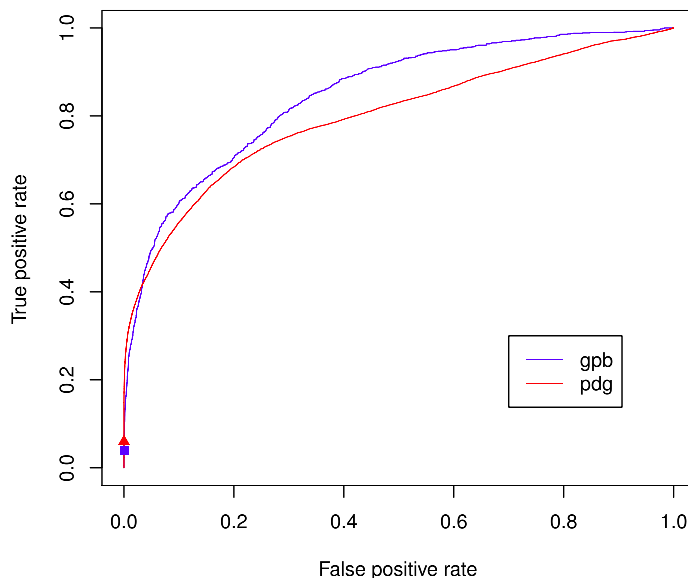


Figure 2.9: Performance of prediction directly from similarity. ROC curves for `gpb` and `pdg` based on computed similarity and match/non-match labels of chemical pairs. The marks at lower left correspond to the predictions based only on the minimum chemical similarity score 0.8.

high.

This chapter has introduced a greedy algorithm portfolio constructor RSR-WG and a complete algorithm portfolio constructor COM, and compared them with a variety of benchmarking parallelization approaches in intensive experiments. This chapter has also explored the performance of MAMC for compound virtual screening using PLD. In principle, MAMC can be applied to any bioinformatics or chemoinformatics problem, given a domain-specific similarity metric that compensates for individual predictors by its focus on additional relevant features. Moreover, random walk or information flow algorithms can improve such a metric in a case-similarity network.

In contrast to portfolio-based methods, where a CSP solver is treated as a black-box, the next chapter allows moderate modifications to a solver. These approaches do not interfere with the solver's search algorithm or its heuristics. Under this assumption, the

next chapter introduces a novel hybrid paradigm for adaptive parallel search.

3

SPREAD: a Hybrid Paradigm for Adaptive Parallel Search

Portfolio-based methods provide a simple solution for parallelizing pre-existing solvers without interference in their internal structure. Although portfolio-based methods can benefit from the leverage of different solvers, these methods have limited power to boost an individual solver in a parallel environment. Of course, a portfolio-based method can race a solver with different parameter configurations and random seeds, and racing may be quite promising for some CSPs, given the heavy-tail phenomenon [28, 29] and the recent results in parameter tuning [35, 41, 66]. Nonetheless, when it treats a solver as a black box, a portfolio-based method abandons the opportunity to parallelize constraint search by splitting and distributing the workload on a solver. Indeed, splitting-based methods offer another dimension for parallel search, and the benefits splitting introduces can be substantial, as demonstrated in this chapter.

This chapter introduces SPREAD, an adaptive paradigm that harnesses parallel computation to enhance an underlying solver. SPREAD combines portfolio-based and splitting-based methods to solve challenging CSPs in a general, simple, and effective way. The next section describes the partitioning method SPREAD uses to split CSPs.

Subsequent sections describe SPREAD's paradigm, and evaluate it in intensive experiments on a broad range of CSPs.

3.1 Iterative bisection partitioning

Search space splitting (SSS) can explore different search subspaces on different processor. For SAT problems, SSS usually relies on a guiding path (e.g., Figure 3.1(a)) [13]. Each node on a guiding path is either open or closed, conventionally indicated by a boolean flag δ_i. A *closed* node has attempted both domain values ($\delta_i = 0$, shown as red nodes in Figure 3.1(a)). When search backtracks to a closed node, it must backtrack further to some higher level of the binary search tree. In contrast, an *open* node has only one value under consideration ($\delta_i = 1$, shown as green nodes in Figure 3.1(a)). Although identification of a helpful guiding path is non-trivial (as in [10]), variables with particular properties have proved effective for splitting [12]. Iterative partitioning with clause learning, where search spaces of SAT subproblems may overlap, can also be an effective strategy [16].

Given their non-boolean domains and their variety of constraints, search space splitting for CSPs is more complex. Indeed, a SAT solver can conveniently partition its search space by adding new clauses, without any modification to its search strategy or propagation methods (as in [67]). A CSP solver that tries to add new constraints to split a search space, however, might confront constraints it could not directly process. For example, it is convenient to use parity constraints ($X_i + X_j \mod 2 = 0$) and ($X_i + X_j \mod 2 = 1$) to split a CSP into two subproblems, but a solver that supports only extensional constraints must be revised to process parity constraints. Even splitting only with already existing constraints (as in [68]) might have to contend with

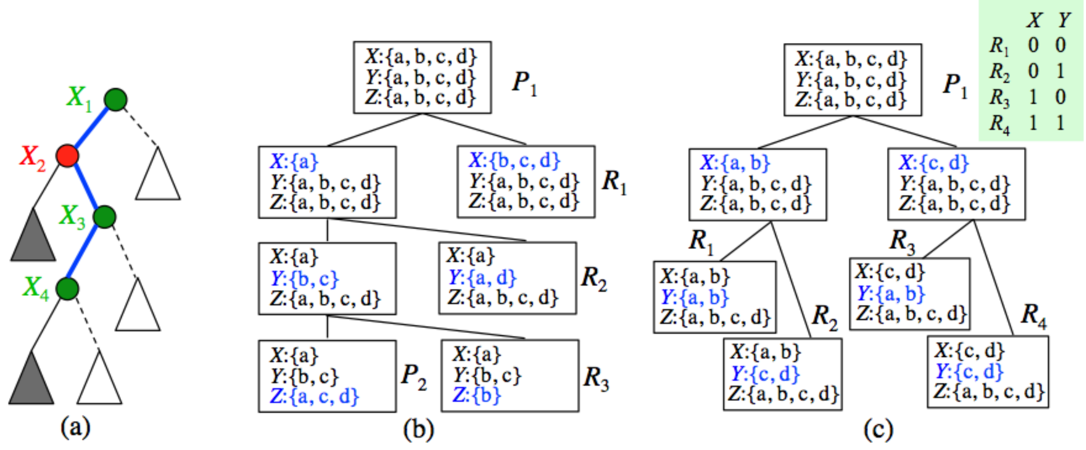


Figure 3.1: Three splitting methods. (a) A guiding path with open nodes at X_1, X_3, and X_4. (b) Extraction of subproblem P_2 from P_1 (under variable order X, Y, Z) produces subproblems R_1, R_2, R_3. (c) Iterative bisection partitioning on X and Y creates a virtual binary search tree of subproblems, shown with their bit-vector representations.

different formulations and different models for the same problem under different solvers.

Partitioning by domain manipulation avoids such difficulties. One approach, *network extraction* (*NE*), performs a sequence of domain splits on a subset of $\mathcal{X}$ under a given variable ordering for a single processor [69] (Figure 3.1(b)). A split on the ith variable produces subproblems P_i^1 and P_i^2 that differ only in the ith variable's domain: P_i^1 has some values for the ith variable, and P_i^2 has the others. NE was developed to avoid duplicate search on visited search spaces after restart [69].

Iterative bisection partitioning combines ideas from guiding path and network extraction to exploit the advantages of both. A *bisection partition* (*BP*) on variable X_i associated with domain D_i replaces X_i with two variables, X_i' and X_i'', whose respective domains D_i' and D_i'' partition D_i. To generate subproblems with search spaces that may have similar sizes, without bias toward particular domain values, we adopt an (almost) even bisection partition where $D_i' = \{v_1, ..., v_\chi\}$, $D_i'' = \{v_{\chi+1}, ..., v_{|D_i|}\}$, and $\chi = \lceil |D_i|/2 \rceil$. *Iterative bisection partitioning* (*IBP*) repeats BP on v ordered *splitting*

variables to generate 2^v subproblems. Figure 3.1(c) illustrates IBP on variables X and Y of P_1 to generate subproblems R_1, R_2, R_3, and R_4. Represented as bit vectors, these subproblems are convenient to describe, deliver, and reproduce. Moreover, if the workload for processing these subproblem were comparable, overall search performance on P_1 would be improved by processing them on different processors in parallel.

3.2 Spread

Spread (Search by Probing and REcursive Adaptive Domain-splitting) is an adaptive paradigm for parallel constraint search in an MPI environment. Spread uses a *manager-worker* framework, where a *manager* assigns tasks and coordinates messages among all the workers. Each *worker* executes the solver exactly as it would on a single processor, but may communicate with the manager about information necessary for parallelization. Workers do not communicate with one another.

Spread consists of two phases, a time-limited portfolio phase (detailed in Section 3.2.1) followed by a splitting phase (detailed in Section 3.2.2). In the *portfolio phase*, Spread's multiple workers search in parallel from random seeds; if any worker reports a solution or proves that there is none, the problem is solved. Otherwise, once the portfolio phase exhausts its time allocation, Spread begins its *splitting phase*, where the manager partitions the original problem into subproblems based on weights learned thus far. The manager distributes the subproblems to the workers with search limits based on the search effort during the portfolio phase. If any worker reports a solution, or if all the subproblems are proved to have no solution, the problem is solved. Any subproblem returned unsolved to the manager undergoes further partitioning (if the subproblem queue's capacity permits). Those new subproblems are enqueued and

eventually re-distributed with larger search limits as workers become available. This recursive partitioning mechanism naturally directs computational power to difficult subproblems.

In the static version of SPREAD (SPREAD-S), given problem P with search limit ℓ and restart schedule *policy*, the manager executes the portfolio phase with Algorithm 3.1 on workers under the control of Algorithm 3.2. The manager then uses Recursive Splitting with Iterative Bisection Partitioning (*RS-IBP*) during the splitting phase with Algorithm 3.3. As is traditional in MPI, for all algorithms in this chapter, the manager executes on processor 0, and the other K processors, indexed from 1 to K are workers. MPI.Send(*message*, i) sends *message* to processor i; MPI.Recv(*message*, i) receives *message* from processor i. Subsection 3.2.3 extends SPREAD-S to dynamic SPREAD (SPREAD-D).

3.2.1 Portfolio phase

In SPREAD, the manager uses different signals to guide the workers (1 = receive a subproblem, 0 = start portfolio phase, -1 = termination). To begin, the manager sends the initialization signal 0 to each of the K workers (Algorithm 3.1, lines 2-5). On receipt of that message, workers (under the direction of Algorithm 3.2) execute a simple algorithm portfolio (i.e., race with random seeds) where each attempts to solve P within limit ℓ. Any proof of either P's satisfiability or unsatisfiability within ℓ leads to an immediate report as well as to the termination of the MPI environment, including execution on all workers (Algorithm 3.2, lines 7-8). Otherwise, worker i has exhausted ℓ, and reports to the manager its learned weights w_l and search effort b_l (Algorithm 3.2, line 10).

Algorithm 3.1 Portfolio phase (Manager)

Input: CSP P, restart policy *policy*
Output: variable weights w, backtrack counts

```
1:  signal ← 0
2:  for i =1 to K do
3:      w_i ← 0, b_i ← 0
4:      MPI.Send(signal, i)
5:  end for
6:  while i > 0 do
7:      MPI.Receive(⟨w_p, b_p⟩, p)
8:      i ← i − 1
9:  end while
10: Compute w from all w_p and base from all b_p
11: return ⟨weights, base⟩
```

The manager receives w_p and b_p from worker p (Algorithm 3.1, line 7) from each worker. The manager records in w the average of the variable weights received from all the workers, and in *base* the average of the search effort received from them. Finally, the manager forwards w and *base* to the splitting phase, where they will be used to guide splitting and search on the subproblems (Algorithm 3.1, lines 10).

A portfolio-based method that shares information must address the trade-off between diversification and intensification [70]. Diversification uses dramatically different search strategies, and expects its searchers to proceed independently. In contrast, intensification explores with relatively small variations around a single strategy, and expects to share the information it gathers among all its searchers. Because SPREAD is intended to solve difficult CSPs, it does intensification in its portfolio phase, as recommended in [70]. Indeed, the primary purpose of SPREAD's portfolio phase is to glean information to support search space splitting, although it often solves easy CSPs quickly.

Algorithm 3.2 Worker

Input: CSP P, resource limit ℓ, restart policy *policy*
Output: search result on P

```
1:  while true do
2:    MPI.Receive(signal, 0)
3:    if signal = -1 then
4:      break
5:    end if
6:    if signal = 0 then {/* portfolio phase */}
7:      if solve(P, ℓ, policy, rand_seed) then
8:        Output result and MPI.Abort()
9:      else
10:       MPI.Send(⟨w_l, b_l⟩, 0)
11:     end if
12:   else {/* splitting phase */}
13:     MPI.Receive(⟨P^S,ℓ_r,weights⟩,0)
14:     Initialize variable weights of P^S
15:     if solve(P^S, ℓ_r, policy, rand_seed) then
16:       if P^S is satisfiable then
17:         Output solution and abort MPI
18:       else
19:         MPI.Send(⟨0, P^S⟩, 0)
20:       end if
21:     else
22:       MPI.Send(⟨1, P^S⟩, 0)
23:     end if
24:   end if
25: end while
```

3.2.2 Splitting phase of Spread-S

Figure 3.2 shows the task flow and information flow in the splitting phase of Spread-S (Algorithm 3.3). In its splitting phase, Spread-S recursively splits the search space with IBP. Initially, the manager partitions the original problem P into several subproblems based on the weight vector w from the portfolio phase. The manager then stores those subproblems in a subproblem queue Q, each represented as a bit vector for the partition that gave rise to it, and allocates search limit *base* (stored in $\mathcal{L}$) to each

subproblem (Algorithm 3.3, line 1-9).

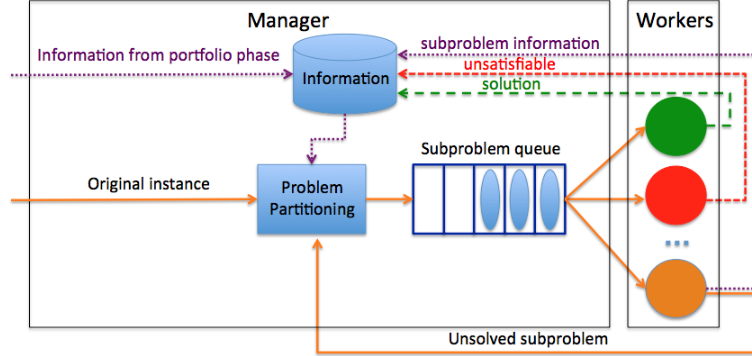


Figure 3.2: Splitting phase of SPREAD-S

When the initial subproblems are ready, the manager starts to distribute subproblems to the workers and receive feedback from them. Before it distributes subproblems together with search limits and variable weights to workers, the manager notifies the worker with signal 1 that it is about to do so. At the beginning of the splitting phase, all the workers are idle, so the manager dequeues and sends the first K subproblems with their corresponding variable weights and search limits from $\mathcal{L}$, and then awaits feedback (Algorithm 3.3, line 13-15). As in the portfolio phase, a worker immediately reports any detected solution to the manager, and terminates the MPI environment (Algorithm 3.2, line 17). If a worker proves its subproblem P^S unsatisfiable, however, it notifies the manager with message 0 (Algorithm 3.2, line 19). The manager immediately replies with a new subproblem from Q (together with its corresponding weights and search limit), if any is waiting (Algorithm 3.3, lines 20-21).

Otherwise, the worker has exhausted its resources ℓ_r and cannot solve this subproblem. The worker then returns the subproblem to the manager, marked as unsolved with flag 1 (Algorithm 3.2, line 22). On receiving a returned subproblem, if Q has fewer

Algorithm 3.3 RS-IBP (Manager)

Input: CSP P, variable weight vector w, initial limit *base*, restart policy *policy*
Output: search result on P

1: $v \leftarrow get_splitting_number(K)$
2: $threshold \leftarrow compute_threshold(v)$
3: $splitting_variables \leftarrow choose_splitting_variables(v, P, w)$
4: **for all** P^S in IBP(P, *splitting_variables*) **do** {/* generate initial subproblems */}
5: Q.push(P^S)
6: **end for**
7: **for** $i = 1$ to 2^v **do** {/* record search limit in L */}
8: L.push(*base*)
9: **end for**
10: $signal \leftarrow 1$ {/* indicates splitting phase */}
11: $\#enqueued \leftarrow Q.size()$
12: $\#feedback \leftarrow 0$
13: **for** $i = 1$ to K **do**
14: $distribute_subproblem(i)$
15: **end for**
16: **while** $\#feedback < \#enqueued$ **do** {/* iterates until P is solved */}
17: MPI.Receive($\langle feedback, P^S \rangle$, p)
18: $\#feedback \leftarrow \#feedback + 1$
19: **if** $feedback = 0$ **then** {/* subproblem is unsatisfiable */}
20: **if** !$Q.empty()$ **then**
21: $distribute_next_to(p)$ {/* feedback was from processor p }
22: **end if**
23: **else** {/* subproblem is unsolved */}
24: **if** $Q.size() < threshold$ **then** {/* if Q is too short, generate more */}
25: $\xi \leftarrow get_splitting_number(Q.size(), threshold)$
26: $splitting_variables \leftarrow choose_splitting_variables(\xi, P^S, w)$
27: **for all** P_i^S in IBP(P^S, *splitting_variables*) **do**
28: Q.push(P_i^S)
29: L.push($get_search_limit(P_i^S, base)$)
30: $\#enqueued \leftarrow \#enqueued + 1$
31: **end for**
32: **else** {/* otherwise, directly push the returned subproblem to Q */}
33: Q.push(P^S)
34: L.push($get_search_limit(P^S, base)$)
35: $\#enqueued \leftarrow \#enqueued + 1$
36: **end if**
37: **for** $i = 1$ to K **do**
38: **if** $isIdle(i)$ && !$Q.empty()$ **then**
39: $distribute_next_to(i)$
40: **end if**
41: **end for**
42: **end if**
43: **end while**
44: Send *signal* -1 to all processors for termination

than *threshold* subproblems, the manager partitions the returned subproblem on ξ new splitting variables, and enqueues the resultant subproblems along with their search limits (Algorithm 3.3, lines 24-31). Otherwise, there are still sufficient subproblems in the queue, and thus the manager re-enqueues the returned subproblem as it was, but with an increased search limit (Algorithm 3.3, lines 32-35). Whether or not it repartitions returned subproblems, the manager checks though all workers and distributes subproblems from Q to any idle worker (Algorithm 3.3, lines 37-41). When eventually distributed, an unresolved subproblem, even without repartitioning, will break ties with a random seed, and may therefore have a different search experience. SPREAD terminates when some worker finds a solution, or when all subproblems are proved unsatisfiable (and thus P is unsatisfiable).

SPREAD-S is complete, because IBP generates subproblems with mutually exclusive, collectively exhaustive search spaces, and a subproblem is always partitioned or receives larger search limits. Note that it would violate completeness if SPREAD-S were to distribute a returned subproblem without either re-partitioning it or increasing its search limit. Section 3.4 demonstrates that SPREAD is also effective.

SPREAD-S leaves open both the function *get_splitting_number* (Algorithm 3.3, line 25) and the parameter *threshold* (Algorithm 3.3, line 24). Arbitrary choices, however, could lead to Q of unbounded size. In this work, for K workers, the manager chooses $v = \lceil \log_2 2K \rceil$ initial splitting variables and computes the queue length $threshold = 2^v$. For each split on a returned subproblem, SPREAD-S here chooses the number of splitting variables ξ to be $\max\{\lceil \log_2(threshold - Q.size()) \rceil, 1\}$. These choices bound the size of Q by $2^v + 2^{v-1} - 1$, which can be achieved only when an unsolved subproblem returns and confronts a queue of length $2^{v-1} - 1$. Nonetheless, IBP's concise bit-vector

representation makes it space-efficient, and in practice allows large queues.

Spread-S always chooses for splitting variables those with the highest weights (Algorithm 3.3, lines 3, 26). This allows Spread-S to exploit the knowledge accumulated by multiple workers during the portfolio phase, and to avoid unpromising splits that could dramatically increase subproblem hardness and degenerate search performance. Note too that the learned weights are transmitted along with the subproblems, so that each worker can initialize its own variable weights to them before search on each subproblem. The plausibility of this design choice will be investigated in Section 3.4, which compares Spread to a parallelization approach that partitions on randomly chosen splitting variables.

3.2.3 Spread-D

Spread-S always uses the same splitting variables in the same order, determined by the weights first learned during its portfolio phase. Intuitively, for a returned subproblem, it could be more accurate to determine splitting variables dynamically, with weights learned during search on that subproblem. Spread-D is an extension of Spread-S that dynamically chooses its splitting variables. When a Spread-D worker fails to solve a subproblem within the allocated resource limit ℓ, it returns to the manager both the subproblem and the weights learned on it (i.e., received initially from the manager and modified during this search). This requires modification of Algorithm 3.2, line 22, where workers should also return weights of P^S, and of Algorithm 3.3, where the manager should receive those weights, and partition and distribute subproblem based on them. The manager then chooses additional splitting variables with the highest weights acquired during search on the returned subproblem (Algorithm 3.3, line 26). Because

Spread-D only adds splitting variables to a returned subproblem and never changes the previously designated ones, it guarantees mutually exclusive subproblems.

3.3 Experimental design

The experiments reported here evaluate multiple parallelization methods on their ability to solve both problems difficult for the underlying solver and problems difficult for all the solvers in the two most recent international CSP solver competitions [6, 7]. From the repository of more than 7000 problems in those competitions, we selected 51 representative classes that cover a broad variety of CSPs with relatively uniform population distribution, shown in Figure 3.3. To avoid any bias toward large classes, we stratified selection from each class to reflect any pre-specified subclasses and naming conventions, and chose a subset from each class in proportion to its original subclass sizes. This produced 1765 problems in classes with sizes from 7 to 65.

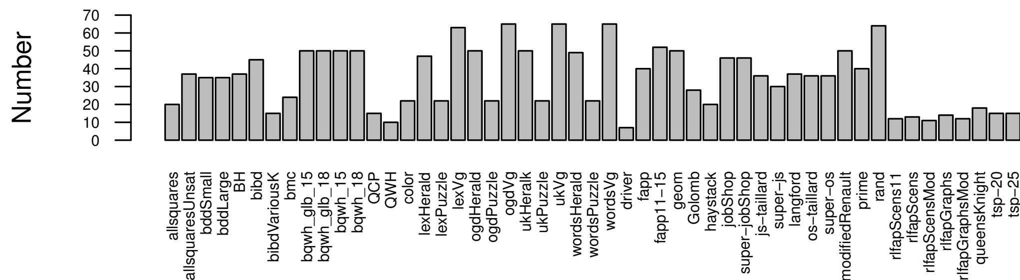


Figure 3.3: Problem distribution. Population distribution of the 1765 problems in the hard and harder problem sets, identified under stratified selection from 51 CSP competition classes.

The experimental platform was a Cray XE6m system with 160 dual-socket compute nodes. Each node contains two 8-core AMD Magny-Cours processors running at 2.3 GHz. (Here a Spread processor corresponds to a Cray core.) Without a readily-available parallel CSP solver as a benchmark, this paper compares the performance of

SPREAD-S and SPREAD-D to a variety of parallelization methods inspired by relevant work. We chose to parallelize the solver Mistral-1.331 (with C++ source code from [6]) because it is compatible with MPI on the Cray, and allows us to curate sets of difficult problems from recent CSP solver competitions and to evaluate the performance improvement under SPREAD-S and SPREAD-D. Appendix C describes Mistral's parameter configuration for all experiments in this chapter. Mistral can be compiled to run sequentially on the Cray XE6m either under the GCC compiler (*Mistral-GCC*) or the CC compiler (*Mistral-CC*), but Mistral-GCC runs about two to three times faster than Mistral-CC. This gives the Mistral-GCC benchmark a considerable advantage over all our parallel solvers, which require the CC compiler for MPI.

We solved each of the 1765 problems with Mistral-GCC, and eliminated the 1398 problems solved by Mistral-GCC in less than one minute. The 119 that could be solved by sequential Mistral-GCC within 1 to 30 minutes on the Cray became the *hard set*; the remaining 248 became the *harder set*. Finally, the *challenge set* consists of the 133 problems never solved by any solver within 30 minutes in either competition, and not already included in the hard set or the harder set.

We tested Mistral-GCC and Mistral-CC alone, as well as SPREAD-S and SPREAD-D with Mistral-CC. We also tested the following parallelization approaches:

- *Naive Random* (NR) races 63 copies of Mistral-CC with random seeds.
- *Parallel Portfolio* (PP) races 63 combinations of heuristics and restart policies. The heuristics were *impact*, *dom/wldeg*, *dom/wdeg*, and *impact/wdeg*. The restart policies were Luby-k (k (backtracks per unit) $\in \{128, 256, 512, 1024, 2048, 4096\}$), geometric (restart limit $x(n) = 100p^n$ at step n, where $p = 1.3$, 1.5

or 2.0), arithmetic ($x(n) = 16000n$, $8000n$, $1000n^2$, and $500n^2$), and dynamic. Dynamic adaptively determines whether to execute geometric restart with exponent 1.3, 1.5, or 2.0 based on the problem formulation, and restarts on the minimum of 1000 and the number of variables. Given these 4×16 possibilities but only 63 workers, PP did not execute *impact/wdeg* with dynamic restart and exponent 2.0.

- *Naive Variable* (*NV*) partially fixes the variable orders, as suggested in [67]. NV races 63 copies of Mistral, each of which randomly selects and orders the first 3 variables it assigns (but not their values) and reuses those variables on every restart.

- *Random Partitioning* (*RP*) splits on 7 randomly-chosen splitting variables, and enqueues those 128 subproblems for distribution to 63 workers, which run them to completion. Some workers process more than one subproblem.

- *No-Weight* SPREAD (NWSPREAD) is an ablated version of SPREAD intended to gauge the impact of learned weights. NWSPREAD does not use the weights from the portfolio phase for the workers, either to split or to search.

3.4 Experimental results

Unless otherwise stated, all results reported in this chapter use the median of the values from three runs (as in recent parallel SAT solver competitions [49]) on 64 processors, under a 30-minute per problem time limit, with the portfolio phase in both versions of SPREAD limited to 100 seconds. The imposed search limit is measured in number of backtracks. The initial backtrack limit was *base*, the average generated in the portfolio

phase (Algorithm 3.1, line 10). When a subproblem was partitioned on ξ additional splitting variables, this limit was multiplied by $(1.5)^{\xi}$.

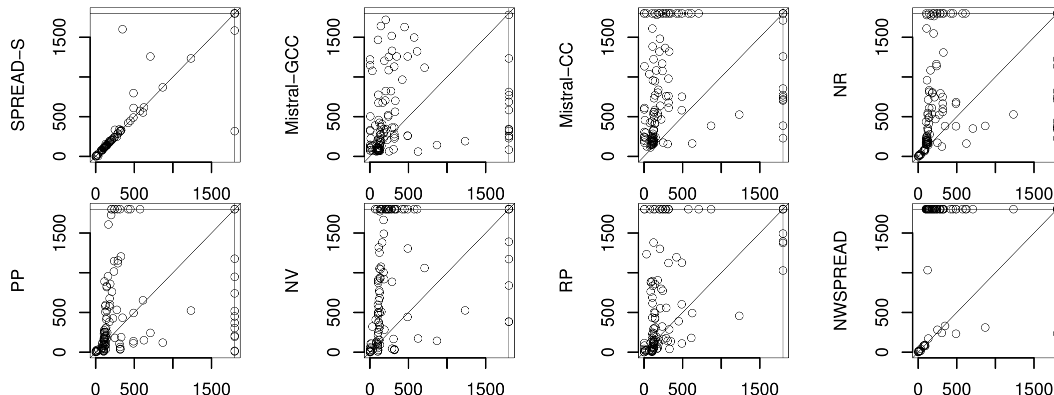


Figure 3.4: Comparison of Spread-D to benchmark methods. On the 119-problem hard set, solution time in seconds for Spread-D (x-axis) plotted against that for each of eight other methods (y-axis). Each circle represents a problem; black areas indicate a heavy concentration of problems. Circles at the top and far right represent unsolved problems.

On the hard set. Fig 3.4 compares Spread-D's runtime to that of the other approaches described above. Although a few instances (along the right margin) went unsolved under Spread-D, Fig 3.4 shows that both versions of Spread clearly outperform most of the other benchmark methods. Indeed, on the problems solved both by Spread-D and each competitor, Spread-D achieved significant average speedups compared to other benchmark approaches, as shown in Table 3.1.

Both Spread-S and Spread-D solved 43.70% of the hard set within 100 – 200 seconds. This is the time when both Spread versions have just begun to use critical

Table 3.1: Speedup of Spread-D over other approaches. Because speedup was calculated based on problems solved by both approaches for comparison, it does not indicate the performance of individual benchmark approaches.

	Mistral-GCC	Mistral-CC	NR	PP	NV	RP	NW
μ	19.08	27.91	2.65	1.98	4.03	3.34	1.59
σ	(79.41)	(148.32)	(2.77)	(1.92)	(3.89)	(6.48)	(1.61)

splitting variables, while PP tries a complementary algorithm portfolio instead. The plot for PP on the lower left in Figure 3.4 is a clear demonstration that search space splitting is essential. Moreover, search space splitting without the knowledge from the portfolio phase (NWSPREAD, on the lower right in Figure 3.4) was dramatically inferior; it could not solve 75.63% of these problems in 30 minutes, even though NR solved 17.65% of them within 100 seconds. Given their performance, NWSPREAD, sequential Mistral-CC, and Mistral-GCC were excluded from further comparisons.

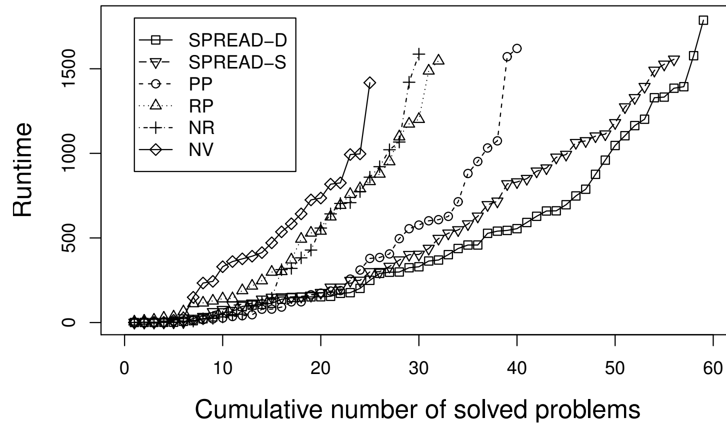


Figure 3.5: Comparison of the cumulative number of solved problems. On the 248 problems in the harder problem set, cumulative number of solved problems across 1800 seconds for SPREAD-D, SPREAD-S, and four competitors.

On the harder set. Figure 3.5 compares both versions of SPREAD to the remaining parallelization methods. Table 3.2 lists the number of problems solved by each algorithm. SPREAD-D solved 59 (46 satisfiable), 3 more than SPREAD-S. As one would expect, both versions of SPREAD behaved early on much like the portfolio-based methods NR and PP. SPREAD-S solved 10 (all satisfiable) within the first 100 seconds, its portfolio phase, while SPREAD-D solved 12 (all satisfiable). Both versions of SPREAD also solved more problems that required more time. In the last 800 seconds, SPREAD-S

Table 3.2: **Cumulative numbers of solved problems in Figure 3.5.**

Spread-D	Spread-S	PP	RP	NR	NV
59	56	40	32	30	25

solved 18 (11 satisfiable) and Spread-D solved 12 (6 satisfiable).

On the challenge set. Spread-S and Spread-D significantly outperformed the other parallelization methods on the challenge set. Table 3.3 compares runtimes for Spread-S, Spread-D, and RP on the 35 problems solved by at least one of them more than once. (The other approaches from Figure 3.5, NR, NV, and PP, solved only 1, 1, and 2 problems, respectively, all among these 35.) Spread-S and Spread-D are shown with both 10-second and 100-second portfolio phases; neither ever solved a problem during the portfolio phase. Although Spread did best with *rand* problems, it also solved problems in such categories as *Langford*, *crossword*, *super-jobshop*, and *graph-coloring*.

Spread's search is influenced by the variables it splits on and by their order, but the portfolio-phase search limit also has a strong effect. (Recall that the splitting-phase search limits are proportional to the backtracks consumed in the portfolio phase.) Because we report a median of three runs, to record a problem on any but the last line in Table 3.3, a program must have solved it at least twice. Both versions of Spread actually solved more problems; the last row indicates how many different problems they solved at least once in the three runs. Solved problems not listed in Table 3.3 include the satisfiable queenAttacking-8 and tdsp-C5-3-91, and the unsatisfiable pseudo-par-32-3-c, super-js-taillard-20-12, and super-js-taillard-20-22. Were the splitting-phase search limit infinite, Spread would partition only once and would probably benefit from a longer portfolio phase, but could readily be modified to search for all solutions.

Table 3.3: Challenge problem solution time in seconds for Spread-S (S-S), Spread-D (S-D), and the benchmark approach RP. Best value for each row is in boldface. 10 denotes a 10-second (rather than a 100-second) portfolio phase. '–' denotes failure to solve within the 30-minute time limit.

Problem	SAT	RP	S-S-10	S-S-100	S-D-10	S-D-100
costasArray-20	yes	**721.84**	–	–	876.13	1120.20
crossword-m1-words-21-10	yes	–	–	846.01	**746.36**	795.21
crossword-m1c-ogd-vg10-13_ext	no	–	744.89	**583.22**	748.62	750.28
crossword-m1c-ogd-vg10-14_ext	no	–	1302.41	**402.33**	1264.02	1280.17
crossword-m1c-ogd-vg12-12_ext	no	–	461.62	586.02	450.51	**450.22**
crossword-m1c-uk-vg11-12_ext	no	–	–	**1081.71**	–	–
frb53-24-2-mgd_ext	yes	1240.12	749.33	**329.85**	749.25	330.59
frb53-24-5_ext	yes	331.19	63.04	255.86	**62.74**	256.10
frb56-25-2-mgd_ext	yes	–	661.94	822.12	**661.32**	822.18
graphcoloring-myciel6-6	no	–	–	–	–	**1185.96**
graphcoloring-myciel7-6	no	–	–	–	–	**1178.54**
langford-2-14	no	485.81	187.39	401.30	**150.96**	–
langford-3-16	no	567.92	659.73	446.50	537.89	**129.86**
rand-3-24-24-76-632-17_ext	yes	358.80	240.02	326.82	240.18	**239.56**
rand-3-24-24-76-632-fcd-47_ext	yes	823.91	693.89	**207.30**	691.45	697.14
rand-3-24-24-76-632-fcd-50_ext	yes	692.31	59.71	168.40	59.63	**58.01**
rand-3-28-28-93-632-16_ext	yes	–	1551.52	–	1551.47	**1541.81**
rand-3-28-28-93-632-23_ext	yes	–	551.02	758.57	**550.41**	592.62
rand-3-28-28-93-632-25_ext	yes	–	448.20	464.58	**448.14**	449.93
rand-3-28-28-93-632-3_ext	yes	–	1306.23	**648.04**	1305.86	1304.37
rand-3-28-28-93-632-30_ext	yes	–	**893.93**	1061.22	894.57	897.32
rand-3-28-28-93-632-35_ext	no	–	**1186.84**	1321.97	1192.83	1189.95
rand-3-28-28-93-632-37_ext	yes	–	–	**238.10**	–	–
rand-3-28-28-93-632-8_ext	no	–	1126.08	–	1126.21	**1118.44**
rand-3-28-28-93-632-fcd-16_ext	yes	–	1531.64	**530.76**	1529.82	1519.74
rand-3-28-28-93-632-fcd-20_ext	yes	**24.79**	299.64	314.52	299.73	295.269
rand-3-28-28-93-632-fcd-21_ext	yes	–	1322.25	–	1322.01	**1221.21**
rand-3-28-28-93-632-fcd-24_ext	yes	–	1122.49	–	1116.40	**1116.19**
rand-3-28-28-93-632-fcd-27_ext	yes	–	–	**1349.44**	–	–
rand-3-28-28-93-632-fcd-31_ext	yes	–	700.22	**211.54**	690.09	684.804
rand-3-28-28-93-632-fcd-35_ext	yes	–	494.40	616.61	492.14	**489.88**
rand-3-28-28-93-632-fcd-40_ext	yes	–	**137.29**	219.16	137.32	197.24
rand-3-28-28-93-632-fcd-42_ext	yes	–	144.94	**124.55**	152.84	138.62
rand-3-28-28-93-632-fcd-46_ext	yes	1410.23	168.30	**159.21**	171.66	166.41
super-js-taillard-20-20	no	–	–	–	**1142.61**	–
Problems solved least twice	–	10	27	27	30	30
Problems solved at least once	–	20	31	30	33	32

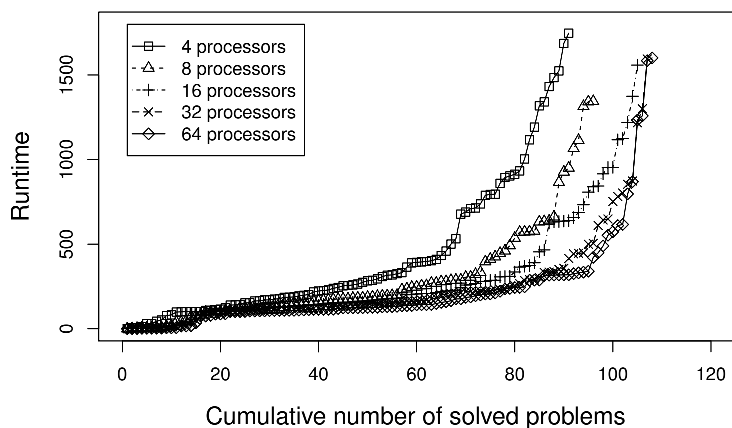


Figure 3.6: Cumulative number of problems from the hard set solved by SPREAD-S

Scalability. Figure 3.6 shows that, given more processors, SPREAD consistently solved more problems from the hard set, and solved the same number of problems more quickly. Under 128 processors, however, SPREAD introduced only marginal improvement on these problems. This is because SPREAD, like any other splitting-based parallelization method, cannot unboundedly improve its performance by using more processors, for two reasons. First, communication on more processors introduces heavier overhead. Second, splitting does not guarantee that partitioning will split the workload evenly, or even reduce the search effort.

Because we did not tune SPREAD specifically for Mistral, we would expect similar improvement with other CSP solvers. Recall that, among our curated problems, the hard set contains the easiest ones, where further improvement by SPREAD is relatively difficult. In contrast, Figure 3.7 shows how SPREAD scales on two typical problems from the harder problem set, given one hour. With more processors, SPREAD was significantly more likely to succeed within the time limit, and its runtime variance decreased, which produced more stable performance.

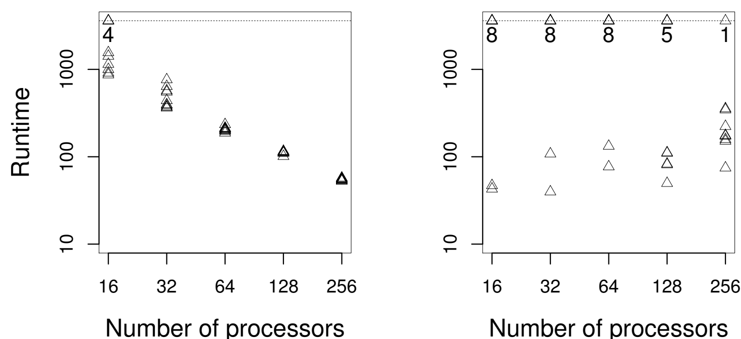


Figure 3.7: Scalability of SPREAD. Runtimes to solution within 1 hour across 10 runs of SPREAD-S with different numbers of processors for (a) rlfapScens11-f1 (unsatisfiable) (b) js-taillard-20-15-105-4 (satisfiable). Numbers at the top count runs that failed to solve the problem.

Other statistics. When a worker completes its subproblem but no subproblems remain in the queue, that worker becomes *idle*. To investigate how well SPREAD uses its computing resources, let the *idle ratio* of the ith worker be the fraction of overall runtime that it was idle. SPREAD-S's average idle ratio on problems solved during the splitting phase rose as high as 0.8251 on hard, 0.8793 on harder, and 0.5015 on challenge problems. Large idle ratios were likely caused by a high backtrack limit on an extremely unbalanced search tree, which forced most other workers to wait for a new assignment. The idle ratio could be improved by a backtrack limit tailored to a particular problem class. Overall, however, for experiments with 64 processors, SPREAD's idle ratio was under 0.1 on 56.92% of the hard, 69.92% of the harder, and 75.00% of the challenge problems. SPREAD-D's idle ratio was similar: 58.50%, 76.81%, and 75.00% under 0.1,

Table 3.4: Subproblem production by of SPREAD. During the splitting phase, mean split subproblems (#), average maximum subproblem queue length (Max), and average maximum number of splitting variables (μ).

	Hard			Harder			Challenge		
Implementation	#	Max	μ	#	Max	μ	#	Max	μ
SPREAD-S	156	129	7.6	518	136	13.6	244	132	9.6
SPREAD-D	146	129	7.5	374	137	11.3	211	130	8.6

respectively. Finally, Table 3.4 provides data on subproblems generated during the splitting phase.

This chapter has introduced SPREAD, a hybrid paradigm for general, simple, and effective parallelization of constraint search in MPI environment. SPREAD adopts two phases, a portfolio phase for accumulating information and potentially quick solution, and a splitting phase that uses RS-IBP for partitioning and balance workload. If SPREAD could better predict the difficulty of its subproblems, it might improve its workload balance, idle ratios, and search performance. The next chapter introduces a novel search effort estimation method. With that method, SPREAD can prioritize subproblems estimated to be the hardest, partition them more aggressively, and substantially improve its performance.

4

Search Effort Estimation to Boost Spread

To balance the workload among its workers, Spread exploits IBP to produce subproblems of comparable solution spaces, and executes RS-IBP to recursively partition on challenging subproblems. Although Spread is simple and effective, as Chapter 3 demonstrates, its controlling mechanism is somewhat passive. Indeed, Spread only processes subproblems in the same order as they were returned by workers. Moreover, it decides how many splitting variables to use without explicit consideration of the hardness of each subproblem it partitions. (Recall that RS-IBP, shown in Algorithm 3.3, computes ξ based only on $Q.size$ and *threshold*.) Spread should benefit from more intelligent control.

This chapter introduces Spread*, an enhanced version of Spread that controls Spread based on search effort estimation. Section 4.1 provides relevant background. Section 4.2 introduces a recursive estimator for the sizes of constraint search trees, and studies its theoretical and practical bias. Section 4.3 proposes a search effort estimator that exploits the recursive estimator and also learns. Section 4.4 elaborates on Spread* and compares its performance to Spread.

4.1 Search effort estimation

To order and partition subproblems better, a parallelization approach requires knowledge about search effort on them. Unfortunately, the real search effort on a subproblem is unknown until the subproblem is solved, and it is of no use to order or partition a subproblem after its solution. Therefore, SPREAD* estimates search effort on subproblems in advance.

Search effort estimation has long attracted significant research attention. It has many potential applications besides the parallelization of constraint search [71, 72, 73]. Methods devised to predict the search effort required by SAT problems and CSPs typically learn predictive models with a variety of machine learning methods such as Bayesian networks [43], linear-regression empirical hardness models [74], or latent class models [40]. Training such models, however, requires problem-specific domain knowledge to select features, extensive benchmarking experiments, and/or large sets of representative training examples. In addition, such methods are often inapplicable in a parallelization paradigm like SPREAD, where subproblems may undergo repeated partitioning and full problems are often too challenging to benchmark against sequential solvers at all.

In contrast, random sampling (*RS*) estimates search tree size without learning a predictive model [75]. A *sampling* on a (static) search tree is a random walk from the root to any leaf. RS estimates the search tree size by repeated sampling. Formally, let the depth of the root be 0, and let b_i denote the branching factor a sampling experienced at depth i. From a sampling that ends at a depth-d leaf, RS estimates the size N of

the search tree as

$$\hat{N} = 1 + b_1 + b_1 b_2 + \ldots + b_1 \cdots b_d \tag{4.1}$$

Given a search tree T, the expected search tree size $E[T]$ is the expected value of $\hat{N}$ over all possible samplings. *Stratified sampling* reduces the variance of RS [76], but both approaches assume a static search tree.

Consider the search tree T in Figure 4.1(a). From a sampling that visits nodes A, B, and E, RS has knowledge about T as shown in Figure 4.1(b). Based on its assumption of symmetry, RS anticipates T as shown in Figure 4.1(c), and therefore RS estimates $\hat{N} = 1{+}3{+}6 = 10$ by this sampling. Moreover, because a sampling branches at random, the probability of the sampling ABE is $\frac{1}{3} \times \frac{1}{2} = \frac{1}{6}$. By similar analysis of the remaining five possible samplings on T (i.e., ABF, ACG, ACH, ACI, and ADJ), it is clear that RS estimates $E[T] = 10$, which is indeed the size of T.

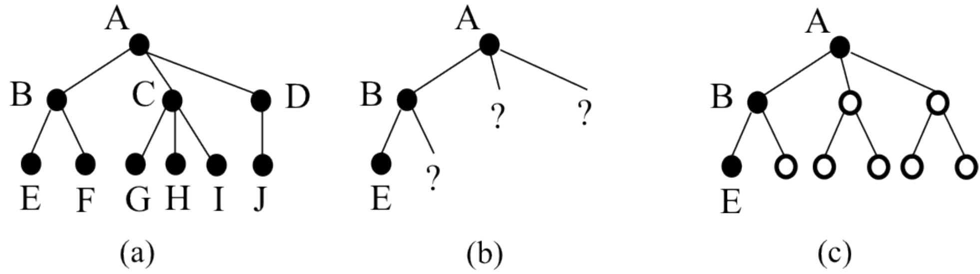


Figure 4.1: Random Sampling with RS. (a) A search tree T. (b) A sampling that visits nodes A, B, and E of T. (c) Search tree anticipated by RS based on sampling ABE.

A sampling estimator is *unbiased* if it guarantees $E[T] = N$ for any T. Although RS is unbiased, its straightforward application in constraint search is problematic. RS assumes a static search tree; it also assumes knowledge of branching factors for its estimation. Both assumptions are unlikely in constraint search.

More realistic estimation methods for constraint search are online estimators such as

weighted backtrack estimation (WBE) and *recursive estimation* (RE), which estimate the size of a binary search tree with chronological backtracking [71]. Given the multiset S_B of lengths of branches visited so far during search, and the probability $\Pr[d] = 2^{-d}$ that a sampling visits a branch of depth d, WBE estimates the search tree size by

$$\frac{\sum_{d \in S_B} \Pr[d](2^{d+1} - 1)}{\sum_{d \in S_B} \Pr[d]} \tag{4.2}$$

WBE can be viewed as a weighted version (by probability of sampling a particular branch) of RS with restricted sampling set (i.e., the set of branches visited so far during search). In contrast, RE estimates search tree size recursively. When search in a binary search tree from left to right reaches a leaf, RE recursively estimates the size of tree rooted at node v as

$$estimate(v) = 1 + estimate(leftChild(v)) + \mathcal{F}(rightChild(v)) \tag{4.3}$$

To calculate the size of the full tree, RE first calls equation 4.3 on the root, and then recursively. If v's right child, $rightChild(v)$, has been expanded, then $\mathcal{F}$ calls equation 4.3 on it; otherwise, $\mathcal{F}$ estimates the size of the subtree rooted at $rightChild(v)$ based on the (estimated) size of the subtree rooted at $leftChild(v)$. A leaf has estimate 1. On the root of a fully expanded subtree, equation 4.3 is exact.

As discussed in Section 1.3, constraint search can branch in different ways. (See Figure 1.3.) This work focus on 2-way branching and represents a search tree in the compact representation format of Figure 1.3 (c). This representation tolerates nodes with children that result from assignments to different variables, which is impossible under k-way branching. Moreover, unlike conventional representation of 2-way branching, this representation also allows branching factors greater than two, as in Figure 1.3 (b).

4.2 GRE, a recursive estimator

This section introduces *generalized recursive estimator* (*GRE*), an estimator that extends RE beyond binary trees to the systematic backtracking search trees created by a CSP solver. This section analyzes GRE's bias probabilistically. The more extreme examples show how estimation can be misled by search in a tree with an exaggerated shape. Although this section proves that on static search trees GRE is unbounded, the next section devises a way to harness it for dynamic search trees, where tree shape often improves under learned heuristics.

4.2.1 The GRE algorithm

This subsection first defines some symbols for subsequent use. Without loss of generality, assume that search orders the subtrees of a node from left to right in the same order that those subtrees are explored. When search reaches a leaf, the path to it from the root is the rightmost search frontier. For example, when search reaches leaf H of the search tree in Figure 4.2(a), the rightmost search frontier is ACH. The children of any internal node π on this path (here, A and C) are partitioned into two sets: $\Gamma_V(\pi)$, the children of π to the left of or on the path, and $\Gamma_U(\pi)$ the (implicit) children of π to the

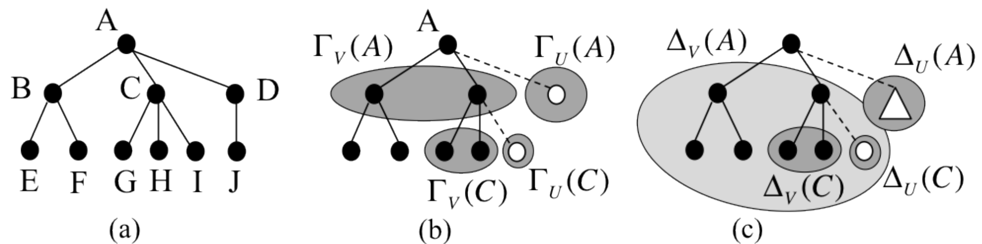


Figure 4.2: Categorization of nodes of a search tree. (a) A search tree with search frontier ACH. (b) Categorization of the children of internal nodes on the search frontier. (c) Categorization of search tree nodes based on categories from (b).

Algorithm 4.1 est(π)

```
1: if isLeaf(π) then
2:    return 1
3: else
4:    for all v in Γ_V(π) do
5:       e_v = est(v)
6:    end for
7:    return 1 + (|Γ_V(π)|+|Γ_U(π)|)/|Γ_V(π)| Σ_{v∈Γ_V(π)} e_v
8: end if
```

right of the path. Let $\Delta_V(\pi)$ (resp. $\Delta_U(\pi)$) denote the nodes in $\Gamma_V(\pi)$ (resp. $\Gamma_U(\pi)$) and all their descendants. Obviously, some nodes in $\Delta_V(\pi)$ may have not been visited (e.g., node I in $\Delta_V(A)$), and all nodes in $\Delta_U(\pi)$ are as yet unexplored.

Like the estimator in equation 4.3, when search arrives at a leaf, GRE recursively estimates the search-tree size, initially from the root:

$$est(\pi) = 1 + \sum_{v \in \Gamma_V(\pi)} est(v) + \sum_{v \in \Gamma_U(\pi)} f(v) \tag{4.4}$$

For simplicity, GRE in this work assumes a symmetric tree and expects $\frac{|\Delta_U(\pi)|}{|\Delta_V(\pi)|} = \frac{|\Gamma_U(\pi)|}{|\Gamma_V(\pi)|}$. Specifically, GRE calls the tree-size estimator shown in Algorithm 4.1. (A different $\mathcal{F}$, however, could characterize domain-specific search-tree shapes and lead to more accurate estimation.)

Example 4.1 To estimate the size of the search tree in Figure 4.2 after search reaches H, GRE recursively calls est(A), est(C), and est(H). According to equation 4.4, $|\Delta_U(C)| = 1$ and $|\Delta_U(A)| = 3.5$, so GRE estimates the full search tree size as 11.5.

Another important design decision for GRE is how to estimate the branching factor b for a node π. For k-way branching, it is natural to estimate b as the dynamic domain size $|D'_i|$ of the variable X_i chosen to expand π. For 2-way branching, since $\Gamma(\pi)$ may have children that correspond to assignments for different variables, GRE estimates b as

the sum of the dynamic domain sizes of all variables assigned a value at least once during the search to the most recently expanded child in $\Gamma_V(\pi)$. Without further information, this is the best GRE can anticipate, since it has no knowledge of $\Gamma_U(\pi)$.

4.2.2 Properties of GRE

Like RS, this work temporarily assumes a static search tree T for the analysis of GRE, and calculates an estimate, $E[T]$, for the real size N of T by sampling. To emphasize the fact that a sampling in GRE can experience multiple failures, this work probes instead of sampling. Formally, a *probe* P is a systematic backtracking with random value ordering under a cutoff c that limits the number of failures it tolerates. Note that the left-to-right branching order used above was an intuitive introduction; in practice, a probe actually explores branches systematically and depth-first, in arbitrary order. We can, however, freely reorder subtrees in T without changing N. Let $est(T,P)$ denote the estimated size of T based on probe P. Under cutoff c,

$$E[T] = \sum_{P} \Pr[P\ occurs] \cdot est(T, P) \tag{4.5}$$

Example 4.2 Consider the binary search tree T of size 6 in Figure 4.3 (a). There are 4 possible probes with cutoff $c = 2$: ABCBD, ABDBC, AEFEABC, and AEFEABD. Each probe occurs with probability $\frac{1}{4}$. Figures 4.3(b) and (c) illustrate the estimated search trees for probes ABCBD and AEFEABC, respectively, with symmetrically anticipated nodes in white. (Estimates for ABDBC and AEFEABD are similar.) GRE thus estimates $E[T] = (7 + 7 + 6 + 6) \cdot \frac{1}{4} = 6.5$, an overestimate of $N = 6$.

Example 4.3 With cutoff $c = 2$, GRE underestimates the tree in Figure 4.3(d) as 7.5, smaller than the true size $N = 8$.

RS is unbiased, as proved in [75]. Examples 4.2 and 4.3 show, however, that GRE

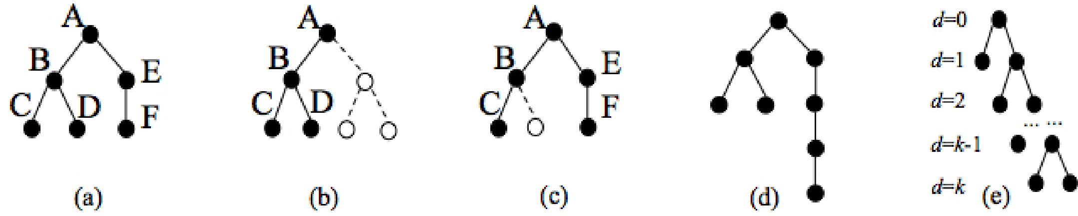


Figure 4.3: Search trees and their size estimation

loses this property when it uses a probe with $c > 1$. To examine GRE's bias more closely, this work defines an estimator as *T-unbiased* if and only if $E[T] = N$ for any tree T given some cutoff c. An estimator is *c-unbiased* if and only if $E[T] = N$ for any c on some T (or on some class of search trees). Moreover, an estimator is *unbiased* if and only if it is c-unbiased for any T, or T-unbiased for any c; otherwise, it is *biased*.

According to these definitions, GRE is T-unbiased for $c = 1$, where it degenerates to RS [75]. As an immediate result from Examples 4.2 or 4.3, GRE is biased. In addition, it is trivial that GRE is T-unbiased for c at least as large as the number of leaves in T (where it is exact), and GRE is unbiased on any perfectly-symmetric k-ary search tree.

An estimator has an *upper* (resp. *lower*) bound on a class Ω of trees if and only if there exists a constant $B > 0$ such that $\frac{E[T]}{N} \leq B$ (resp. $\geq B$) for any c and for any $T \in \Omega$. This work proves that, without any restrictions on search tree depth or width, GRE's bias is unbounded. The discussion is based on probes ($c = 2$) on the search tree T shown in Figure 4.3(e). Let an $\langle i, j \rangle$ probe be one with first failure at depth i and second failure at j. For systematic backtracking, $j \geq i$ always holds.

Lemma 4.2.2.1. *The probability p(i,j) of an* $\langle i, j \rangle$ *probe occurring on* T *in Figure 4.3(e) is* 2^{-j}.

Proof. At each level, a probe chooses either the left or the right branch at random,

and therefore with probability 2^{-i} it reaches the first failure at depth i. Then, search backtracks to a higher level and chooses the other branch. Because there are $j-i$ levels between depth i and j, with probability 2^{i-j} search reaches a failure at depth j, given the condition that it has reached the failure at depth i. Therefore, the probability that an $\langle i,j\rangle$ probe occurs on T is $2^{-i}\cdot 2^{i-j}=2^{-j}$ □

Lemma 4.2.2.2. *The estimated search tree size est(i,j) of an $\langle i,j\rangle$ probe occurring on T in Figure 4.3(e) is 2^i+2^j-1.*

Proof. Case (1) If $i=1$, the estimated search tree size is $|root| + est(left(root)) + est(right(root)) = 1+1+2^j-1 = 2^j+1$.

Case (2) If $i>1$, then consider the subtree T^{i-1} rooted at the parent node of the leaf at depth i. Since the depth-i leaf is the left child of the root of T^{i-1}, this situation obviously belongs to case (1) and T^{i-1} has estimated subtree size $2^{i-(i-1)}+2^{(j-(i-1))}-1$ $=2+2^{j-i+1}-1=2^{j-i+1}+1$. This is covered by Case (1), where the depth of the subtree is $j-i+1$. By GRE's symmetric assumption, all 2^{i-1} subtrees rooted at depth-$(i-1)$ nodes have the same size. Therefore, the estimated search tree size is the sum of sizes of all depth-$(i-1)$ subtrees and the size of the top of T above those subtrees, that is, $(2^{j-i+1}+1)2^{i-1}+2^{i-1}-1=2^j+2^i-1$. □

Theorem 4.2.2.1. *GRE has no upper bound.*

Proof. For $c=2$, there are $k-i+1$ possible probes for $i\le k-1$ and two for $i=k$, so there are $\frac{k(k+1)}{2}+1$ probes in all. By Lemma 4.2.2.1, the probability $p(i,j)$ of an $\langle i,j\rangle$ probe is 2^{-j}. Moreover, Lemma 4.2.2.2 shows that the estimated search tree size

$est(i,j)$ for an $\langle i,j \rangle$ probe is $2^i + 2^j - 1$, so by equation 4.5 the tree-size estimate is

$$E[T] = \sum_{\langle i,j \rangle} p(i,j) est(i,j) = \sum_{\langle i,j \rangle} (2^{i-j} + 1 - 2^{-j}) \geq \sum_{\langle i,j \rangle} 1 = \frac{k(k+1)}{2} + 1 \qquad (4.6)$$

Clearly, since $N = 2k+1$, $\frac{E[T]}{N} \to \infty$ as $k \to \infty$. □

Although Theorem 4.2.2.1 may not be of great interest as a search tree size estimator, restrictions on the shape of T can further clarify the bias of GRE. Consider, for example, the class $\Omega^{F,2}$ of full binary search trees, where each node has either two children or none, a class that has attracted broad interest [71]. This work next proves that GRE never underestimates the search tree size on $\Omega^{F,2}$.

Two search trees are *probe-equivalent* if and only if one can be transformed into the other by a sequence of reorderings of its subtrees. A binary tree is *left-dominant* if and only if $size(T_L) \geq size(T_R)$ hold for each of its nodes, where T_L and T_R respectively denote the left and right subtrees of the corresponding node. Figures 4.4(a) and (b) show a non-left-dominant full binary tree and a left-dominant tree, respectively. These two trees are probe-equivalent, since they can be transformed from one into the other simply by a sequence of swaps of left and right subtrees.

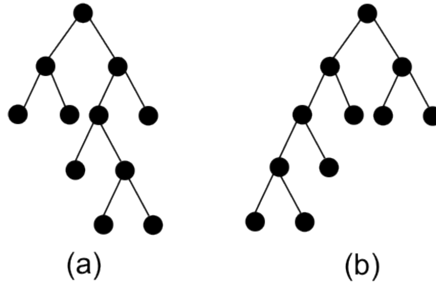


Figure 4.4: Two probe-equivalent full binary trees. (a) A non-left-bias full binary tree and (b) its left-bias probe-equivalence.

Theorem 4.2.2.2. *GRE has lower bound $B = 1$ on $\Omega^{F,2}$.*

Proof. Since $E[T]$ is calculated over all possible probes, it is invariant under any reordering of its subtrees. Because every non-left-dominant tree in $\Omega^{F,2}$ has a left-dominant probe-equivalent, which can be obtained by recursive checking and any required swapping of T_L and T_R at each node, it is adequate to prove this theorem only for left-dominant trees in $\Omega^{F,2}$. For proof by induction, Figure 4.5 establishes the base case for depth $k \leq 3$, where clearly $E[T] \geq N$ for any $1 \leq c \leq N$.

Figure 4.5: Left-dominant, full binary search trees for $k \leq 3$

Consider a left-dominant tree T of depth $k+1$ in $\Omega^{F,2}$. By the definition of left-dominant, the left subtree T_L and the right subtree T_R of the root of T are also left-dominant with depth $\leq k$. Let e denote the event that a probe visits T_L first, and $\bar{e}$ denote the event that a probe visits T_R first. By probability decomposition, $E[T] = \Pr[e]E[T|e] + \Pr[\bar{e}]E[T|\bar{e}] = 0.5 \cdot E[T|e] + 0.5 \cdot E[T|\bar{e}]$. For any cutoff c, there are three cases: (1) $c \leq size(T_L)$ and $c \leq size(T_R)$, (2) $c \leq size(T_L)$ and $c > size(T_R)$, (3) $c > size(T_L)$ and $c > size(T_R)$.

For case (1), probes that begins with T_L have $E[T|e] = 2 \cdot E[T_L] + 1$, because they will never visit T_R. It is clear that $E[T_L] \geq size(T_L)$, since the trees T_L and T_R have depth $\leq k$. Based on a similar argument, $E[T|\bar{e}] = 2 \cdot E[T_R] + 1$, where $E[T_R] \geq size(T_R)$. Therefore, $E[T] \geq size(T_L) + size(T_R) + 1 = N$.

For case (2), probes that begins with T_L are the same as case (1). If $\bar{e}$ occurs,

however, the probe will eventually exhaust T_R and probe T_L but with a smaller cutoff, $c' = c - size(T_R)$. For any admissible $c \leq size(T)$, $c' = c - size(T_R) \leq size(T_L)$, and $E[T_L; c'] \geq size(T_L)$, where $E[T_L; c']$ is the estimated size of T_L under probe limit c'. Clearly, $E[T] = 0.5 \cdot (2 \cdot E[T_L] + 1) + 0.5 \cdot (E[T_L; c'] + size(T_R) + 1) \geq N$.

Finally, for case (3), whether e or $\bar{e}$ occurs, it is equivalent to the situation where $\bar{e}$ occurs in case 2, and once again $E[T] \geq N$. □

Given its bias, the use of GRE under $c > 1$ to estimate search-tree size would not appear promising. The preceding analysis, however, assumed that the search tree was static. Modern CSP solvers usually employ a complex combination of search algorithms, variable-ordering and value-ordering heuristics, symmetry breaking, nogood learning, and restart strategies. The work reported here uses a popular, efficient combination: systematic backtracking with weight-based variable-ordering heuristics and restart. Under this regimen, previous search experience changes the unknown part of the search tree, and the entire search tree's shape may change dramatically after each restart.

Neither RS nor GRE is accurate in such a dynamic environment. For example, consider Knuth's RS estimator on rlfapScens11_f3, a modified real-world radio link frequency assignment problems that drops the three largest values from each domain. RS dramatically overestimated its search-tree size, even after more than 100 probes. Figure 4.6(a) in the next section shows that GRE also heavily overestimates on it at cutoff $c = 1$. Moreover, a single probe with GRE requires exploring a significant part of the search tree to achieve an estimate close to the true size. To achieve $E[T] < 100N$ with a single probe, for example, GRE had to expand 77% of the rlfapScens11_f2 search

tree, 79% for rlfapScens11_f4, and 89% for rlfapScens11_f6. Naive repeated probing with GRE was no better.

Inspection indicated that RS and GRE fail because initially they often pursue a deep, left-heavy tree. Such a path can cause overestimation of the search-tree size. As search learns more accurate weights, however, it better identifies key variables and addresses them near the top of the search tree, so that early search trees on subsequent probes are shallower, and the final search tree is more balanced. Thus, RS and GRE are subject to the same difficulties that led to random probing and restart. This motivated EHP.

4.3 EHP, a stationary estimator

EHP, shown as Algorithm 4.2, executes at most K probes on a CSP. Each probe uses the same variable-ordering heuristic and delivers any learned information (here, variable weights) to the next probe. After each probe, EHP estimates the search-tree size with GRE (line 3). After K iterations of probing, EHP returns its average estimate over an observation window of size w (line 11).

Algorithm 4.2 EHP(c, K, w)

```
1:  for (k=1; k < K; k++) do
2:      Probe with search limit c
3:      Estimate search tree size as E_k
4:      if (k ≥ w) then
5:          E ← {E_{k−w+1}, ..., E_k}
6:      end if
7:      if (isStationary(E)) then
8:          return (1/w) Σ_{i=k−w+1}^{k} E_i
9:      end if
10: end for
11: return average of estimation in E
```

EHP may probe fewer than K times, however. When it detects that the sequence of estimated search-tree sizes has become stationary, EHP stops and returns that estimate (line 8). EHP uses the Augmented Dickey-Fuller (ADF) wide-sense stationary test (line 7), which requires that the first- and second-order moments of the sequence of estimates do not vary with time. Dickey-Fuller tests for a unit root in a time series sample. ADF is a version of Dickey-Fuller that assumes the errors may be serially correlated. (See [77] for further details.)

EHP substantially differs from the random-sampling estimator RS, for which a probe is a pure observation on a static search tree. In contrast, EHP combines sampling for observation and learning to boost constraint search. In particular, EHP naturally embeds its estimating procedure, repeated sampling by GRE, into a sequence of quick probes (i.e., restarts), which drives the search tree anticipated by GRE closer to the real dynamic tree (because they are both produced under similar weights) and enhances the estimation accuracy. The idea of random probing to learn weights to boost constraint search on random CSPs was originally proposed in [31]. This work, however, is the first to combine estimation-oriented sampling and learning-oriented probing in the same framework.

This work implemented EHP within the sequential CSP solver Mistral-1.550 ([8]). Appendix D describes Mistral's parameter configuration for all experiments in this chapter. Experiments to evaluate EHP's performance on a variety of CSP classes and to measure true search effort were executed on an 8 GB Mac Pro with a 2.93 GHz Qued-Core Intel Xeon processor, while Mistral did systematic backtracking with the variable-ordering heuristic *dom/wdeg*, default value ordering, and geometric restart with default factor 1.3333. Experiments for EHP adopted the same configuration but used random

value ordering.

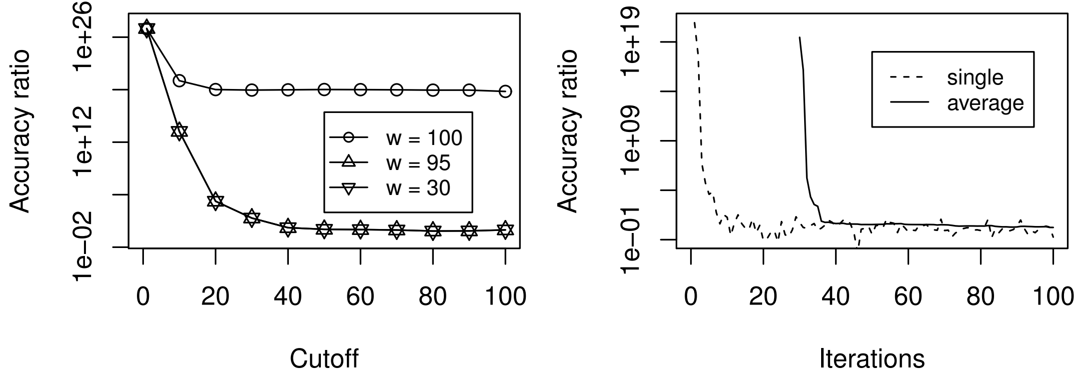


Figure 4.6: The impact of cutoff c and iteration K on EHP's estimation accuracy for the unsatisfiable CSP rlfapScens11_f3. (left) Impact of c, where curves for $w = 95$ and $w = 30$ are nearly identical. (right) Impact of K. Ideal accuracy ratio is 1.

For EHP with $K = 100$ probes and observation window sizes $w = 30$, 95, and 100, Figure 4.6(a) shows how the cutoff c impacts EHP's *accuracy ratio*, the ratio of its estimate to the true search effort. The ideal accuracy ratio is 1, which indicates perfect estimation. When EHP probes with $c = 1$, its behavior is similar to that of RS, which does not learn variable weights. Clearly $c > 1$ reduces the accuracy ratio. Values of w slightly larger than 1 quickly removed the noisy estimate of the first several probes, and thereby reduced estimation error. The dashed curve in Figure 4.6(b) shows how the ratio decreased and then remained stable as the number of probes K increased. The solid line curve in Figure 4.6(b) is the average ratio over the 30 most recent probes, and motivated the choice of $w = 30$ in the remainder of this work.

Table 4.1: Nodes expanded by Mistral per problem class

	os-taillard	rand	Scens11	langford
Mean	94021	1165356	1293121	4917237
STD	465257	2978760	2719731	19128808
Max	2952872	16552914	8448335	120534877

Table 4.2: Comparison of estimation errors for EHP, GRE, GWBE, and RS on 226 CSP instances of four CSP classes.

CSP	Method	ALL			SAT			UNSAT		
		Median	Mean	STD	Median	Mean	STD	Median	Mean	STD
os-taillard	EHP	1.88	2.85×10^{5}	3.03×10^{6}	1.88	4.28×10^{5}	3.71×10^{6}	1.87	5.51	11.84
	GRE	2.63	1.29×10^{40}	1.41×10^{41}	2.65	1.93×10^{40}	1.73×10^{41}	2.55	2.19×10^{25}	1.38×10^{26}
	GWBE	2.63	2.77×10^{26}	2.81×10^{27}	2.65	4.15×10^{26}	3.44×10^{27}	2.55	1.29×10^{21}	8.01×10^{21}
	RS	3.53×10^{10}	1.32×10^{37}	1.42×10^{38}	50.52	3.28×10^{15}	2.71×10^{16}	2.75×10^{32}	3.96×10^{37}	2.46×10^{38}
rand	EHP	2.83	25.77	93.10	4.26	34.90	108.43	1.68	1.69	0.27
	GRE	2.65	33.04	119.60	4.00	44.86	139.27	1.48	1.88	0.74
	GWBE	1.85	15.37	57.91	2.33	20.68	67.57	1.35	1.39	0.28
	RS	4.02	26.38	89.71	6.06	35.40	104.41	1.40	2.59	2.73
scen11	EHP	6.85	14.18	18.68	-	-	-	6.85	14.18	18.68
	GRE	5.48×10^{13}	2.76×10^{20}	9.36×10^{20}	-	-	-	5.48×10^{13}	2.76×10^{20}	9.36×10^{20}
	GWBE	8.46×10^{11}	2.22×10^{17}	7.67×10^{17}	-	-	-	8.46×10^{11}	2.22×10^{17}	7.67×10^{17}
	RS	9.66×10^{22}	4.16×10^{24}	1.40×10^{25}				9.66×10^{22}	4.16×10^{24}	1.40×10^{25}
langford	EHP	1.20	1.89×10^{4}	1.01×10^{5}	2.19	6.36×10^{4}	1.83×10^{5}	1.09	1.30	0.64
	GRE	1.44	1.96×10^{32}	1.44×10^{33}	2.00	6.60×10^{32}	2.64×10^{33}	1.41	1.72	1.17
	GWBE	1.20	5.77×10^{31}	4.20×10^{32}	2.00	1.91×10^{32}	7.64×10^{32}	1.14	1.26	0.32
	RS	1.65	5.64×10^{3}	2.78×10^{4}	2.88	1.90×10^{4}	4.96×10^{4}	1.50	1.52	0.43

We analyzed predictive accuracy on all problems in four categories at the Fourth International CSP Solver Competition [7] that could be solved by Mistral in six hours or less: 120 os-taillard (open-shop scheduling problems, 80 satisfiable), 40 random (29 satisfiable), 12 scen11 (modified real-world radio link frequency assignment problems, RLFAPs, all unsatisfiable), and 54 langford (Langford pairing problems, 16 satisfiable). Table 4.1 shows the search effort Mistral required to solve them.

We compared the predictive accuracy of EHP on those problems to three other approaches: GRE, generalized WBE (GWBE), and RS. *GWBE* generalizes WBE similarly to the way GRE generalizes RE. RS is similar to EHP with $c = 1$, but RS does not learn weights on failures, and it estimates as an average across all samples. EHP had at most $K = 200$ probes per instance, with $w = 30$ and $c = 50$, but stopped when its estimate became stationary ($p < 0.05$). (This significance test was considered valid only when the ratio of the maximum and minimum estimates observed within the window was less than 10^3.) For a fair comparison, GRE and GWBE were allowed to estimate on at most 10^4 failures, while RS could sample as often as it chose as long as its total number of failures was no more than 10^4.

The results appear in Table 4.2, where each entry is the error in the method's estimate compared to the true value ac-curate. (To penalize both overestimation and underestimation fairly, estimation error here is the larger of the ratios estimate/accurate and accurate/estimate.) EHP had a clear advantage over GRE, GWBE, and RS. Although EHP did not always minimize the error, it had the most stable performance across problem classes.

Figure 4.7 plots the results for 88 challenging instances. Excluded were the 138 solved by EHP in its first 29 iterations, where EHP has trivial perfect estimation, and

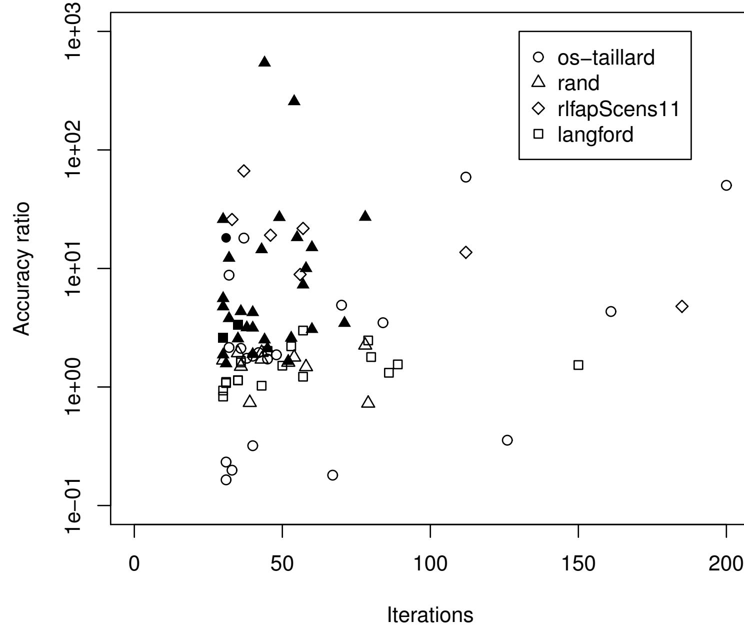


Figure 4.7: Ratio of EHP's estimate to the true search effort. Symbols for satisfiable problems are filled; symbols for unsatisfiable problems are unfilled. The legend at the upper-right corner shows the problem classes.

the 4 with ratio greater than 10^3 (2 os-taillard and 2 langford problems). Most instances lie on the lower-left in Figure 4.7, which indicates that EHP required less than 50 probes to provide a satisfactory estimate of the true search effort. Note that EHP is somewhat less accurate on satisfiable instances in Figure 4.7, and never underestimates on them. This is probably because, for satisfiable problems, search stops at the first solution, without consideration of the full search tree and all solutions.

4.4 Spread*, an advanced adaptive parallel paradigm

This section introduces Spread*, an enhanced version of Spread that incorporates EHP for more intelligent control. To exploit information provided by EHP (specifically, estimated search effort on subproblems), Spread* has a different queuing system and a modified partitioning mechanism.

SPREAD* executes its portfolio phase just the way SPREAD does (Algorithm 3.1). During its splitting phase, however, SPREAD* maintains two queues: a FIFO queue Q_1 for subproblems ready for distribution, and a priority queue Q_2 for subproblems returned unsolved. When SPREAD* assigns a subproblem to a worker, the worker first applies EHP to it and then searches on its subproblem as usual. If the worker does not solve the subproblem during EHP, it then starts standard search with the learned weights themselves and records the search effort estimated by EHP. This estimate is ignored if the worker finally solves the subproblem. When a worker returns a subproblem unsolved, however, it also returns its recorded estimated search effort. Q_2 orders unsolved subproblems in descending order by estimated effort. Whenever the length of Q_1 drops below threshold τ, the manager extracts the lead subproblem from Q_2, partitions it, and enqueues the new subproblems in Q_1.

SPREAD* determines the number of splitting variables ξ for a subproblems more flexibly than SPREAD does. Both for the queue in SPREAD and for current queue Q_1 with length $L < \tau$ and maximum capacity C in SPREAD*, ξ is bounded by M:

$$\xi \leq M = \lfloor \log_2(C - L) \rfloor \tag{4.7}$$

SPREAD, however, uses $M = \xi$, that is, it fills as much of the available space as possible. In contrast, let e_i be the estimated search effort on the ith subproblem in Q_2, SPREAD* uses

$$\xi = \left\lfloor \log_2 \left(\frac{e_1}{\sum_i e_i} (2^M - 2) + 2 \right) \right\rfloor \tag{4.8}$$

This new control policy in SPREAD* reflects our intuition about subproblem partitioning and ordering for parallel constraint search: partition on difficult subproblems early and aggressively. Specifically, if the first estimate is considerably larger than the

Table 4.3: Comparison of SPREAD*, and SPREAD-D and SPREAD-I on 16 processors. † denotes unsatisfiable problems.

CSPs	SPREAD*		SPREAD-D		SPREAD-I	
	Time(#TO)	#subps	Time(#TO)	#subps	Idle ratio	#subps
	Idle ratio	#equs	Idle ratio	#equs	Idle ratio	#equs
*scen*11_f1†	**1265.60**	1536	1906.15	740	1938.24	32
9529.18	0.03	179	0.09	1048	0.23	32
*scen*11_f2†	592.10	890	**500.73**	513	942.59	32
3574.69	0.06	115	0.11	614	0.42	32
*scen*11_f3†	233.37	543	**156.85**	330	305.40	32
835.19	0.06	45	0.14	357	0.32	32
langford-2-14†	533.76	443	524.89	463	778.95	32
4821.05	0.07	43	0.14	500	0.55	32
langford-4-17†	491.17	537	537.49	427	**339.24**	32
1085.92	0.06	64	0.11	532	0.54	32
langford-4-18†	2996.05	1101	3696.52	742	**1401.32**	32
4897.52	0.01	312	0.07	1066	0.49	32
js-20-15-1	**312.80**	126	2134.28(3)	48	3853.16(16)	32
3222.77	0.00	79	0.00	308	0.00	32
js-20-15-4	**228.22**	118	788.03(4)	48	1652.80(10)	32
678.45	0.00	71	0.00	227	0.00	32
js-15-100-5	**3957.13**(13)	363	5318.20(25)	53	5857.10(17)	32
>21600	0.00	307	0.00	475	0.00	32
*e0ddr*1-2†	**632.24**	1381	3210.71	805	1602.59	32
5374.02	0.08	120	0.08	1352	0.36	32
*e0ddr*1-4†	**147.41**	493	428.68	410	240.03	32
2070.05	0.06	33	0.08	559	0.23	32
*e0ddr*1-9†	**368.47**	1115	1778.01	657	1436.53	32
3040.55	0.07	79	0.09	1056	0.33	32

others in Q_2, by equation 4.8, when e_1 is larger compared to other estimates, the manager will uses ξ of value close to M (i.e., partitions the lead subproblem into more subproblems). Otherwise, if the first estimate is comparable to the others in Q_2 and Q_2 is long, it only partitions it into two subproblems.

Tables 4.3, 4.4, and 4.5 evaluate the impact of EHP on SPREAD with a dozen challenging CSPs that cover a broad spectrum of CSP classes and constraint types, taken from recent CSP solver competitions [7]. In a naive parallel race of 63 copies of Mistral with random seeds for 6 hours, no problem was solved in less than 10 minutes. (The best race time appears below each problem's name in these tables.) The experimental platform was a Cray XE6m on 160 dual-socket compute nodes, each with two 8-core

2.3 GHz AMD Magny-Cours processors. The time limit for the portfolio phase was 10 seconds.

We tested each problem on 16, 32, and 64 processors, and compared SPREAD* to SPREAD under two cutoff control policies: SPREAD-D and SPREAD-I. SPREAD-D is the dynamic version of SPREAD, as introduced in Section 3.2.3. For the experiments here, SPREAD-D calculates cutoff $c = (1.5)^{\xi} \cdot base$, where ξ is the number of splitting variables that splits the subproblem, and *base* is the average search tree size produced by all workers during the portfolio phase. SPREAD-D also doubles the cutoff on a subproblem if the queue is too full to split it. SPREAD-I uses an infinite cutoff for the splitting phase, that is, it never returns a subproblem unsolved. SPREAD-I is an aggressive strategy that may succeed on balanced search trees but suffers on unbalanced ones. SPREAD* used a configuration similar to SPREAD-D, but executed its new control strategy ($K = 50$, $c = 30$, $w = 30$ for EHP) before every search in the splitting phase, with 15, 31, or 63 workers, threshold $\tau = \min\{n : n \geq 2r\}$, and queue capacity $C = 2\tau$. The time limit was 3 hours per problem.

Tables 4.3, 4.4, and 4.5 report the results for 16, 32, and 64 processors, respectively. There are four values in each cell: wall-clock runtime in seconds, number of subproblems generated by IBP (#subps), idle ratio (average ratio of worker idle time and overall splitting phase time), and number of subproblems ever enqueued (#enqs, into the FIFO queue for SPREAD-D or SPREAD-I, into Q_2 for SPREAD*). (For SPREAD-D or SPREAD-I, #enqs includes subproblems returned to the manager and so is always larger. For SPREAD*, however, #enqs is only those returned for repartitioning.) Each call is the average of 30 runs with a random seed on the same problem; a number in parentheses counts runs that did not solve the problem. Boldface data is statistically significantly

Table 4.4: Comparison of SPREAD*, and SPREAD-D and SPREAD-I on 32 processors. † denotes unsatisfiable problems. † denotes unsatisfiable problems.

CSPs	SPREAD*		SPREAD-D		SPREAD-I	
	Time(#TO)	#subps	Time(#TO)	#subps	Idle ratio	#subps
	Idle ratio	#equs	Idle ratio	#equs	Idle ratio	#equs
*scen*11_f1†	567.93	2377	**380.15**	1228	806.69	64
9529.18	0.06	200	0.15	1380	0.49	64
*scen*11_f2†	245.34	1363	**167.27**	1026	319.12	64
3574.69	0.11	71	0.20	1094	0.42	64
*scen*11_f3†	**102.05**	721	115.03	819	263.15	64
835.19	0.13	28	0.32	847	0.46	64
langford-2-14†	265.23	679	247.63	796	718.08	64
4821.05	0.19	20	0.29	810	0.78	64
langford-4-17†	**261.78**	953	300.01	1006	283.17	64
1085.92	0.09	57	0.24	1054	0.70	64
langford-4-18†	1462.65	1721	1685.25	1602	**1115.89**	64
4897.52	0.03	284	0.12	1881	0.65	64
js-20-15-1	**105.12**	178	2475.48	96	2401.06(3)	64
3222.77	0.00	83	0.00	585	0.00	64
js-20-15-4	**100.02**	177	922.26	96	1242.36(2)	64
678.45	0.00	82	0.00	432	0.00	64
js-15-100-5	**4167.42**(7)	655	4982.02(25)	122	4641.97(16)	64
>21600	0.00	533	0.00	957	0.00	64
*e0ddr*1-2†	**346.36**	2573	1226.66	1775	1000.27	64
5374.02	0.19	124	0.15	2284	0.54	64
*e0ddr*1-4†	**91.26**	1096	200.45	883	177.22	64
2070.05	0.12	55	0.15	1026	0.44	64
*e0ddr*1-9†	**190.18**	1898	700.23	1531	833.25	64
3040.55	0.14	73	0.16	1871	0.49	64

better than the other two approaches in a two-tailed heteroscedastic t-test ($p < 0.05$). It is also shown that all versions of SPREAD consistently improved their performance with more processors, which implies SPREAD's promise to scale to large number of processors.

The interaction between EHP and SPREAD* is complex. Recall that SPREAD itself already devotes considerable effort to workload balance. To outperform SPREAD, SPREAD* must balance the workload still better, that is, the extra benefit from EHP must justify the overhead it incurs. Table 4.6 shows the overhead introduced by EHP is non-trivial. This is an important reason why EHP cannot guarantee higher performance in SPREAD*. To confirm that it is EHP, and not just the two-queue structure with subproblem probing that provides its power, we also tested SPREAD-R, an ablated version

Table 4.5: Comparison of SPREAD*, and SPREAD-D and SPREAD-I on 64 processors. † denotes unsatisfiable problems. † denotes unsatisfiable problems.

CSPs	SPREAD*		SPREAD-D		SPREAD-I	
	Time(#TO) Idle ratio	#subps #equs	Time(#TO) Idle ratio	#subps #equs	Idle ratio Idle ratio	#subps #equs
*scen*11_f1†	200.34	2925	196.79	2916	419.90	128
9529.18	0.09	110	0.21	3091	0.50	128
*scen*11_f2†	112.96	2185	**88.61**	2481	211.32	128
3574.69	0.14	67	0.25	2543	0.54	128
*scen*11_f3†	**72.03**	1650	77.95	1675	146.98	128
835.19	0.19	48	0.43	1696	0.49	128
langford-2-14†	127.51	1168	133.02	2275	490.37	128
4821.05	0.22	17	0.42	2291	0.82	128
langford-4-17†	**163.85**	2103	185.05	2310	213.16	128
1085.92	0.16	56	0.30	2363	0.79	128
langford-4-18†	771.20	2925	763.47	3532	987.43	128
4897.52	0.07	261	0.18	3728	0.80	128
js-20-15-1	**67.10**	326	580.84	196	790.77	128
3222.77	0.00	134	0.00	677	0.00	128
js-20-15-4	**57.67**	324	417.07	196	517.91	128
678.45	0.00	133	0.00	614	0.00	128
js-15-100-5	**3440.11**(3)	1087	2875.52(24)	279	3956.91(12)	128
>21600	0.00	815	0.00	1370	0.00	128
*e0ddr*1-2†	**211.98**	5398	506.55	3762	504.62	128
5374.02	0.29	137	0.20	4146	0.66	128
*e0ddr*1-4†	**63.35**	2400	123.46	1589	105.47	128
2070.05	0.14	84	0.14	1735	0.57	128
*e0ddr*1-9†	**132.04**	3592	326.04	2955	434.92	128
3040.55	0.26	93	0.23	3217	0.64	128

of SPREAD* that replaced EHP's estimates with random numbers. Table 4.7 shows the average percentage of saved search time of SPREAD* with respect to SPREAD-R. The results in Table 4.7 indicate that EHP's benefit depends on the problem class. SPREAD* performed better on more than 80% of the problems tested.

This chapter has introduced several parallelization methods and evaluated their performance. It has discussed theoretical bounds for the bias for GRE, a search tree

Table 4.6: Average part of runtime devoted to probing in SPREAD*

	scen11	langford	js-taillard	e0ddr
16 processors	20.54%	2.40%	23.13%	11.54%
32 processors	26.90%	1.50%	24.52%	15.53%
64 processors	30.13%	3.79%	29.78%	17.53%

Table 4.7: Average percentage of search time saved by SPREAD* compared to SPREAD-R on 16, 32, and 64 processors over 30 runs. † denotes unsatisfiable problems. (Because *js*-15-100-5 was not solved on every run, its entry here is the difference between the frequency with which it was solved in 30 runs by SPREAD* and the frequency with which it was solved by SPREAD-R.)

	16 proc	32 proc	64 proc
*scen*11_f1†	0.74%	-6.81%	8.11%
*scen*11_f2†	-5.23%	1.74%	0.20%
*scen*11_f3†	0.09%	4.61%	1.56%
langford-2-14†	3.83%	-7.78%	5.81%
langford-4-17†	11.74%	6.05%	20.15%
langford-4-18†	-3.36%	-8.96%	20.26%
js-20-15-1	14.98%	32.89%	-9.12%
js-20-15-4	-46.87%	11.38%	5.07%
js-15-100-5	6.67%	13.33%	6.67%
*e0ddr*1-2†	8.70%	5.92%	5.29%
*e0ddr*1-4†	18.78%	5.05%	3.90%
*e0ddr*1-9†	8.58%	10.91%	4.45%

size estimator generalized from RE. It has revised GRE for EHP where a learned search heuristic changes the search trees, and demonstrated EHP's accuracy on a variety of problem instances. This chapter then incorporated EHP into SPREAD as SPREAD*. The revised paradigm was shown to be a more effective way to balance processor workload and improve search performance. The next chapter discusses some interesting issues raised by the approaches in previous chapters, and outlines future work.

5

Conclusions

This thesis develops learning and parallelization approaches to support and improve the performance of modern constraint solvers on multiple processors. For convenience and portability, these approaches do not share memory among processors and require at most moderate modification to solvers. For CSPs responsive to at least one candidate away from a set of available CSP solvers, the portfolio-based approach RSR-WG that constructs parallel schedules for those solvers is the first choice. When a CSP is challenging for all the candidate solvers, however, the hybrid paradigms SPREAD and SPREAD* offer better parallelization. Experimental results presented here show promise for the parallelization of solvers in a non-shared-memory environment. In addition, the meta-predictor MAMC, inspired by the learning mechanism in RSR-WG, demonstrates its promise for bioinformatics applications like protein-ligand docking. The next section summarize the contributions of this work. Subsequent sections discuss more about those approaches and describe the future work.

5.1 Summary of the contributions

This thesis begins with portfolio-based approaches that leverage a variety of solvers without interference in their internal structure. WG combines CBR and greedy algorithm to construct non-parallel algorithm portfolios for the first time. This thesis then formulates parallel algorithm portfolio construction as an integer-programming problem and proves that the optimal algorithm portfolio problem is NP-complete. Given the computational hardness of the construction of optimal algorithm portfolios, this thesis proposes RSR-WG and COM for parallel algorithm portfolio construction. RSR-WG is essentially greedy; it generalizes WG for parallel processors based on the properties of the optimal solution to the inherent IP problem.

In contrast, COM is a complete method based on problem reformulation and exploitation of pre-existing IP solvers. Specifically, this work reformulates parallel algorithm construction as K-BMCP. Based on this formulation, COM solves the parallel construction problem with an IP solver (here, SCIP 2.1.0), and uses heuristics MRF or MWNF to convert the resultant portfolios to parallel schedules. Empirical results show that both RSR-WG and COM are promising. In particular, with only a few processors, RSR-WG achieves performance close to that of an oracle. The success of these approaches inspired generalization of their CBR mechanism to MAMC, and then exploited it for PLD. Experimental results show that MAMC can improve the prediction accuracy of PLD, particularly on cases of high prediction confidence.

To exploit the power of search space splitting, this work introduces IBP for balanced workload partitioning of CSPs, and executes it recursively (RS-IBP) to direct computing resource to the difficult subproblems. SPREAD, a hybrid adaptive paradigm,

executes two consecutive phases to harness parallel computation and enhance the performance of an underlying sequential solver. The first, the portfolio phase, executes a naive parallel portfolio to solve responsive CSPs quickly and to learn information for subsequent partitioning. The second, the splitting phase, executes RS-IBP to tackle challenging CSPs. SPREAD, for the first time, uses portfolio-based and splitting-based parallelization for constraint search based on learned weight.

Finally, to order and split subproblems more efficiently, this thesis introduces SPREAD*, an advanced version of SPREAD with a more intelligent control mechanism, one based on search effort estimation by EHP. EHP combines the GRE sampling method for search tree size estimation and weight learning to boost constraint search. Experimental results show that SPREAD* can better balance workload, reduce processors' idle ratio, and further improve the performance of SPREAD. The probabilistic analysis given here for the bias of GRE could eventually lead to bounds for other interesting classes of search trees.

5.2 Discussion

To address challenging CSPs, this study initially sought to generate similar problems but of smaller sizes, learn promising heuristics on them, and then exploit the learned heuristics to solve the original CSPs. This approach, however, was problematic [78]. Trivial perturbation on a class of CSPs can dramatically change their satisfiability, even when the perturbation changes neither the structure of their constraint graphs nor the tightness of their constraints. Further investigation revealed that such perturbation could also change the responsiveness of the problems to heuristics. Given these observations, heuristics learned from problems of smaller sizes would be unlikely to help on

the original CSPs.

This thesis deliberately keeps its learning simple. It does not exploit such machine learning methods as neural networks, support vector machine, Bayesian models, or probabilistic graph models. The reasons are threefold. First, because the goal of this work is to boost constraint search, it prioritizes learning methods that can be efficiently embedded into the parallelization framework to speed search. Complex methods that require significant learning time are of less interest, even if they could achieve better prediction accuracy. Second, CSPs that are challenging in a parallel environment are likely to be too difficult for a sequential solver, so that construction of an adequate training set of CSPs is time-consuming. In addition, for CSPs that undergo unexpected, repeated partitioning, finding representative training examples is dramatically more difficult than doing so in a non-parallel environment. To tackle those challenges, RSR-WG requires only that at least one solver be able to solve each individual CSP within the time limit, while SPREAD and SPREAD* learn without any training examples. Third, to parallelize constraint search effectively and conveniently, this work advocates learning methods that accept the original design of a sequential solver, rather than violate it. In particular, this thesis proposes approaches that learn and deliver critical information to the underlying solver among multiple processors. For example, the portfolio phase of RSR-WG learns variable weights for the splitting phase. As another example, EHP not only estimates search effort, but also influences restarts and weight learning.

The success of RSR-WG relies heavily on the diversity of the performance of its constituent algorithms and the relevance of the extracted features. Typically, algorithm portfolio constructors select their algorithms and features based upon domain-specific knowledge. The reader may, for example, wonder how RSR-WG would perform if it re-

lied on the three solvers CPHydra used in CPAI'08. The difficulty here is that CPHydra included solvers from the 2006 competition, solvers that did not enter CPAI'08, and whose performance was therefore unavailable on the 2008 problems. Although algorithm choice based on domain knowledge and feature selection can further enhance a portfolio's performance, it could also make it vulnerable to overfitting. When the number of features is larger, feature selection can be of considerable benefit to an algorithm portfolio constructor [41, 43]; it is worthy of further exploration.

Spread could define its phases' search limits in number of backtracks, consistency checks, or search tree size. In the portfolio phase, time is the limiting factor, because it forces all the workers to finish at once. In the splitting phase, however, there is a backtrack limit to reduce the likelihood that all the workers will communicate with the manager at once. In addition, to split a search space, Spread uses IBP, which, for generality, assumes no knowledge about problem domains. It could, however, be profitable to exploit domain characteristics. For example, one might partition the large domain of a critical variable into more subproblems, or partition extremely small domains (e.g., binary, as in SAT) with parity constraints [67].

There are many plausible ways to parallelize a solver. One might perturb initial assignments, to vary the top of the search tree, using the same variables with different values. That was tested here as NV, and shown adequate only for the easiest of our test problems. Given the success of restarts and the ability of solvers to learn about contention, one might race the solver against copies of itself with different seeds. That was tested here as NR, and shown only slightly more effective. Given the success of some splitting and portfolio approaches, one might execute random partitioning, or race different solvers against one another. That was tested here as RP and PP, respectively,

and shown adequate for some problems, but significantly less so for more difficult ones.

Besides portfolio-based and splitting-based methods, additional parallelization methods include various workload-balancing mechanisms, such as work stealing [10, 11, 15] and work sharing [14]. The SAT-solver parallelization methods most relevant to SPREAD are described in [12] and [16]. The first passes information from a portfolio phase to a subsequent splitting phase for effective partitioning; the second iteratively partitions a SAT problem with learned clauses. SPREAD combines and extends them to parallelize adaptive search for CSPs. The earliest (static) version of SPREAD appeared as a modification to an explore-and-follow parallelization paradigm in [79]. Subsequent work extended SPREAD to its dynamic version and evaluated both with intensive experiments [80]. The CSP work most relevant to SPREAD is a parallel solver that gradually increases the number of processors during search and partitions problems recursively [68]. SPREAD, however, uses all its processors immediately, learns variable weights, partition on more than one variables at a time, does not retain nogoods, and is applicable to most modern solvers.

SPREAD offers a complete and effective method to parallelize a CSP solver. As a parallelization paradigm, SPREAD makes no assumption about domains or constraint types, and so accepts any class of CSPs that its solver can handle. Its bit-string representation permits programmers to ignore the implementation details of the solver, which significantly simplifies parallelization, and dramatically reduces communication effort. Its portfolio phase solves easy problems quickly, and informs the splitting phase for effective partitioning. By recursively partitioning difficult subproblems with RS-IBP, it gradually allocates more computing cycles to the difficult parts of a problem, and thereby adaptively balances processor workload. Finally, SPREAD provides a natural

way to embed restart policies into an MPI environment without recoding its underlying solver.

Given its assumption that the search tree is static, GRE's assumption that the search tree is also symmetric is plausible for chronological backtracking on unsatisfiable CSPs. It could, however, require adjustment for backjumping, nogood learning, or satisfiable instances. SPREAD* can improve the performance of SPREAD, but machine learning tells us that there is no free lunch. SPREAD* seeks a tradeoff between speedup and overhead, primarily by gathering additional information about search effort estimates and by repeated probing on subproblems. As a result, SPREAD is actually sometimes a better choice than SPREAD*. One possible situation where SPREAD* would be unlikely to outperform SPREAD is when EHP does not provide accurate estimates, due to an inappropriate probing limit L or probing iteration limit K, or an intractable problem. Another possibility is that EHP does estimate accurately, but SPREAD* receives little benefit when it adjusts its order and partitioning of subproblems, so that EHP simply consumed time. (This was observed on some random CSPs.)

Learning and parallelization can boost constraint search, but their benefit is not guaranteed. For example, search space splitting, particularly inappropriate splitting, can achieve marginal performance improvement, and sometimes can even lead to performance degeneration. (See the comparison of SPREAD and NWSPREAD in Figure 3.4, where NWSPREAD left many CSPs unsolved.) It is even more surprising that using node weights does not always help. For example, for some Golomb Ruler CSPs, a version of SPREAD (or SPREAD*) that does not initialize variable weights of subproblems with weights from the portfolio phase (but does use them to choose splitting variables) outperforms the original SPREAD and SPREAD*. This behavior is caused by the Golomb

Ruler models that the underlying solver Mistral constructs; they have thousands of variables and the resultant search trees are extremely biased. Most subproblems there can be solved quickly ($\leq$ 0.1s), with or without learned variable weights, but the delivery of the learned weights back to the manager requires several seconds on average for each subproblem on our experimental platform. This issue should be exacerbated in asynchronous networks with high latency. Thus it is important to consider both the practices of the candidate solvers and the experimental platform during learning and parallelization.

5.3 Future work

Portfolio-based methods: In practice, many algorithms may perform differently on the same problem in different runs, but still exhibit a certain level of consistency [40]. Indeed, in (sequential) CSP solver competitions, solvers typically fix their parameter values and introduce relatively little randomness to achieve stable performance. In that case, with coarse granularity (e.g., $B = 10$), a solver's performance is nearly deterministic. Greater randomness, however, could change solvers' performance dramatically, and thereby potentially benefit parallel constraint solving. A generalization of RSR-WG could handle such behavior. On the other hand, automatic parameter tuning could introduce substantial diversity, and should fare well in algorithm portfolios [66]. Specifically, one may view different configurations of a solver as different algorithms, and thereby combine parameter tuning and an algorithm portfolio in the same framework.

Hybrid methods: SPREAD currently uses a static duration for the portfolio phase. The information collected during the portfolio phase, however, can be misleading. This

is more likely when the duration for the portfolio phase is not adequate. A typical example comes from the *queens-knights* (QK) problems. Although the contention in QK lies with the knights, weight-based variable-ordering heuristics prefer the queens variables at the beginning of search, when weight-based heuristics (e.g., $dom/wdeg$) are close to those not based on weights (e.g., dom/deg). The search usually requires more time to learn where the true contention is. Future work could explore adaptive methods that dynamically choose the duration for the portfolio phase. Moreover, nogood learning (as clause learning) has proved crucial in SAT, but has thus far received relatively little attention in parallel CSP solvers, including SPREAD. Future work could explore combinations of adaptive splitting variable selection with nogood learning to avoid the loss of useful information.

Case-based meta-prediction: MAMC is a case-based meta-predictor, applied in this work to construct algorithm portfolios and to improve compound virtual screening using PLD. Given a domain-specific similarity metric that compensates for individual predictors by its focus on additional relevant features, MAMC is applicable to other bioinformatics and chemoinformatics problems. Prospective areas include two- and three-dimensional protein structure prediction, protein-protein interaction, protein-nucleotide interaction, disease-causing mutation, and the functional roles of non-coding DNA.

Parallelization of constraint search is an emerging research area. This thesis demonstrates that learning can boost parallelization and thereby accelerate CSP solution. The approaches discussed in this thesis provide insight on how to parallelize constraint search more efficiently and effectively.

Appendix A

Available CSP solvers in the Fourth International CSP Solver Competition

Solver Name	Submitter	Coding Language
Abscon 112v4 AC	Christophe Lecoutre and Sebastien Tabary	Java
Abscon 112v4 ESAC	Christophe Lecoutre and Sebastien Tabary	Java
bpsolver 09	Neng-Fa Zhou	B-Prolog
Choco2.1.1 2009-06-10	The Choco Team	Java
Choco2.1.1b 2009-07-16	The Choco Team	Java
Concrete 2009-07-14	Julien Vion	Java
Concrete DC 2009-07-14	Julien Vion	Java
Conquer 2009-07-10	Marc van Dongen	Java
Mistral 1.550	Emmanuel Hebrard	C++
pcs 0.3.2	Michael Veksler	C++
pcs-restart 0.3.2	Michael Veksler	C++
SAT4J CSP 2.1.1	Daniel Le Berre	Java
Sugar v1.14.6+minisat	Naoyuki Tamura	Java
Sugar v1.14.6+picosat	Naoyuki Tamura	Java

Appendix B

Features used in Mistral 1.550

Number of constant variables (log2)	Percentage of global constraints
Number of Boolean variables (log2)	Percentage of GAC predicates
Number of range variables (log2)	Percentage of decomposition predicates
Number of bit variables (log2)	Square root of average domain size
Number of list variables (log2)	Square root of maximum domain size
Number of total values (log2)	100 * average domain continuity
Number of extra Boolean variables (log2)	100 * minimum domain continuity
Number of extra range variables (log2)	Number of Alldifferent constraints
Number of extra domains variables (log2)	Ratio of binary extensional constraints
Number of extra values (log2)	Ratio of (3,4)-ary extensional constraints
Number of constraints (log2)	Ratio of n-ary extensional constraints (n>4)
Number of searched variables in probing (log2)	Ratio of Alldifferent constraints
Average edge weight in probing (log2)	Ratio of Weightsum to all Alldifferent constraints
STD of edge weights in probing (log2)	Ratio of Element to all Alldifferent constraints
Search tree size in probing (log2)	Ratio of Cumulative to all Alldifferent constraints
Number of checks in probing (log2)	Average predicate shape
Maximum constraint arity	Average predicate size
Percentage of extensional constraints	Average predicate arity

Appendix C

Configuration of Mistral-1.331 for experiments in Chapter 3

Variable-ordering heuristic	dom/wldeg
Restart policy	dynamic (portfolio phase); no (splitting phase)
Geometry factor	1.3333 (portfolio phase); 1.5 (splitting phase)
restart based	dynamic (portfolio phase); dynamic (splitting phase)
Randomize	1 (randomize)
Random Seed	randomly generated
Time limit	10 or 100 sec (portfolio phase); no limit (splitting phase)
Node limit	no limit (portfolio phase); dynamic (splitting phase)

Appendix D

Configuration of Mistral-1.550 for experiments in Chapter 4

Variable-ordering heuristic	dom/wdeg
Restart policy	dynamic (portfolio phase); no (splitting phase)
Geometry factor	1.3333 (portfolio phase); 1.5 (splitting phase)
restart based	dynamic (portfolio phase); dynamic (splitting phase)
Randomize	1 (randomize)
Random Seed	randomly generated
Time limit	10 sec (portfolio phase); no limit (splitting phase)
Node limit	no limit (portfolio phase); dynamic (splitting phase)

Glossary

RPCPHydra Randomized Parallel CPHydra. 37

Spread Search by Probing and REcursive Adaptive Domain-splitting. 4

Spread* enhanced Spread with intelligent subproblem splitting based on EHP. 5

AC Arc Consistency. 10

BP Bisection Partitioning. 57

CBR Case-Based Reasoning. 19

COM Integer-programming-based complete algorithm portfolio constructor. 32

CSP Constraint Satisfaction Problem (it means either a class of constraint satisfaction problems or an instance of a particular problem class). 1

EHP Estimation by Heuristic Probing. 5

GRE Generalized recursive estimator. 80

IBP Iterative Bisection Partitioning. 4

MAC Maintain Arc Consistency. 10

MAMC Multi-Agent Multi-Case-based reasoning. 4

PLD Protein-Ligand Docking. 45

RE Recursive estimation. 78

ROC Receiver Operating Characteristic. 49

RPWG Randomized Parallel WG. 26, 37

RSR-WG WG with Retain, Spread, and Return. 4, 27

SAT Boolean satisfiability problem. 8

SMT Satisfiability Modulo Theories. 8

SSS Search Space Splitting, a parallelization method that partitions the search space of a CSP. 56

WBE Weighted backtrack estimation. 78

WG Weighted Greedy, a scheduler for non-parallel algorithm portfolios based on case-based reasoning and a greedy algorithm. 4

References

[1] M.B. Do and S. Kambhampati. **Planning as Constraint Satisfaction: Solving the Planning Graph by Compiling It into CSP**. *Artificial Intelligence*, **132**:151–182, 2001. 1

[2] G. Simonin, C. Artigues, E. Hebrard, and P. Lopez. **Scheduling Scientific Experiments on the Rosetta/Philae Mission**. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming*, pages 23–37. Springer, 2012. 1

[3] P. Schaus, J.-C. Régin, R.V. Schaeren, W. Dullaert, and B. Raa. **Cardinality Reasoning for Bin-Packing Constraint: Application to a Tank Allocation Problem**. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming*, pages 815–822. Springer, 2012. 1

[4] S. A. Cook. **The complexity of theorem proving procedures**. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, 1971. ACM Press. 1

[5] **Minizinc Challenge 2012 results**. `http://www.g12.cs.mu.oz.au/minizinc/challenge2012/results2012.html`. 1

[6] **CSP Competition (CPAI'08)**. `http://www.cril.univ-artois.fr/CPAI08/`. 1, 14, 19, 34, 37, 67, 68

[7] **CSP Competition (CSC'09)**. `http://www.cril.univ-artois.fr/CPAI09/`. 1, 19, 67, 93, 96

[8] **Mistral**. `http://4c.ucc.ie/~ehebrard/Software.html`. 2, 35, 90

[9] **Gecode**. `http://www.emn.fr/z-info/choco-solver/`. 2

[10] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. **Confidence-based Work Stealing in Parallel Constraint Programming**. In *Principles and Practices of Constraint Programming*, pages 226–241, 2009. 3, 57, 106

[11] Laurent Michel, Andrew See, and Pascal Van Hentenryck. **Parallelizing Constraint Programs Transparently**. In *Principles and Practices of Constraint Programming*, pages 514–528, 2007. 3, 106

[12] Ruben Martins, Vasco Manquinho, and Inês Lynce. **Improving Search Space Splitting for Parallel SAT Solving**. In *Twenty-second International Conference on Tools with Artificial Intelligence*, pages 336–343, Arras, France, 2010. 3, 57, 106

[13] Hantao Zhang, Maria Paola Bonacina, Maria Paola, Bonacina, and Jieh Hsiang. **PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems**. *Journal of Symbolic Computation*, **21**:543–560, 1996. 3, 57

[14] F. Xie and A. Davenport. **Massively parallel constraint programming for supercom- puters: Challenges and initial results**. In *Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2010)*, pages 334–338, 2010. 3, 106

[15] Tobias Schubert, Matthew D. T. Lewis, and Bernd Becker. **PaMiraXT: Parallel SAT Solving with Threads and Message Passing**. *Journal of Satisfiability, Boolean Modeling and Computation*, **6**:203–222, 2009. 3, 106

[16] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. **Grid-based SAT solving with iterative partitioning and clause learning**. In Jimmy Ho-Man Lee, editor, *the Proceedings of CP 2011*, **Lecture Notes in Computer Science 6876**, Perugia, Italy, 2011. Springer. 3, 57, 106

[17] P. Prosser, K. Stergiou, and T. Walsh. **Singleton Consistencies**. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 353–368. Springer, 2000. 9, 13

[18] E.C. Sabin, D. Freuder. **Contradicting Conventional Wisdom in Constraint Satisfaction**. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 125–129. Springer, 1994. 9, 10

[19] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003. 10

[20] Philippe Jégou and Cyril Terrrioux. **A new filtering based on decomposition of constraint sub-networks**. In *22th International Conference on Tools with Artificial Integlligence*, pages 263–270, 2010. 10

[21] Christian Bessière and Jean-Charles Régin. **MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems**. In *Second International Conference on Principles and Practice of Constraint Programming*, **LNCS**, pages 61–75. Springer, 1996. 10

[22] Robert M. Haralick and Gordon L. Elliott. **Increasing tree search efficiency for constraint satisfaction problems**. *Artificial Integlligence*, **14**(3):263–313, 1980. 10

[23] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. **Boosting systematic search by weighting constraints**. In *Sixteenth European Conference on Artificial Intelligence*, pages 146–149. IOS Press, 2004. 11

[24] Philippe Refalo. **Impact-Based Search Strategies for Constraint Programming**. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, **LNCS**, pages 557–571, Toronto, Canada, 2004. Springer. 11

[25] L. Michel and P. Hentenryck. **Activity-Based Search for Constraint Satisfaction Problems**. In *CPAIOR-2012*, pages 228–243. Springer-Verlag, 2012. 11

[26] R. Dechter and J. Pearl. **Network-based heuristics for constraint-satisfaction problems**. *Artificial Intelligence*, **34**(1):1–38, 1987. 11

[27] Amnon Meisels, Solomon Eyal Shimony, and Gadi Solotorevsky. **Bayes networks for estimating the number of solutions of constraint networks**. *Annals of Mathematics and Artificial Intelligence*, **28**(1-4):169–186, 2000. 11

[28] C. Gomes, B. Selman, and N. Crato. **Heavy-tail distributions in combinatorial search**. In *Third International Conference on Principles and Practice of Constraint Programming*, **LNCS**, pages 121–135, Linz, Austria, 1997. Springer. 12, 56

[29] C. Gomes, B. Selman, N. Crato, and H. Kautz. **Heavy-tailed phenomena in satisfiablity and constraint satisfaction problems**. *Journal of Automated Reasoning*, **24**:67–100, 2000. 12, 56

[30] M. Luby, A. Sinclair, and D. Zuckerman. **Optimal speedup of Las Vegas algorithms**. In *Second Israel Symposium on the Theory of Computing and Systems*, pages 173–180, 1993. 12, 13

[31] Richard J. Wallace and Diarmuid Grimes. **Experimental studies of variable selection strategies based on constraint weights**. *Journal of Algorithms*, **63**(1-3):114–129, 2008. 13, 90

[32] Susan L. Epstein and Xingjian Li. In *Cluster Graphs as Abstractions for Constraint Satisfaction Problems*, 2009. 13

[33] R. Dechter. **Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition**. *Artificial Integlligence*, **41**:273–312, 1990. 13

[34] George Katsirelos and Fahiem Bacchus. **Generalized nogoods in CSPs**. In *20th National Conference on Artificial intelligence*, pages 390–396, Pittsburgh, Pennsylvania, 2005. AAAI Press. 13

[35] A.R. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown. **SATenstein: Automatically Building Local Search SAT Solvers From Components**. In *Twenty-First International Joint Conference on Artificial Intelligence*, pages 517–524, Pasadena, California, USA, 2009. Morgan Kaufmann Publishers Inc. 14, 56

[36] B. Huberman, R. Lukose, and T. Hogg. **An economics approach to hard computational problems**. *Science*, **256**:51–54, 1997. 16

[37] Carla Gomes and Bart Selman. **Algorithm portfolio design: theory vs. practice**. In *Thirteenth Conference On Uncertainty in Artificial Intelligence*, pages 190–197, New Providence, 1997. Morgan Kaufmann. 16

[38] A. Guerri and M. Milano. **Learning techniques for automatic algorithm portfolio selection**. In *Sixteenth European Conference on Artificial Intelligence*, pages 475–479, 2004. 16

[39] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. **Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving**, 2008. 16, 20, 24

[40] Bryan Silverthorn and Risto Miikkulainen. **Latent class models for algorithm portfolio methods**. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 167–172, 2010. 16, 20, 78, 108

[41] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. **Hydra: automatically configuring algorithms for portfolio-based selection**. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 179–184, 2010. 16, 56, 105

[42] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. **SATzilla: portfolio-based algorithm selection for SAT**. *Journal of Artificial Intelligence Research*, **32**(1):565–606, 2008. 16, 20

[43] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry A. Kautz, Bart Selman, and David Maxwell Chickering. **A Bayesian approach to tackling hard computational problems**. In *Seventeenth Conference in Uncertainty in Artificial Intelligence*, pages 235–244. Morgan Kaufmann Publishers Inc., 2001. 16, 19, 78, 105

[44] J. R. Rice. **The algorithm selection algorithm**. *Advances in Computers*, **15**:65–118, 1976. 16

[45] M. Gagliolo and J. Schmidhuber. **Learning dynamic algorithm portfolios**. *Annals of Mathematics and Artificial Intelligence*, **47(3)**:295–328, 2006. 18, 20, 27

[46] Matteo Gagliolo and Jurgen Schmidhuber. **Towards distributed algorithm portoflios**. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 634–643, 2008. 18, 20, 27

[47] Tom Carchrae and J. Christopher Beck. **Low-knowledge algorithm control**. In *Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 49–54, San Jose, California, USA, 2004. AAAI Press / The MIT Press. 18

[48] Tom Carchrae and J. Christopher Beck. **Applying machine learning to low-knowledge control of optimization algorithms**. *Computational Intelligence*, **21(4)**:372–387, 2005. 18

[49] **The SAT 2007 Competition: satcompetition.org/2007/rules07.html**. 19, 69

[50] Agnar Aamodt and Enric Plaza. **Case-based reasoning: Foundational issues, methodological variations, and system approaches**. *AI Communication*, **7**(1):39–59, 1994. 19

[51] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. **Restart schedules for ensembles of problem instances**. In *the Twentysecond National Conference on Artificial Intelligence*, pages 1204–1210, 2007. 20

[52] M. Streeter, D. Golovin, and S.F. Smith. **Combining multiple heuristics online**. In *the Twentysecond National Conference on Artificial Intelligence*, pages 1197–1203, 2007. 24, 26

[53] Alberto Maria Segre, Sean Forman, Giovanni Resta, and Andrew Wildenberg. **Nagging: A scalable fault-tolerant paradigm for distributed search**. *Artificial Intelligence*, **140**(1-2):71–106, 2002. 27

[54] Pascal Vander-Swalmen, Gilles Dequen, and MichaÃńl Krajecki. **A Collaborative Approach for Multi-Threaded SAT Solving**. *International Journal of Parallel Programming*, **37**(3):324–342, 2009. 27

[55] Matthew Streeter and Daniel Golovin. **An Online Algorithm for Maximizaing Submodular Functions**. Technical report, Carnegie Mellon University, 2007. 29

[56] S. Khuller, A. Moss, and J. Naor. **The Budgeted Maximum Coverage Problem**. *Information Processing Letters*, **70**:39–45, 1999. 33

[57] Tobias Achterberg. **SCIP: solving constraint integer programs**. *Mathematical Programming Computation*, **1**(1):1–41, 2009. 41

[58] E. Plaza and L. McGinty. **Distributed Case-Based Reasoning**. 44

[59] D.B. Leake and R. Sooriamurthi. **Automatically Selecting Strategies for Multi-Casebase Reasoning**. In *ECCBR*, pages 204–233. Springer. 44

[60] S.-Y. Huang and X. Zou. **Advances and Challenges in Protein-Ligand Docking**. *International Journal of Molecular Science*, **11**:3016–303, 2010. 45

[61] Z. Zsoldos, D. Reid, A. Simon, B.S. Sadjad, and P.A. Johnson. **Ehits: An Innovative Approach to the Docking and Scoring Function Problems**. *Current Protein and Peptide Science*, **7**:421–435, 2006. 46

[62] O. Trott and A.J. Olson. **Autodock Vina: Improving the Speed and Accuracy of Docking with a New Scoring Function, Efficient Optimization and Multithreading**. *Journal of Computational Chemistry*, **31**:455–461, 2010. 46

[63] G.M. Morris, D.S. Goodsell, R.S. Halliday, R. Huey, W.E. Hart, R.K. Belew, and A.J. Olson. **Automated Docking Using a Lamarckian Genetic Algorithm and Empirical Binding Free Energy Function**. *Journal of Computational Chemistry*, **19**:1639–1662, 1998. 46

[64] M.A. Miteva, W.H. Lee, M.O. Montes, and B.O. Villoutreix. **Fast Structure-Based Virtual Ligand Screening Combining Fred, Dock, and Surflex**. *Journal of Medicinal Chemistry*, **48**:6012–6022, 2005. 46

[65] H. Fukunishi, R. Teramoto, T. Takada, and J. Shimada. **Bootstrap-Based Consensus Scoring Method for Protein-Ligand Docking**. *Journal of Chemical Information and Modeling*, **48**:988–996, 2008. 46

[66] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas StÃijtzle. **ParamILS: An Automatic Algorithm Configuration Framework**. *Journal of Artificial Intelligence Research*, **36**:267–306, 2009. 56, 108

[67] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. **Experiments with massively parallel constraint solving**. In *Twenty-First International Joint Conference on Artificial Intelligence*, pages 443–448, Pasadena, California, USA, 2009. Morgan Kaufmann Publishers Inc. 57, 69, 105

[68] L. Kotthoff and N.C.A. Moore. **Distributed Solving Through Model Splitting**. In *3rd workshop on Techniques for Implementing constraint programming systems*, St. Andrew, Scotland, 2010. 57, 106

[69] Deepak Mehta, Barry O'Sullivan, Luis Quesada, and Nic Wilson. **Search space extraction**. In *Fifteenth International Conference on Principles and Practice of Constraint Programming*, **LNCS**, pages 608–622, Lisbon, Portugal, 2009. Springer. 58

[70] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais. **Diversification and Intensification in Parallel SAT Solving**. In *Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 252–265. Springer, Heidelberg (2010).* 61

[71] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. **Estimating search tree size**. In *the Twenty-First National Conference on Artificial intelligence*, pages 1014–1019, 2006. 78, 80, 86

[72] Richard E. Korf, Michael Reid, and Stefan Edelkamp. **Time Complexity of Iterative-deepening-A***. *Artificial Intelligence*, **27**(1):97–109, 2001. 78

[73] Osman Y. Özaltin, Brady. Hunsaker, and Andrew J. Schaefer. **Predicting the Solution Time of Branch-and-Bound Algorithms for Mixed-Integer Programs**. *INFORMS Journal on Computing*, **23**(3):392–403, 2011. 78

[74] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. **Hierarchical hardness models for SAT**. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 696–711. Springer, 2007. 78

[75] Donald E. Knuth. **Estimating the efficiency of backtrack programs**. *Mathematics of Computation*, **29**(129):121–136, 1975. 78, 83, 84

[76] Pang C. Chen. **Heuristic sampling: a method for predicting the performance of tree searching programs**. *SIAM Journal of Computing*, **21**(2):295–315, 1992. 79

[77] Jeff B. Cromwell, Walter C. Labys, Michael J. Hannan, and Michel Terraza. *Multivariate Tests for Time Series Models.* Sage Publications Inc, 1994. 90

[78] Susan L. Epstein and Xi Yun. **From Unsolvable to Solvable: An Exploration of Simple Changes**. In *AAAI-10 Workshop on Abstraction, Reformulation, and Approximation (WARA-2010)*, pages 20–25, 2010. 103

[79] X. Yun and S. Epstein. **Adaptive Parallelization for Constraint Satisfaction Search**. In *Fifth Annual Symposium on Combinatorial Search*, pages 145–152, Niagara Falls, Canada, 2012. AAAI Press. 106

[80] X. Yun and S. Epstein. **A Hybrid Paradigm for Adaptive Parallel Search**. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming*, pages 720–734, Quebec City, Canada, 2012. Springer. 106